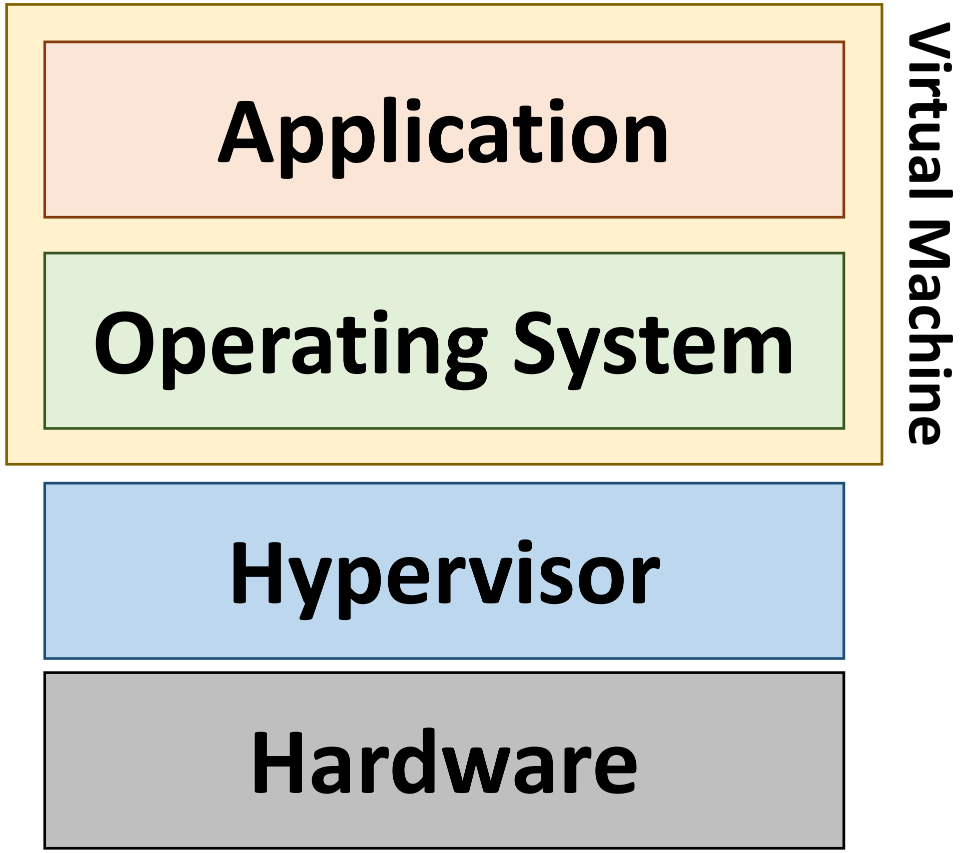
# Containers

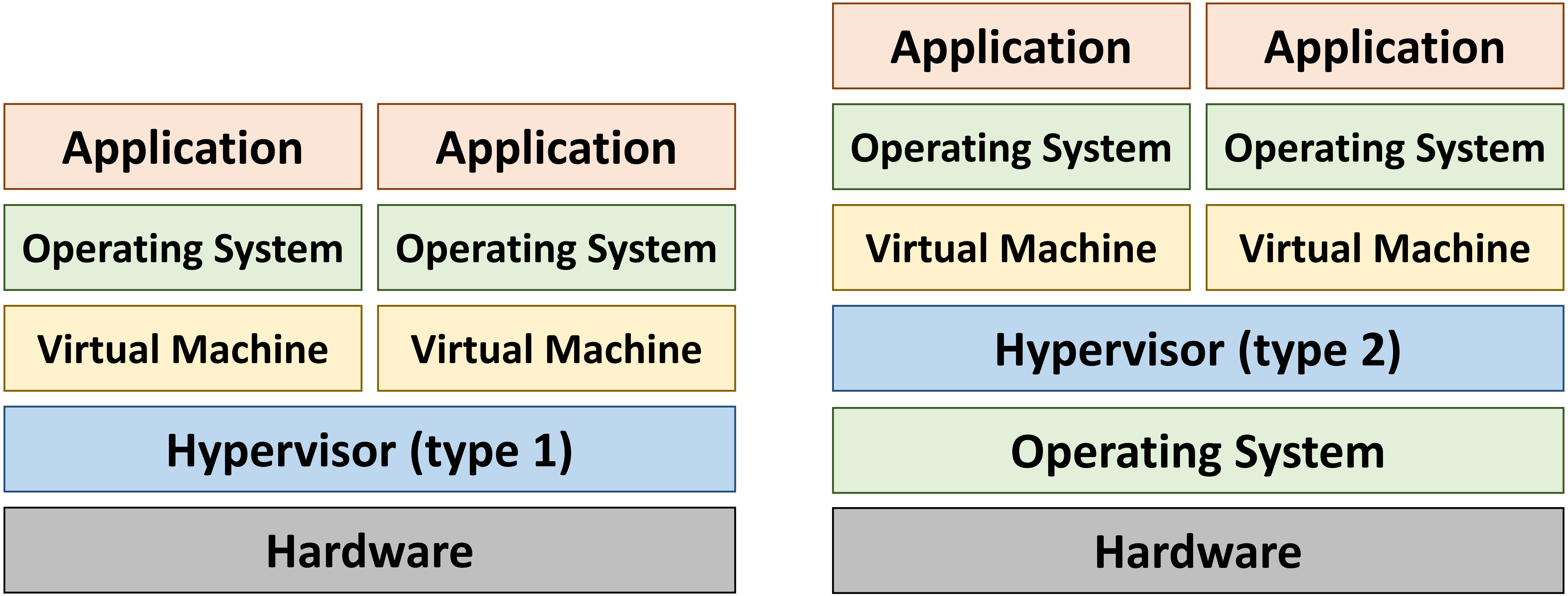
## Docker

# What is a Virtual Machine?

- Software that emulates a server (hardware)
  - Eg. Windows running inside OSX, game console emulators
  - Eg. Virtual Box, VMWare
  - Defines an environment (sandbox) in which applications can run
    - Has its own memory, CPU, network interfaces
- Result is virtualization
  - Allowing multiple virtual machines to run on a single physical server
- Not 'dual boot'



# Types of Virtualization




Type 1

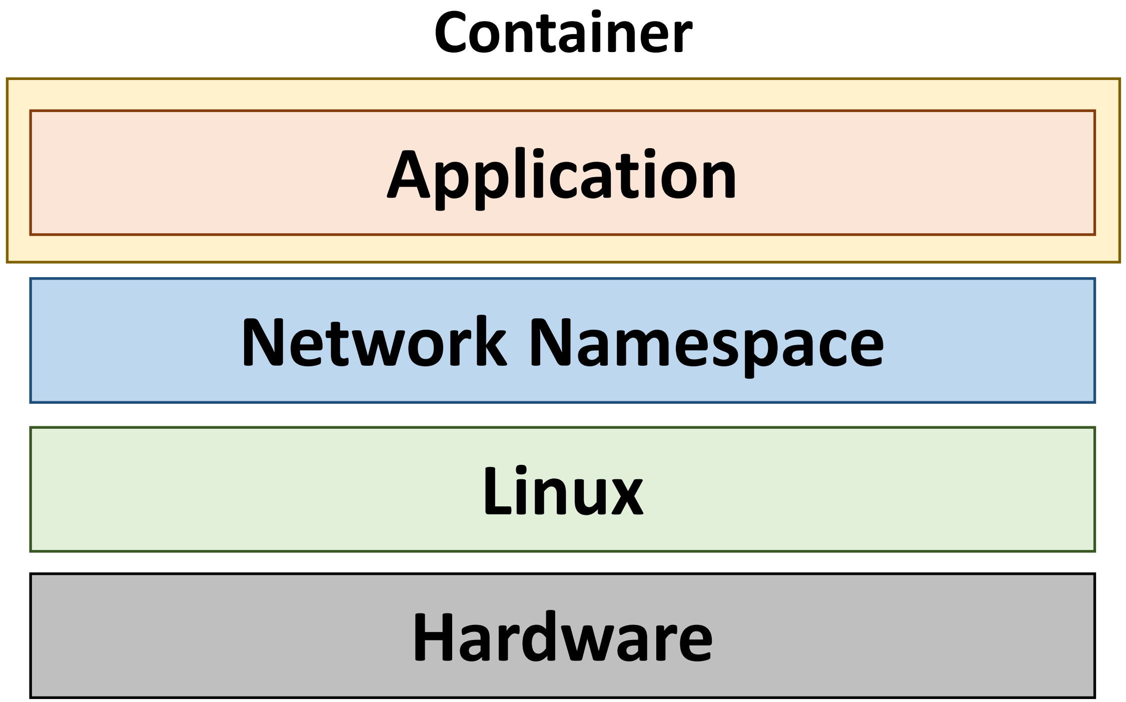
Type 2

# Benefits of Virtualization

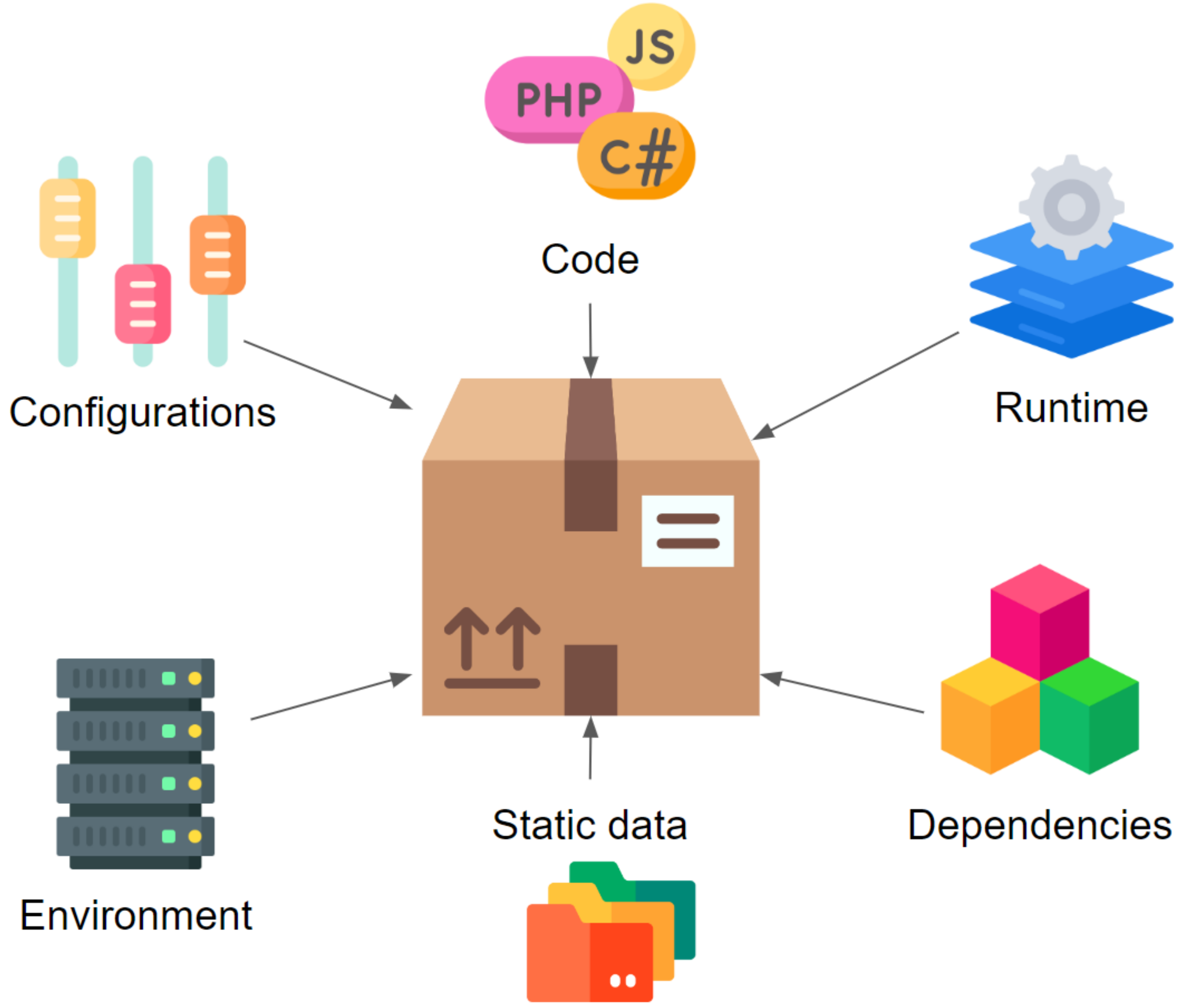
- Improves server efficiency
  - Instead of running one OS on a server, you can now run more workload on a server
- Improves security
  - Multiple applications can run securely because they are effectively isolated from each other by the virtual machine
- Running incompatible versions of application side-by-side
  - Side effect of sandboxing; applications with conflicting requirements or incompatible dependencies can run safely
- Reduces complexity
  - Instead of managing multiple smaller physical servers, these servers are now consolidated into a few bigger servers
- Increase resiliency and high availability
  - Servers with applications can be backed up as a 'file' and spun up at a moment's notice a different server if the existing hardware fails
- Create or recreate any environment on demand
  - Increase operations flexibility; eg supports testing, UAT environment

# Containers

- A container is a standard way of packaging Linux applications, along with all its dependencies, application assets, configuration, etc
- When deployed, these containers runs a independent Linux servers in a Linux host
  - Each Linux servers has its on CPU and memory share, network interface, filesystem, etc
  - Another form of virtualization; virtualizes a Linux environment



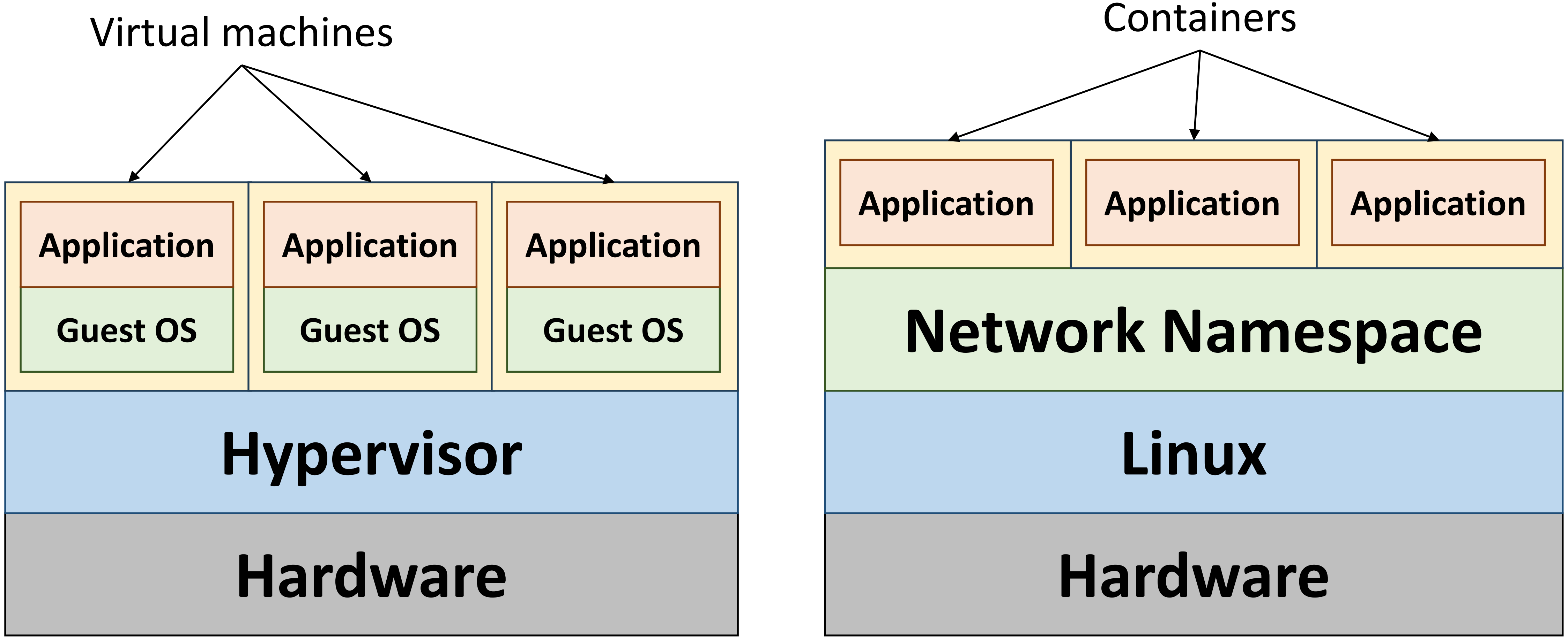
# What is Inside a Container?



# Linux Containers Enabling Technologies

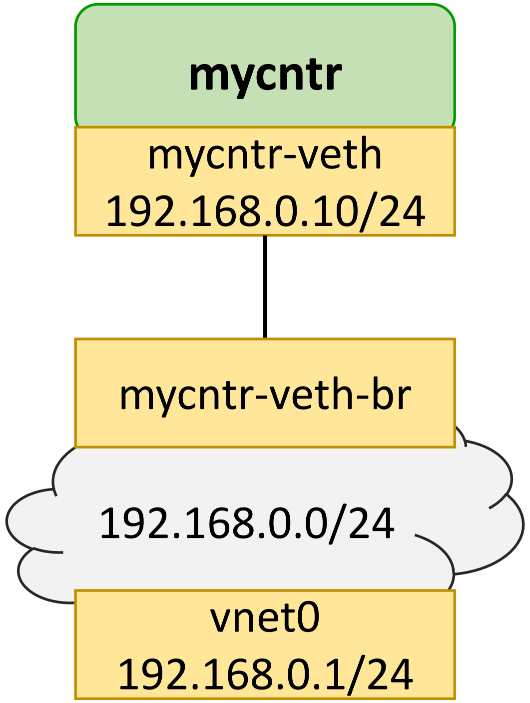
- Namespace
  - Provides isolation
  - Virtualizes Linux resources eg. processes, filesystem, IPC, etc.
- CGroups
  - Allocate resources to the containers
  - Can configure soft and hard limits
- Overlay filesystem
  - Provides a single view of multiple directory by stacking them
  - Provides “copy-on-write”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

# Difference Between VMs and Containers

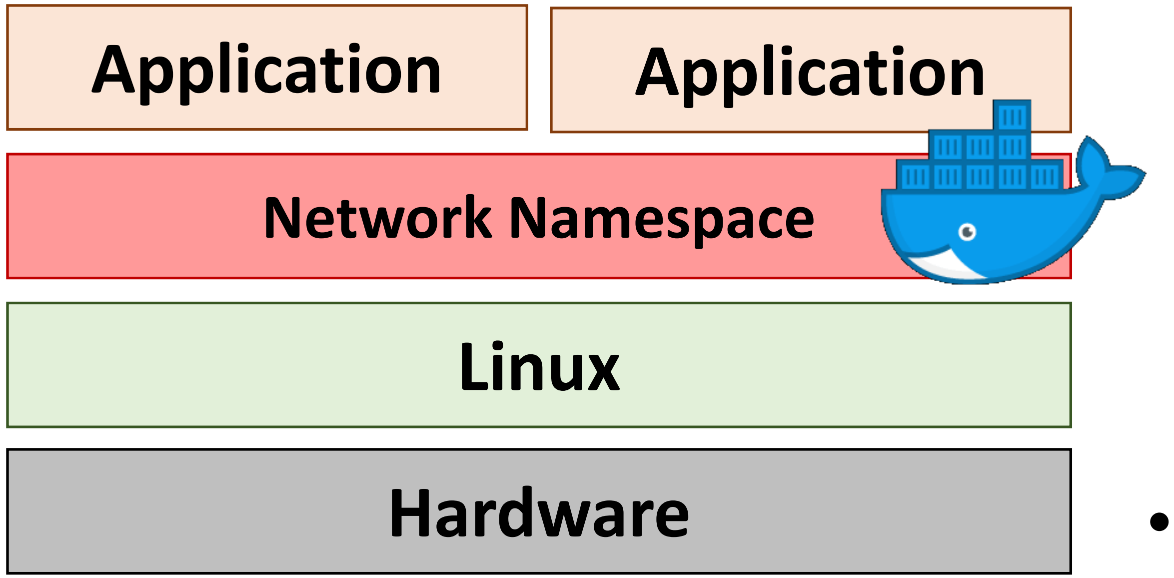


# Example - Manually Creating Containers

```
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
```



# What is Docker?




- A set of container 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

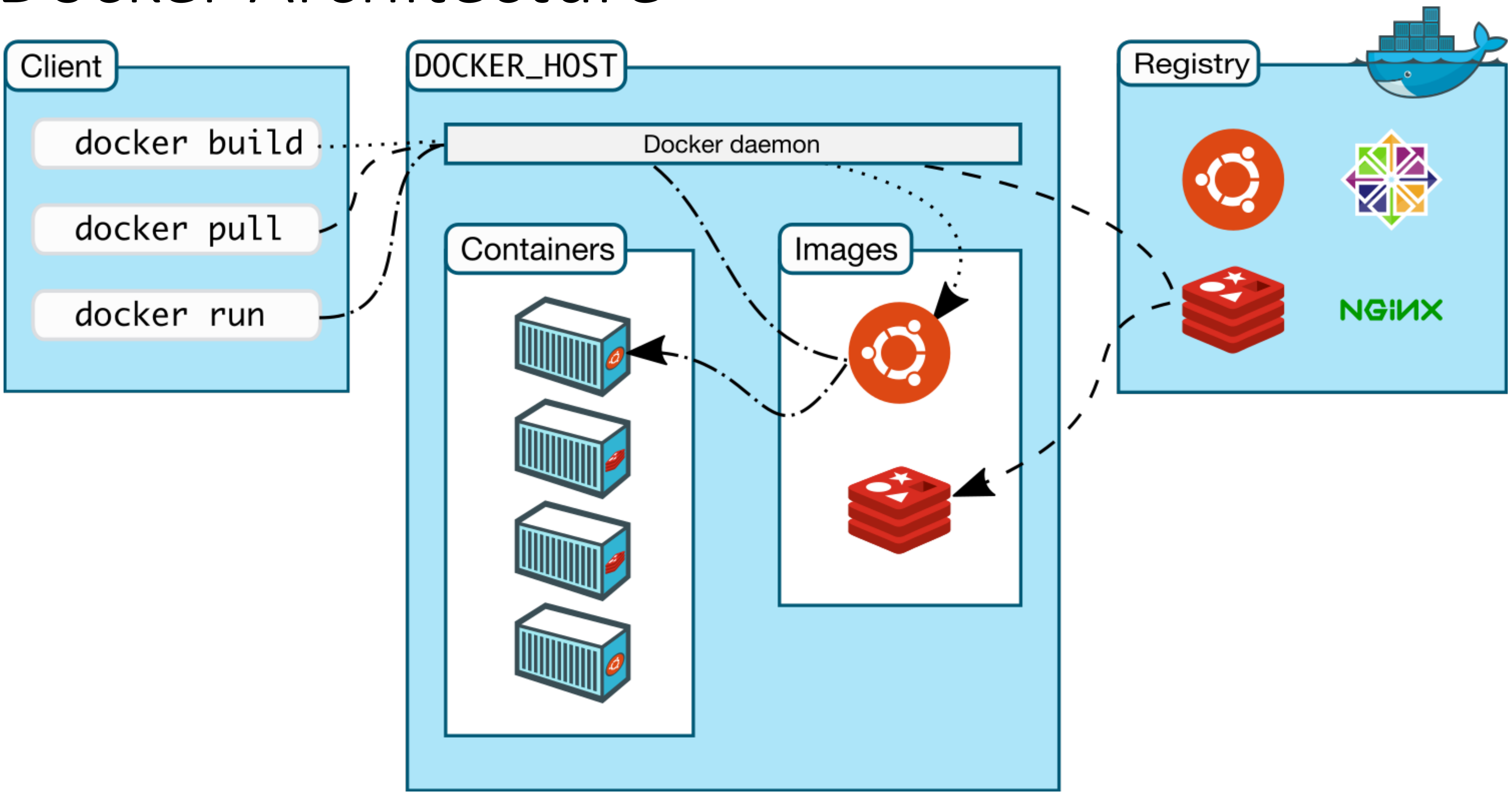
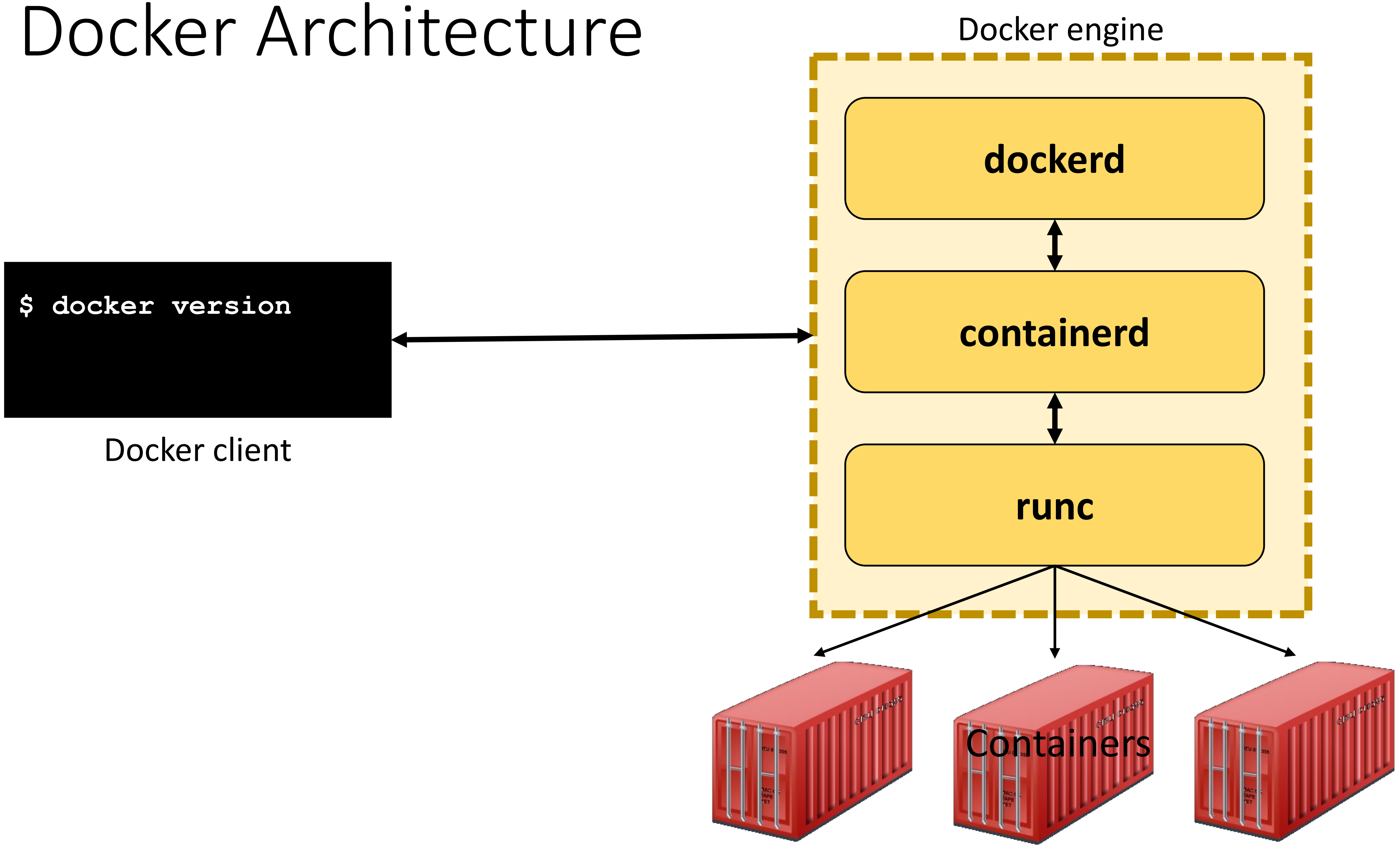



Image from https://docs.docker.com/engine/images/architecture.svg

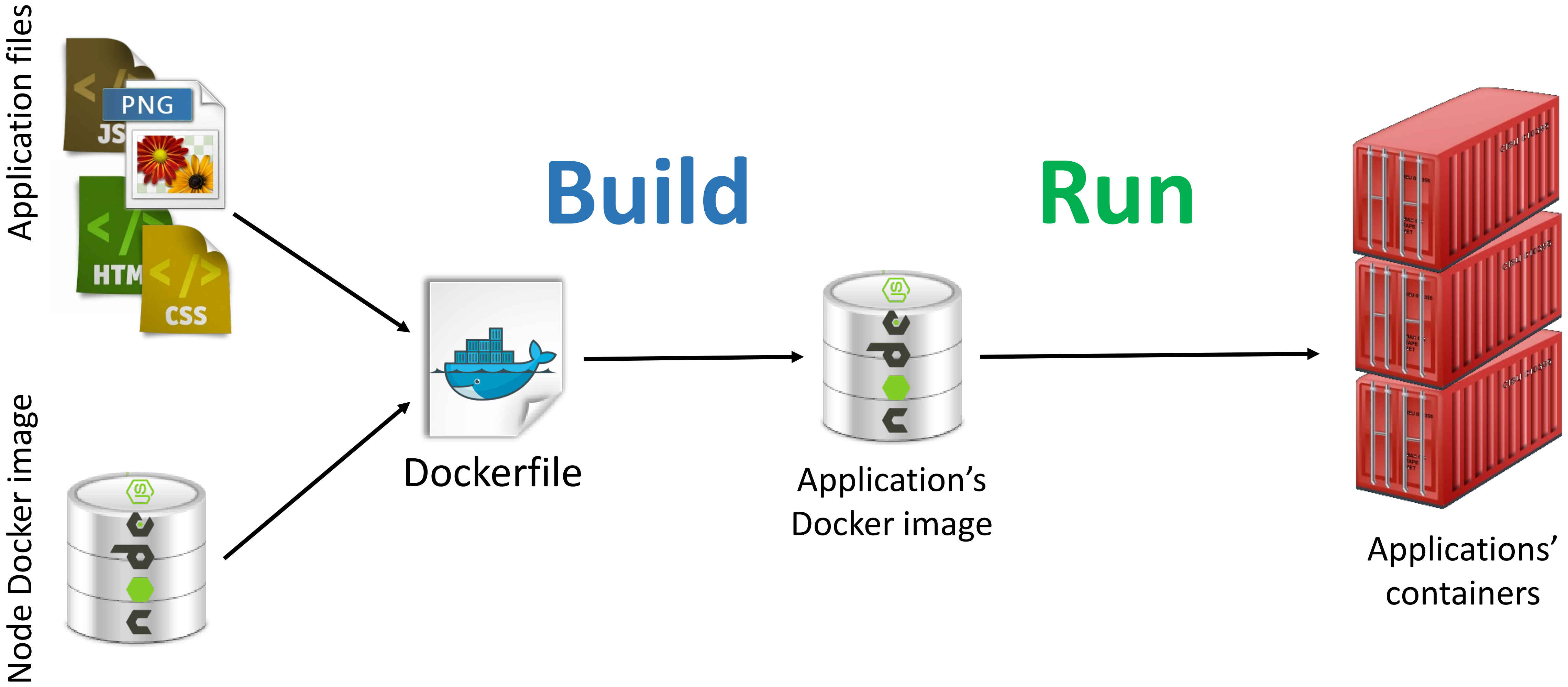
# Docker Architecture



# Benefits of Containers

- Build once, run anywhere
  - Very portable; containerized applications will run on any systems with Docker installed
- Provides application isolation and resource sharing
  - Like VMs, containers provides application isolation; resources from the host can be allocated to containers
- Supports any application that scales horizontally
  - Can efficiently spin up multiple instances of the same application to meet throughput requirements; excellent way to deploy micro services
- Improve developer productivity
  - Create reproducible development and isolated environment like Python, Golang, JavaScript, etc.

# Docker Workflow

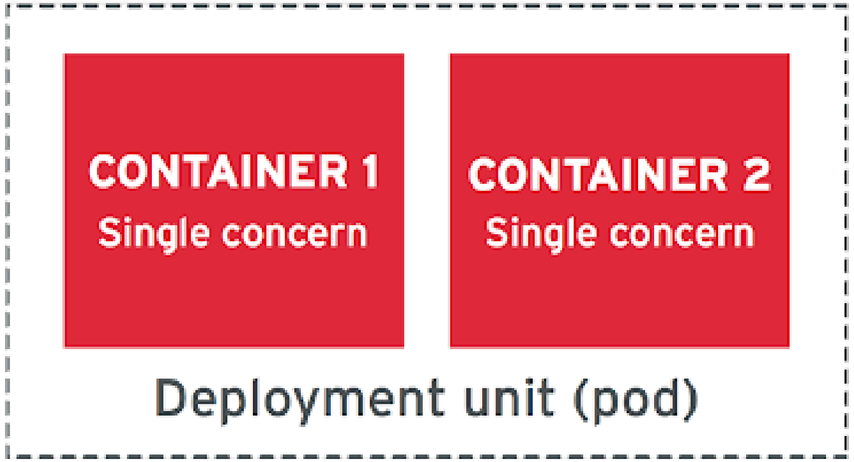


# Containerizing an Application

- `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

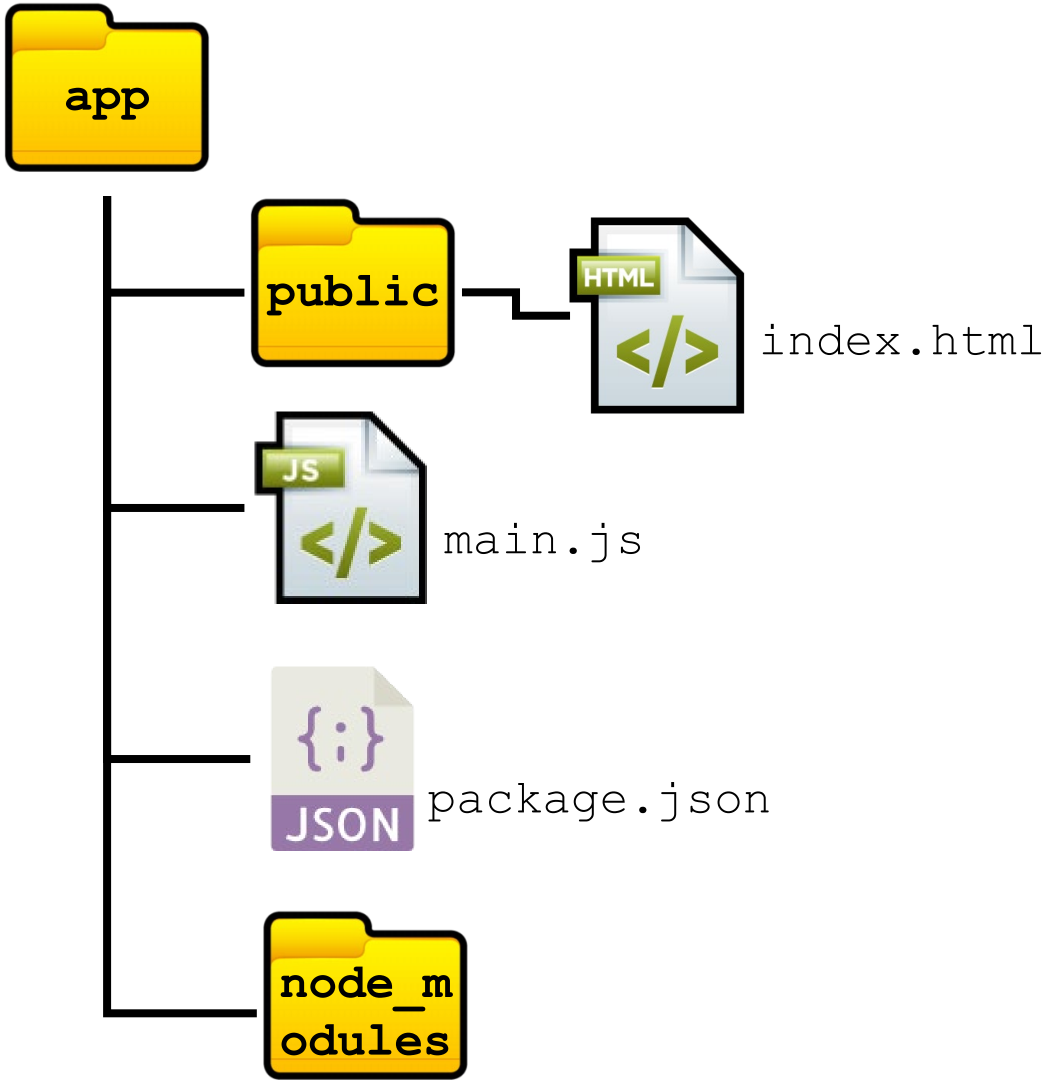
# Single Concern Principle

- 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



# 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:20

LABEL name=myapp

ARG APP_DIR=/app

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 ${APP_PORT}
```

Use the node image as the base to build the application

Add labels to the image

Build arguments

Sets the working directory. Like ‘cd’ into the directory

Add all these files and directories into `$APP_DIR`

Installs node modules

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

Sets the environment variable

Provide a default for `ENTRYPOINT`

Command to execute when container starts

# Dockerfile

```
FROM node:20

LABEL name=myapp

ARG APP_DIR=/app

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 ${APP_PORT}
```

For running the image

# Dockerfile Directives

- `FROM` - creates a new container from the specified base image
- `WORKDIR` - sets the working directory, any directive after this will be performed inside the specified directory
- `ADD` - copies files from local machine into the image
  - Also supports URL and TAR file
- `RUN` - executes a command in the image
- `ARG` - pass build arguments to the image builder
- `ENV` - sets an environment variable
- `EXPOSE` - tells Docker that the container will listen to a port
- `ENTRYPOINT` - configures the default command to run when the container starts
- `CMD` - like `ENTRYPOINT` but can be overwritten by another user specified command

# Building an Image

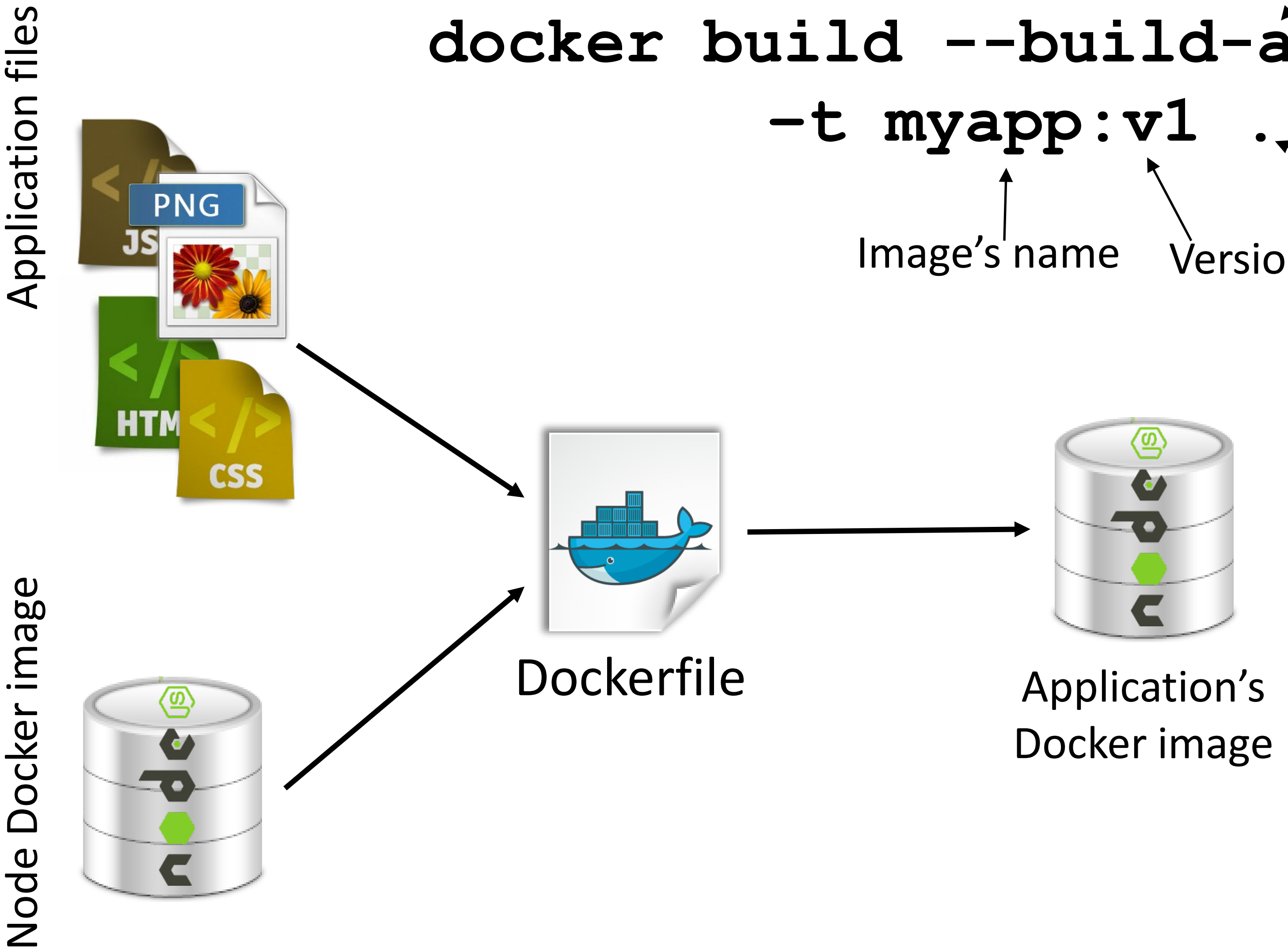
Override the APP_DIR during build

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

Image's name

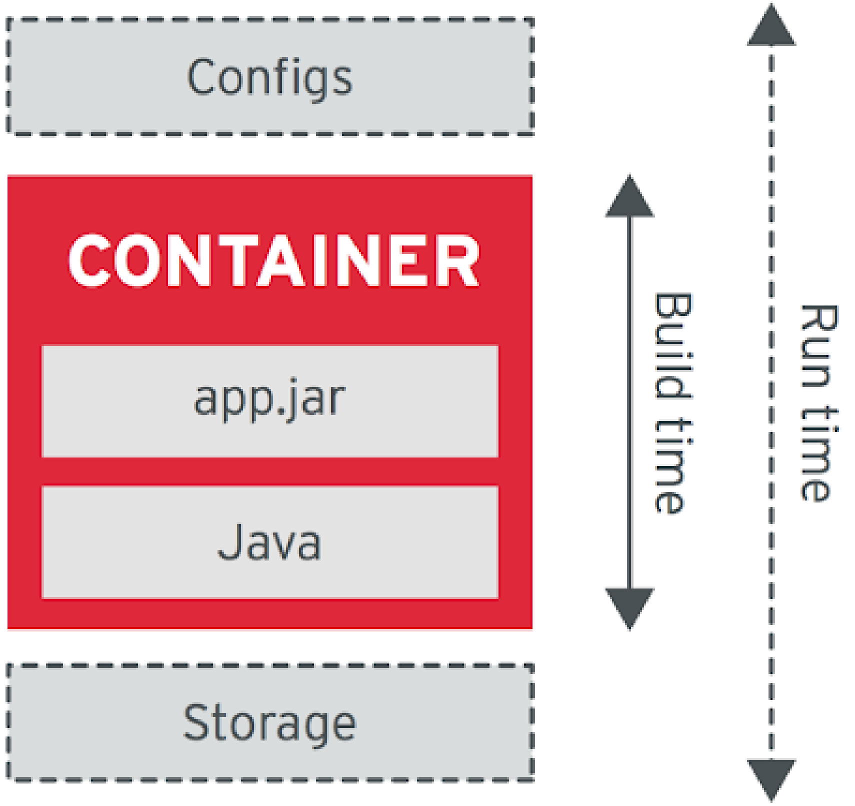
Version

Build context, the location of the files



# 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



# Image Naming

Location of container registry.
If omitted default to `docker.io`

Image name

**docker.io/fred/myapp:v1**

User registered with the container registry. If omitted in defaults to `library`

Image tag. If omitted, default to `latest`

# Starting a Container

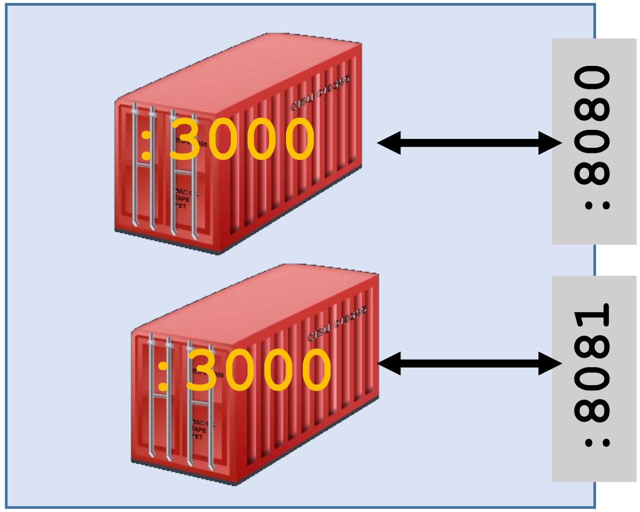
Run in detached mode

Container name

Image name

```
docker container 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

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

Port binding

# Setting 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
```

# Starting a Container

| | No ENTRYPOINT | ENTRYPOINT abc 123 | ENTRYPOINT [ "abc", "123" ] |
|---|---|---|---|
| No CMD | Not allowed | /bin/sh -c "abc 123" | abc 123 |
| CMD xyz 789 | /bin/sh -c "xyz 789" | /bin/sh -c "abc 123" | abc 123 /bin/sh -c xyz 789 |
| CMD [ "xyz", "789" ] | xyz 789 | /bin/sh -c "abc 123" | abc 123 xyz 789 |

From https://docs.docker.com/engine/reference/builder/#understand-how-cmd-and-entrypoint-interact

# Starting a Container

```
CMD node main  ⇒ ls -l

   docker container run -ti mycontainer ls -l

CMD [ "node", "main" ]  ⇒ ls -l

   docker container run -ti mycontainer ls -l

ENTRYPOINT node main  ⇒ node main

   docker container run -ti mycontainer --port=8080

ENTRYPOINT [ "node", "main" ]  ⇒ node main --port=8080

   docker container run -ti mycontainer --port=8080
```

# Starting a Container

```
ENTRYPOINT node main
CMD --port=8080  ⇒ node main

    docker container run -ti mycontainer

ENTRYPOINT [ "node", "main" ]
CMD [ "--port=8080" ]  ⇒ node main --port=8080

    docker container run -ti mycontainer

ENTRYPOINT [ "node", "main" ]
CMD [ "--port=8080" ]  ⇒ node main --port=5000

    docker container run -ti mycontainer --port=5000
```

# Container and Image Management

- List all running containers

```
docker container ps
```

- Stop a container

```
docker container stop mycontainer
```

- Start a container

```
docker container start mycontainer
```

- Delete a container

```
docker container rm mycontainer
```

- List all images

```
docker image ls
```

- Executing a command in the container

```
docker rmi myimage:v1
```

# Primary Process

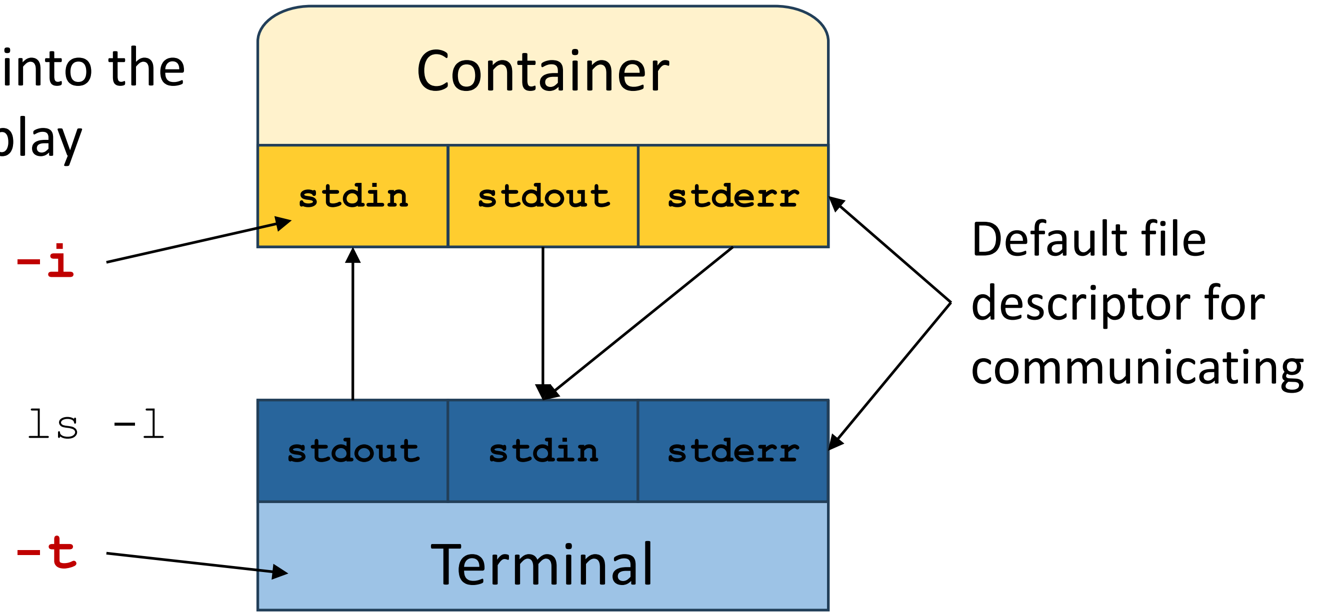
- Primary process is the main process that is running inside the container
  - With the PID of 1
- Primary process is specified by
  - `ENTRYPOINT` directive
  - `CMD` directive is `ENTRYPOINT` is missing
  - Overridden when launching a container
- If this process dies, the container exits
  - Caused by natural termination, application error, misconfiguration, etc

# Running Command Inside Containers

- Can execute command inside container
  - Provided that the command exists
- Can only execute command if the primary process is running
  - The default command is running

Terminal to pass data (command) into the container and to received and display results from the container

```
docker container exec -ti mycontainer ls -l
```



# Pushing Images to Container Registry

- Share image by pushing local images to a container registry service
  - Login to the registry
  - Login to Docker hub

```
docker login -u fred
```

  - Login to other container registry eg. Github

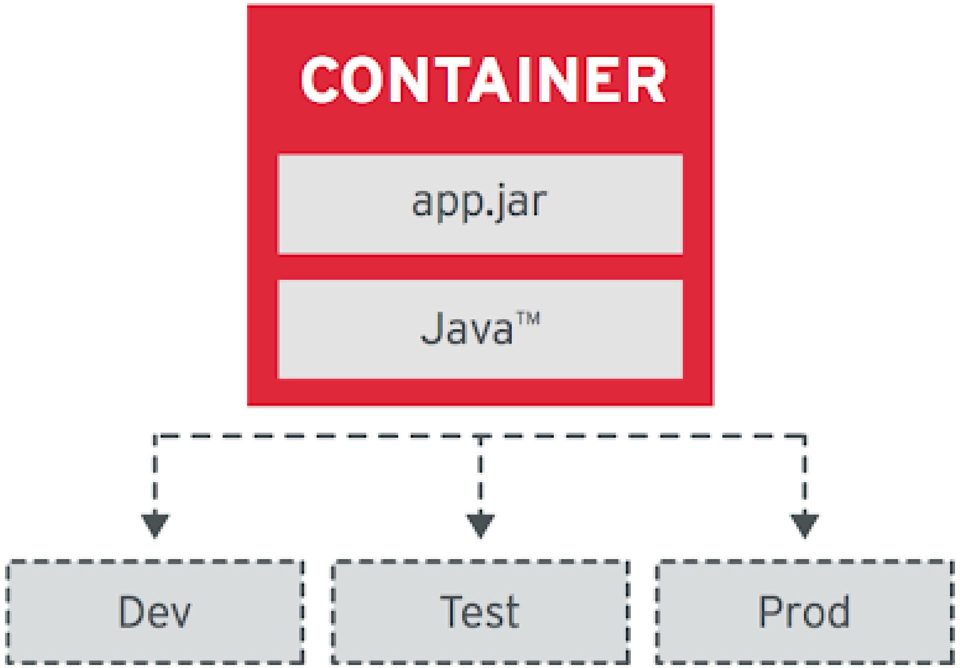
```
echo $PASSWORD | docker login ghcr.io -u fred --password-stdin
```

- Push or pull from container registry
  - Docker will automatically pull from container registry if image is not locally available

```
docker push fred/myapp:v1
docker pull ghcr.io/fred/myapp:v1
```

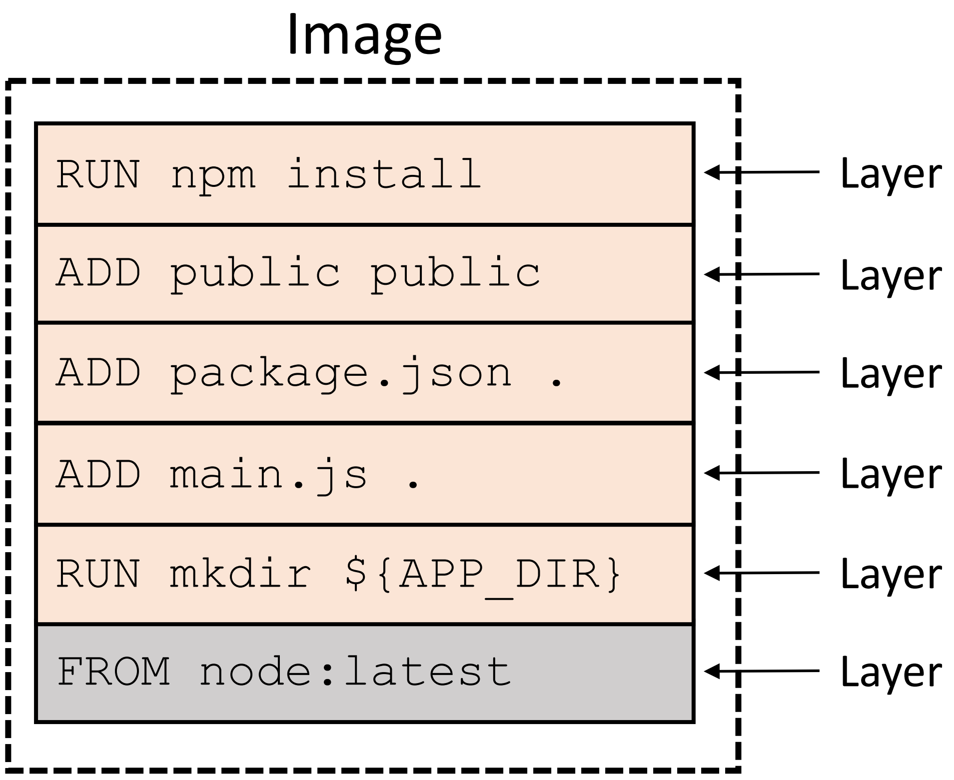
# Image Immutability Principle

- 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



# Image Layers

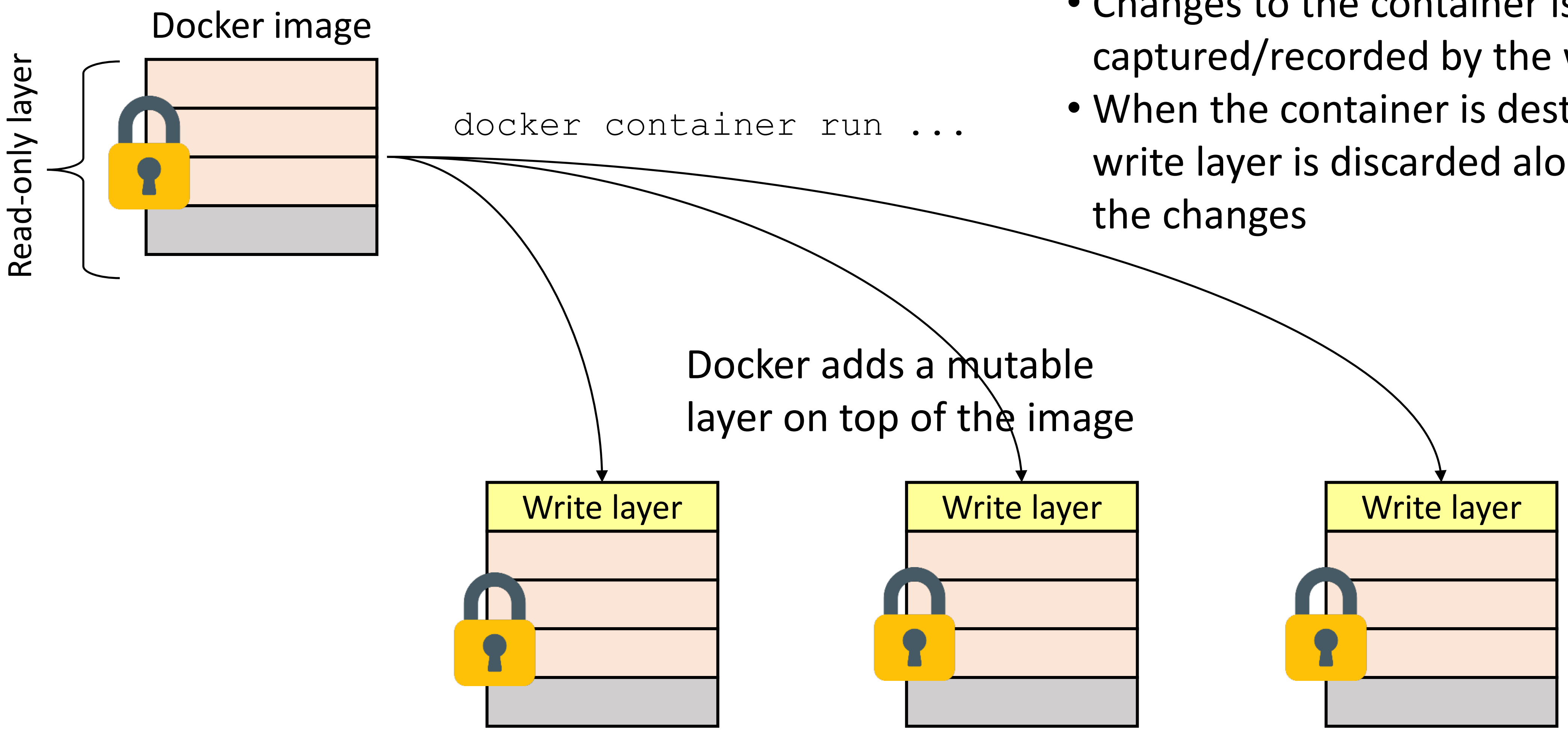
- Docker creates a layer for most directive
  - Layer captures the result of the directive
  - If later layer deletes a file created by a lower layer, the file is still present in the image, just not visible
    - Careful with sensitive information
- Layers are cached and reused, when appropriate
  - Decreases build time
- Layers are immutable which makes the image immutable
- View layers with dive
  - [https://github.com/wagoodman/dive](https://github.com/wagoodman/dive)




**`docker history myapp:v1`**

# Ephemeral Write Layer

- A write layer is added to the top of the image when a container is created
- Changes to the container is captured/recorded by the write layer
- When the container is destroyed, the write layer is discarded along with all the changes



# 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

# Custom Signal

- `STOPSIGNAL` directive allows you to override the default SIGTERM
  - Default to `SIGTERM` if not specified
- Common signals
  - `SIGHUP`
  - `SIGKILL`
  - `SIGQUIT`
  - `SIGUSR1`

```
Dockerfile
...
SIGSTOP SIGHUP
```

# 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
- Logs - generated logs for postmortem analysis
- Metrics - for monitoring systems like Prometheus to monitor and measure the container

# 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

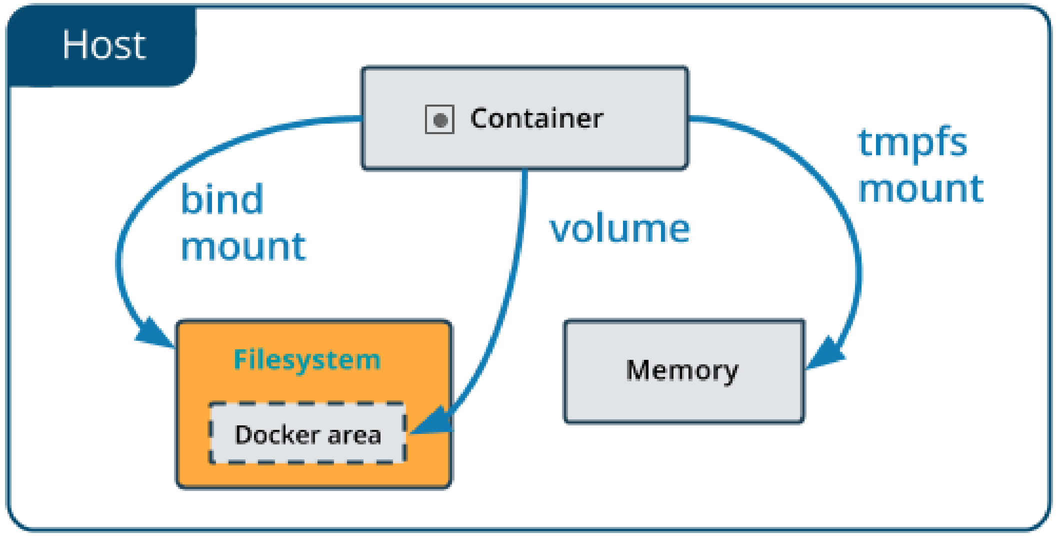
# Process Disposability Principle

- 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

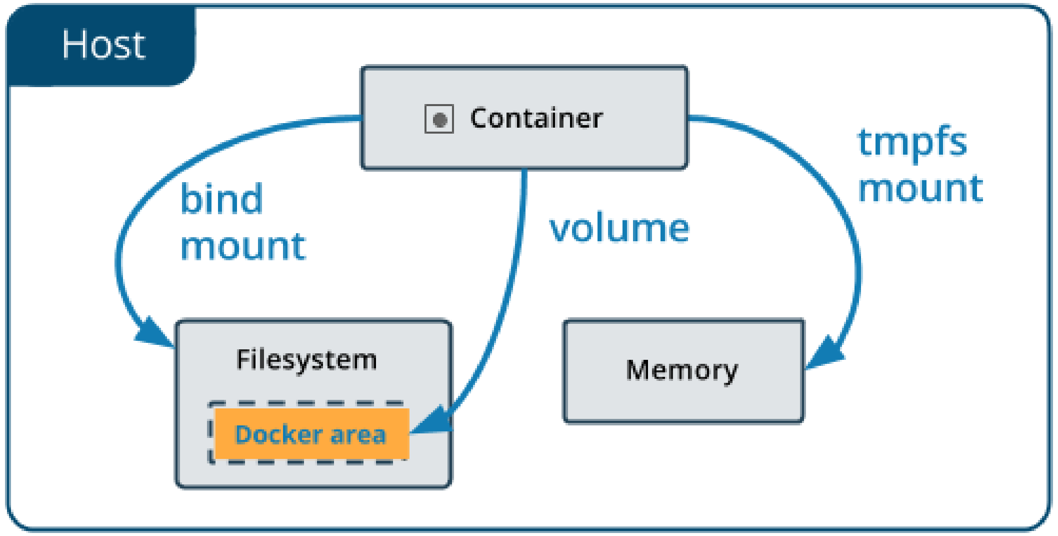
# Persistent Data

- 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 must be externalized
  - Written to storage volumes outside of the container
  - When the container is deleted, the data is not deleted
- Two ways of mapping external storage into Docker
  - Bind mount - mount a directory from the Docker host into the container
  - Volume mount - create a Docker volume and mount the volume into the container

# Persistent Data




Bind mount




Volume mount

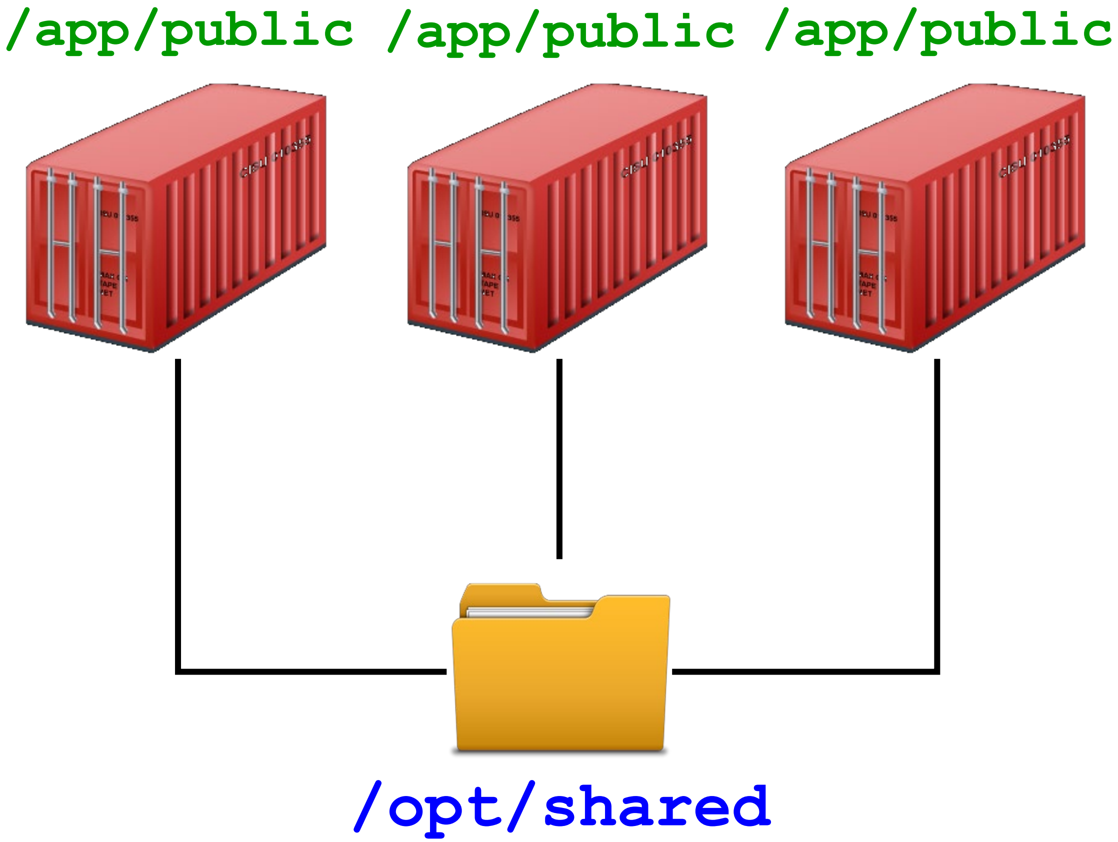
Image from https://docs.docker.com/storage/

# Bind Mount

Sharing read-only content

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

VOLUME ${APP_DIR}/public

EXPOSE ${APP_PORT}
...
```

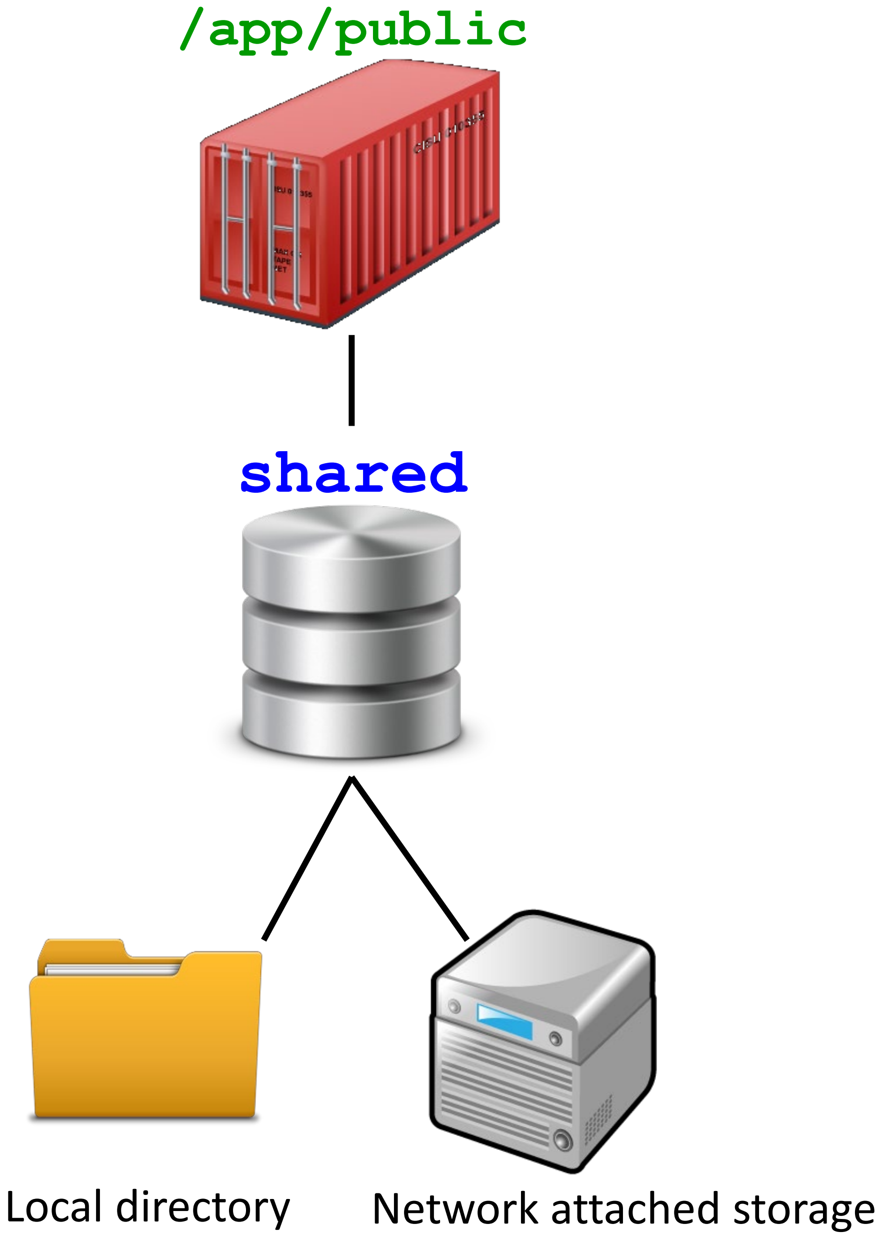
Define a mount point in the container




```
docker run -d -p 8080:3000 \
   --mount \
   type=bind,src=/opt/shared,dst=/app/public,readonly \
   --name app myapp:v1
```

# Volume Mount

- 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
```

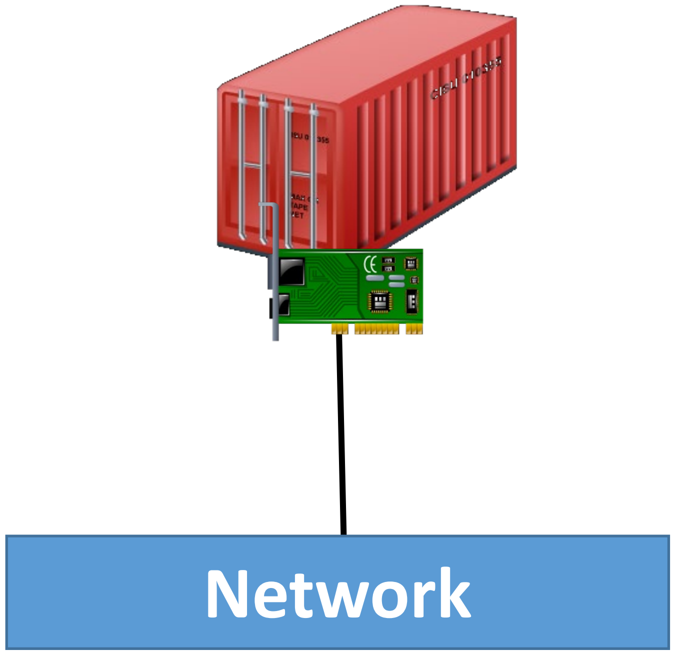
# Creating and Mounting a Volume

```
docker volume create shared

docker run -d -p 3000-3100:3000 \
  --mount type=volume,src=shared,dst=/app/public \
  --name app0 myapp:v1
```

Volume name without the leading /

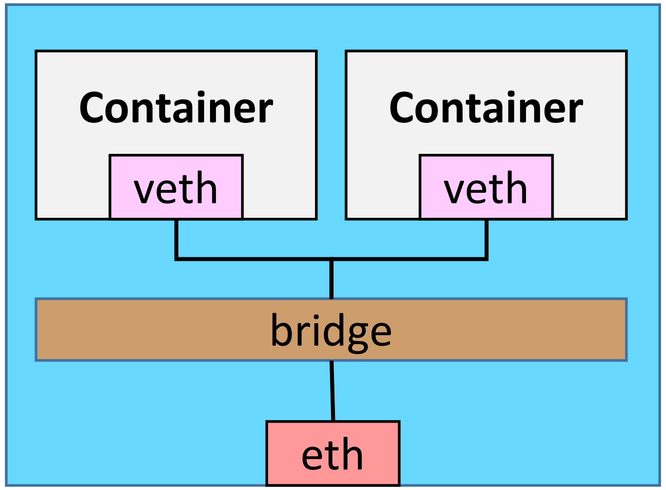
# Networking

- 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



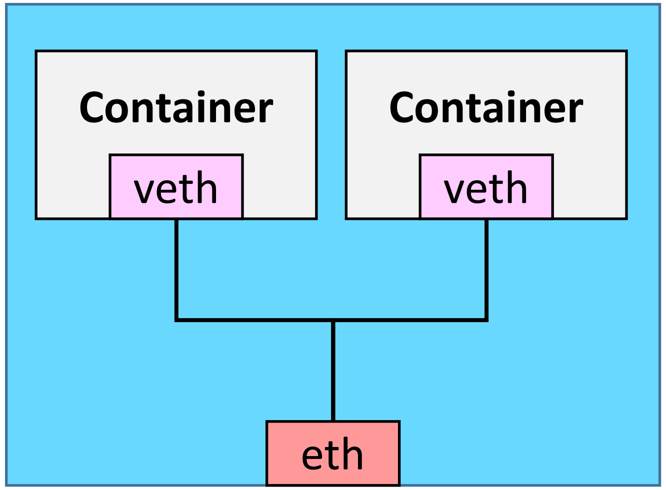
# Docker Network - Bridge

- 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



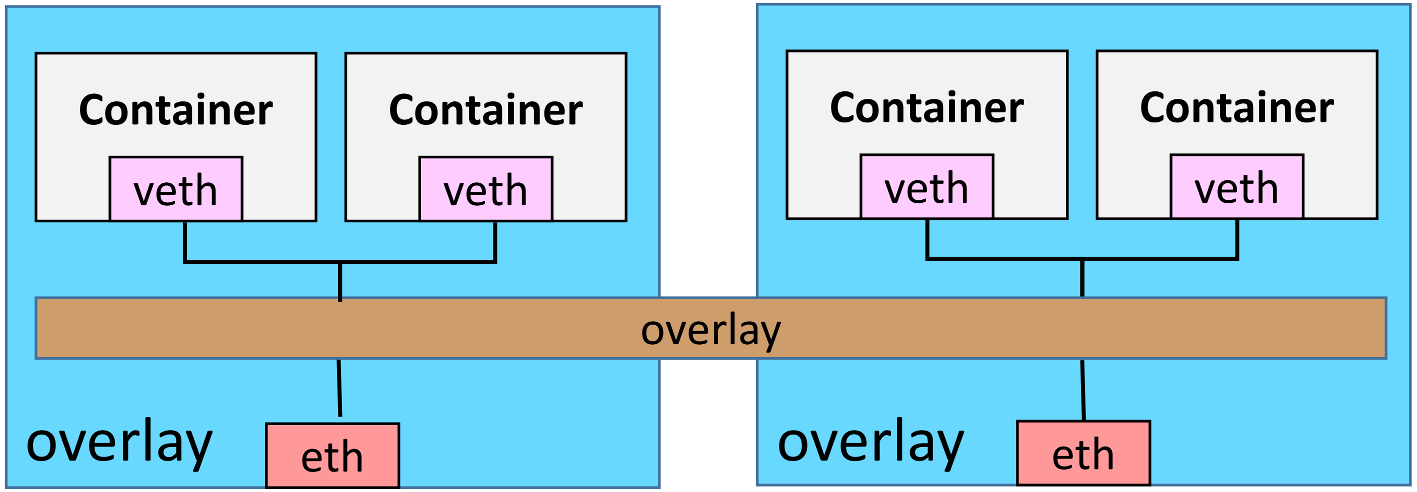
# Docker Network - Host

- 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

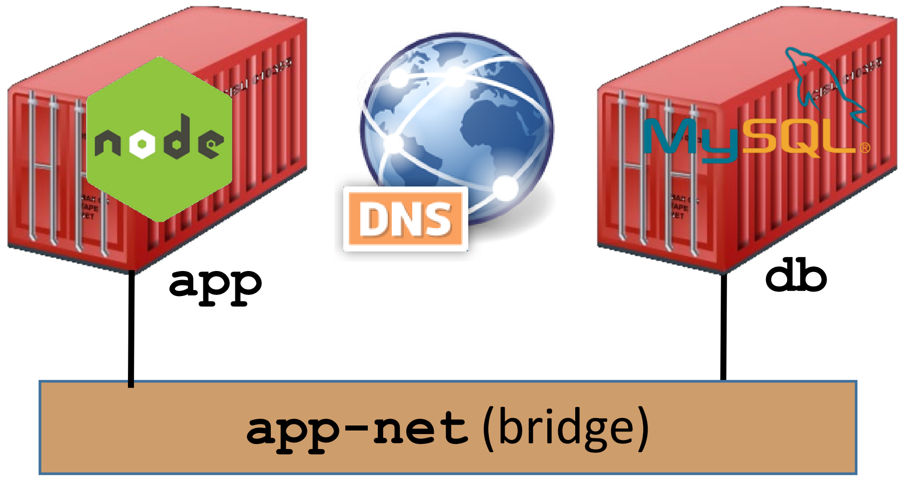
Plumb the container to bridge network

```
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 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

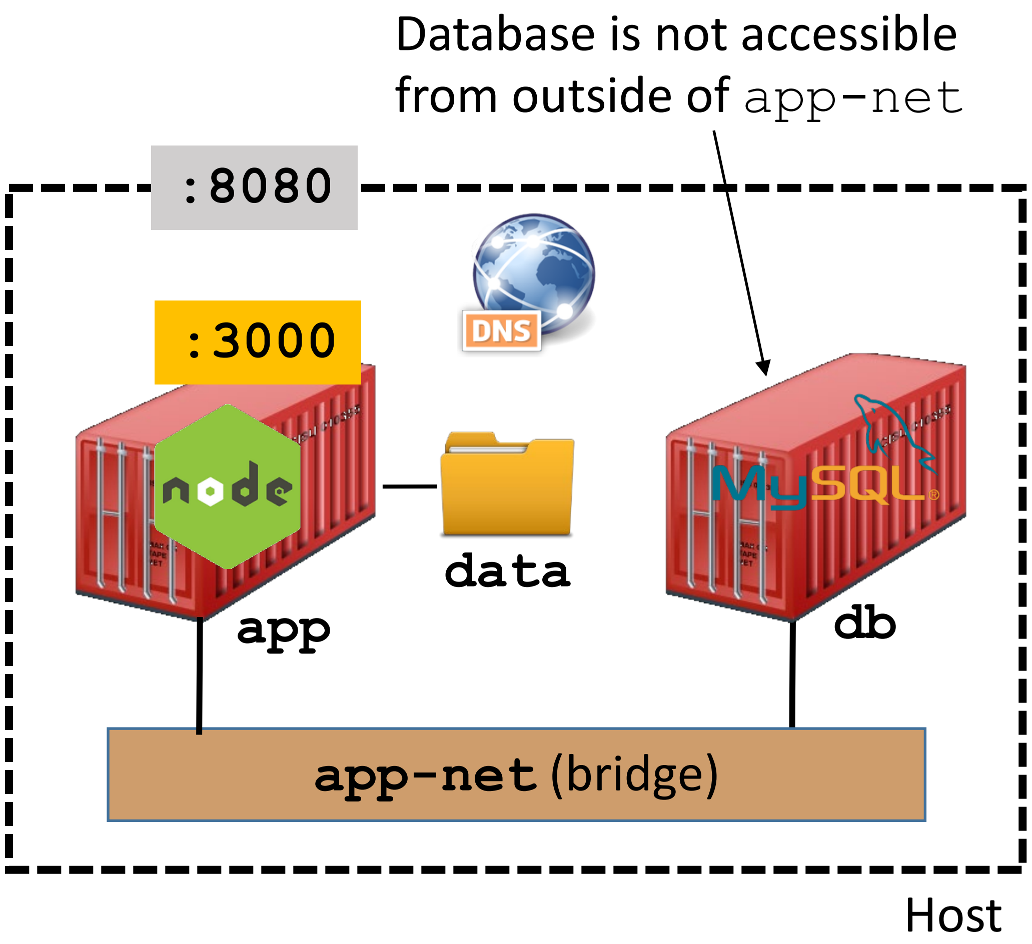
  ```
  docker network inspect mynet
  ```

- Delete a volume

  ```
  docker network rm mynet
  ```

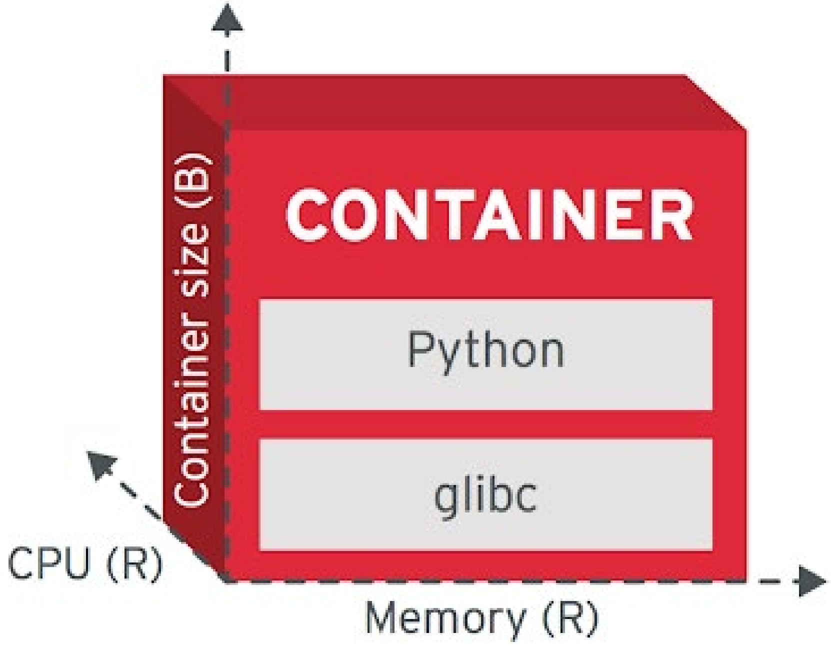
# Deploying Application Stack with Docker

```
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

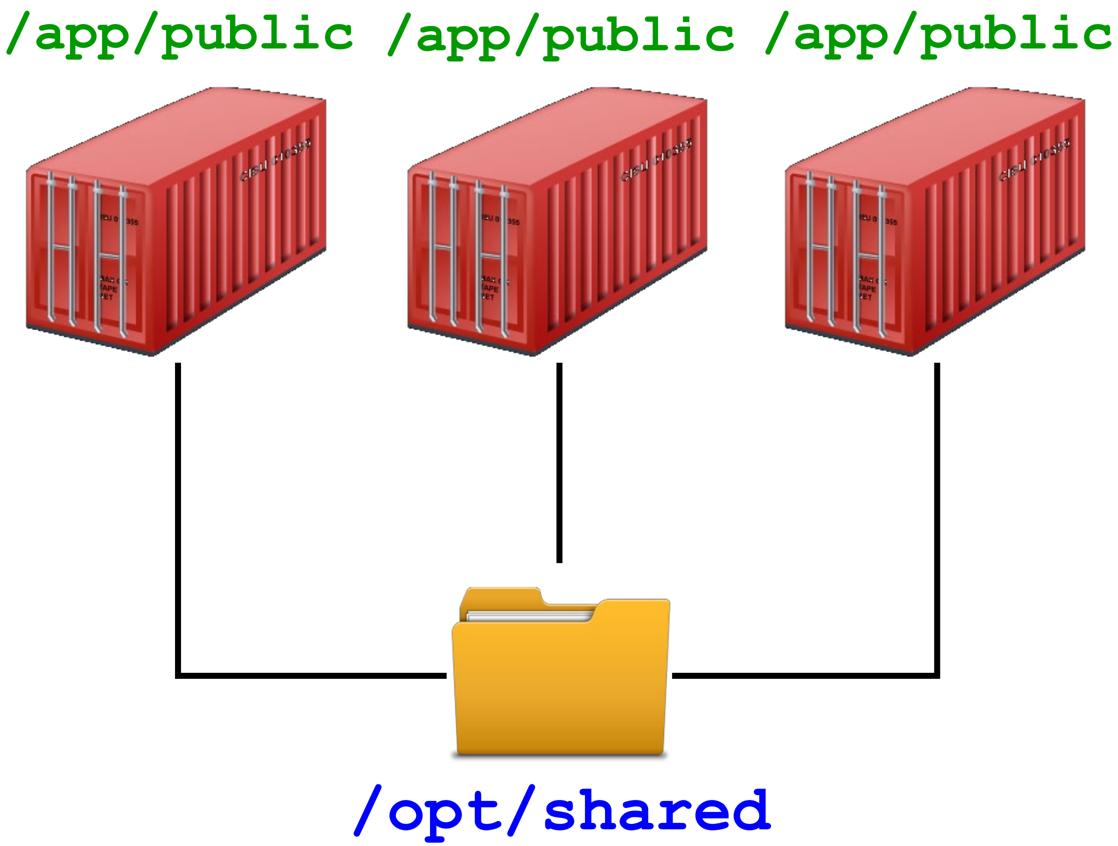
# Appendix

# Bind Mount

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

VOLUME ${APP_DIR}/public

EXPOSE ${APP_PORT}
...
```

Define a mount point in the container

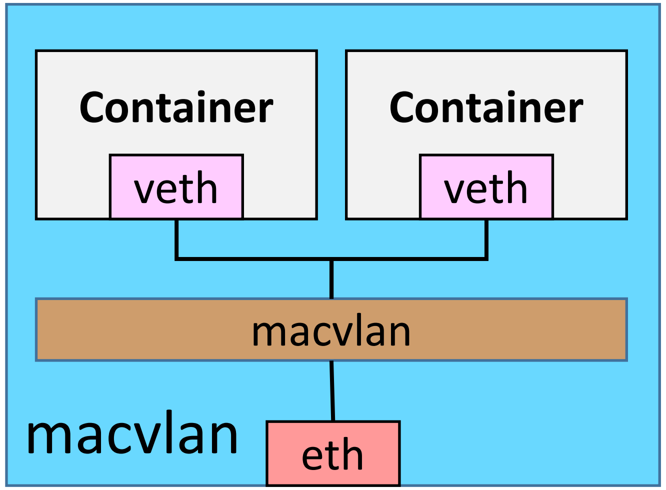
```
docker run -d -p 8080:3000 \
   -v /opt/shared:/app/public \
   --name app myapp:v1
```




Sharing read-only content

# Docker Networking - Macvlan

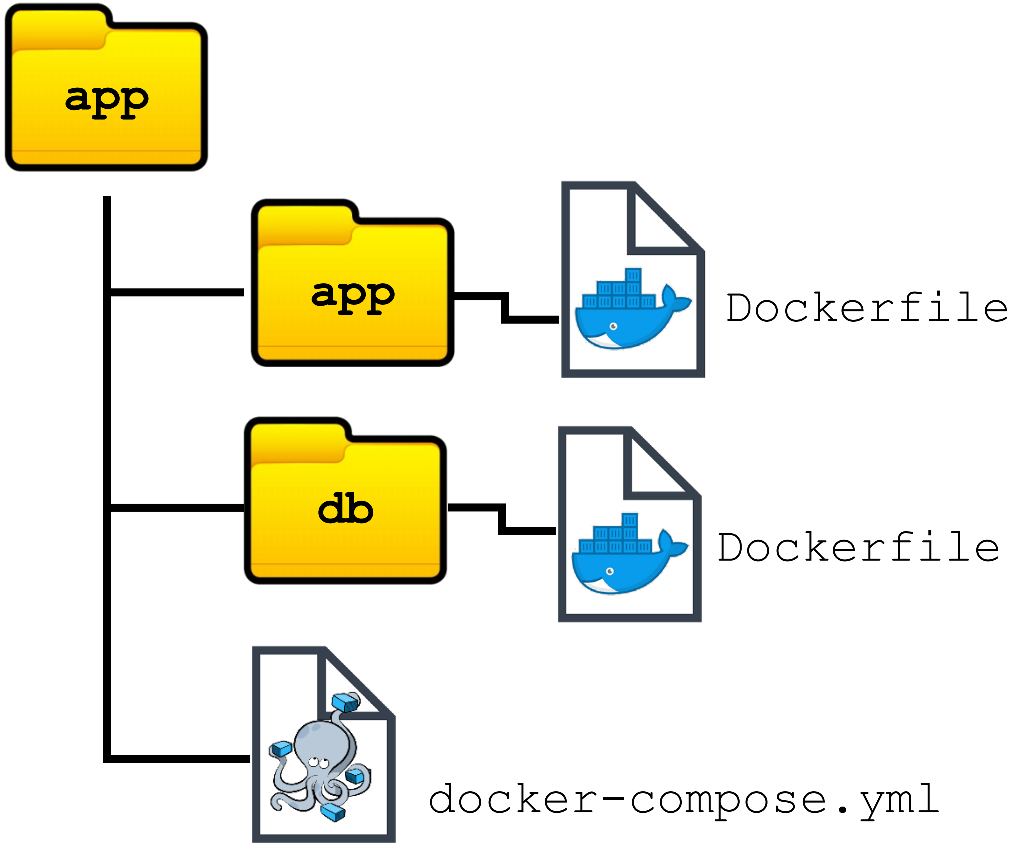
- Allows containers to be directly connected to the physical network
  - Each container will have their own IP address
  - Containers appear as independent systems on the physical network

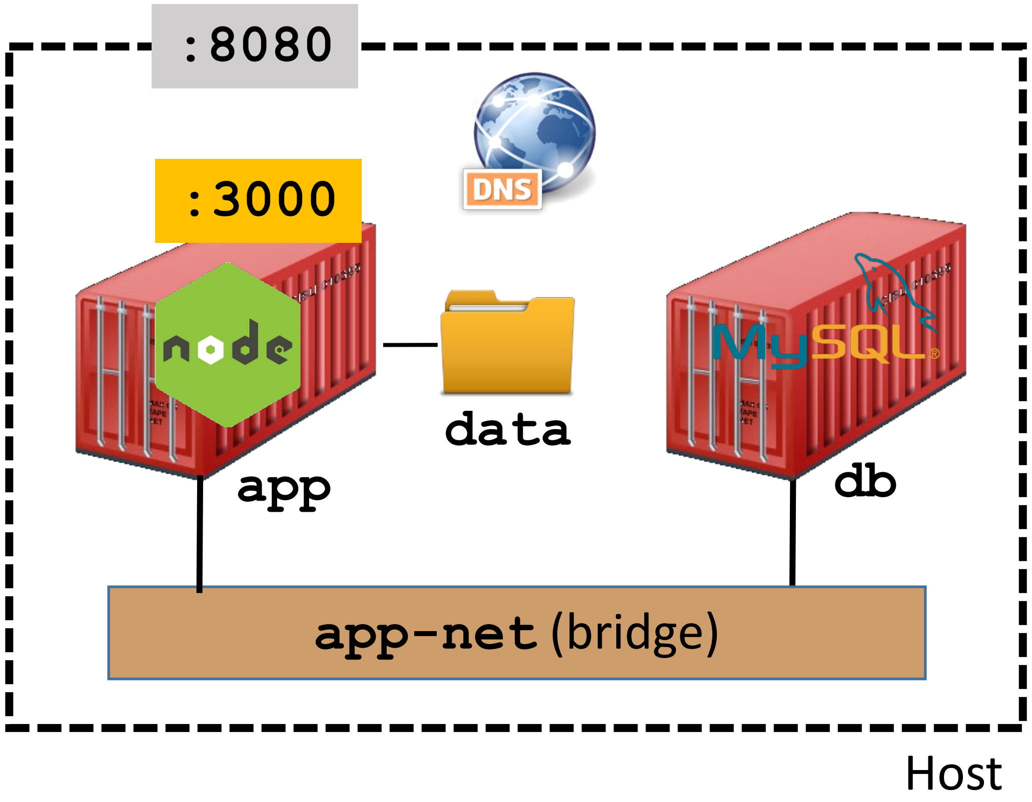


# Docker Compose

- 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

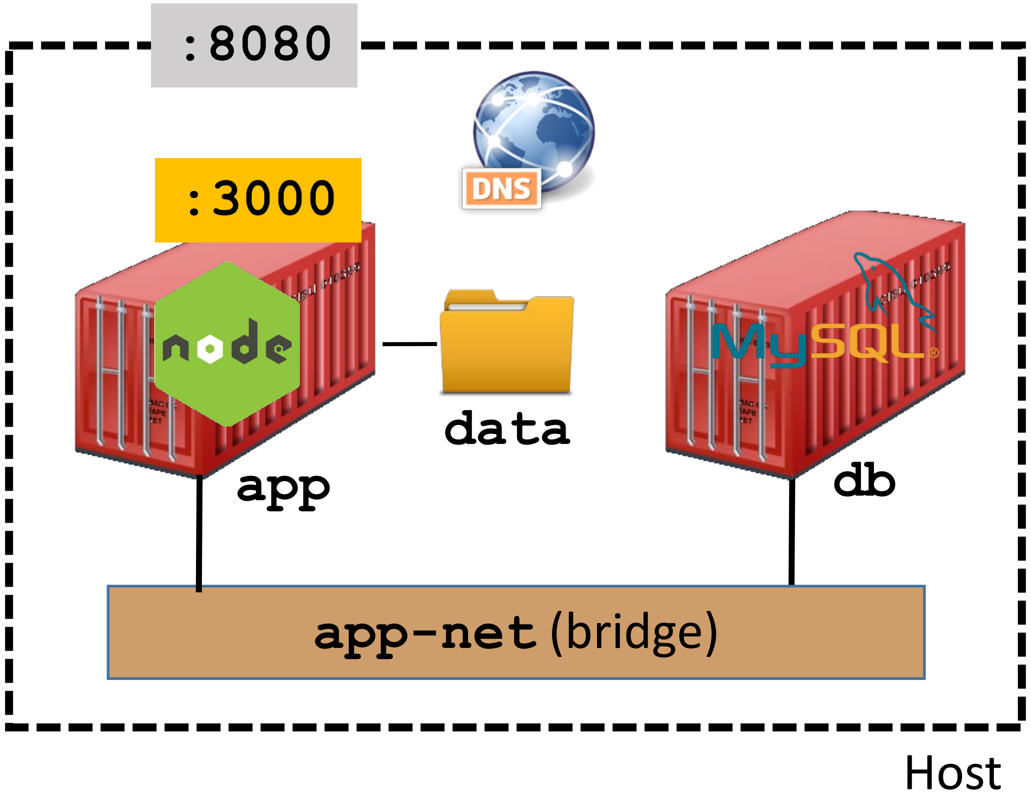
# docker-compose.yml



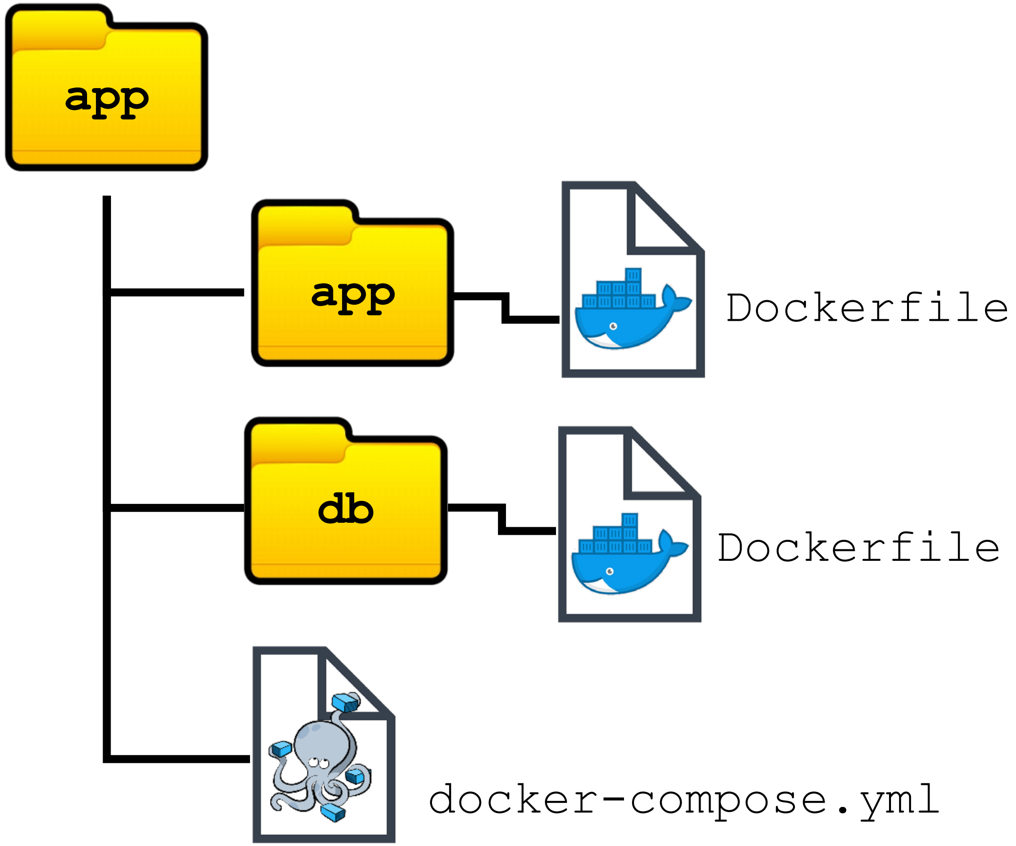
# docker-compose.yml

```
version: '3'

volumes:
  data:

networks:
  app-net:
```



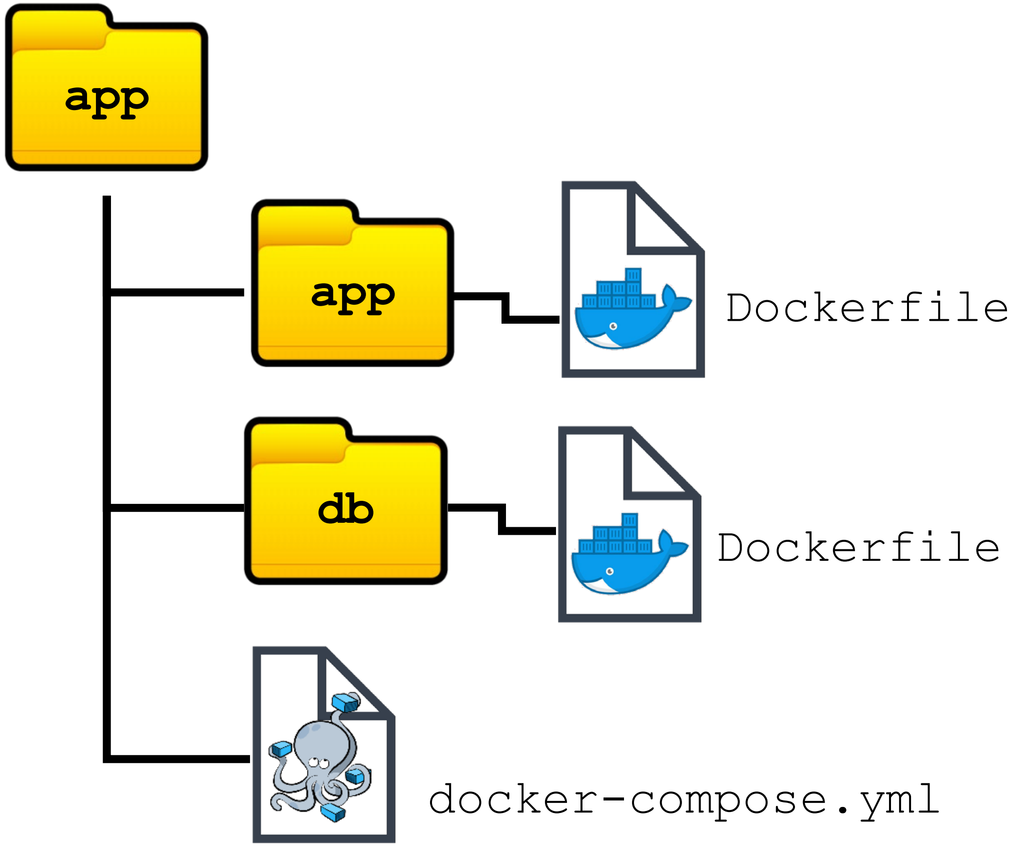
# docker-compose.yml

```
...
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
```