# Docker Fundamentals

TEKsystems Global Services

# WORKFORCE DEVELOPMENT

PARTICIPANT GUIDE

# logistics

- Class Hours:
- Instructor will provide class start and end times.
- Breaks throughout class.

- Telecommunication:
- Turn off or set electronic devices to vibrate
- Reading or attending to devices can be distracting to other students

- Lunch:
- Yes, 1 hour and 15 minutes
- Extra time for email, phone calls, or simply a walk.

- Miscellaneous
- Courseware
- Bathroom

# Course Objectives

By the end of the course you will be able to:

- Describe the Docker platform and architecture
- Build, run, and manage Docker containers & images
- Configure container networks
- Maintain data in containers
- Configure a Docker Swarm cluster
- Install Docker on laptop for developer use

# Hi!

Jason Smith

Cloud Consultant with a Linux sysadmin background.
Focused on cloud-native technologies: automation, containers & orchestration

github
https://github.com/jruels

Expertise

- Cloud
- Automation
- CICD
- Docker
- Kubernetes

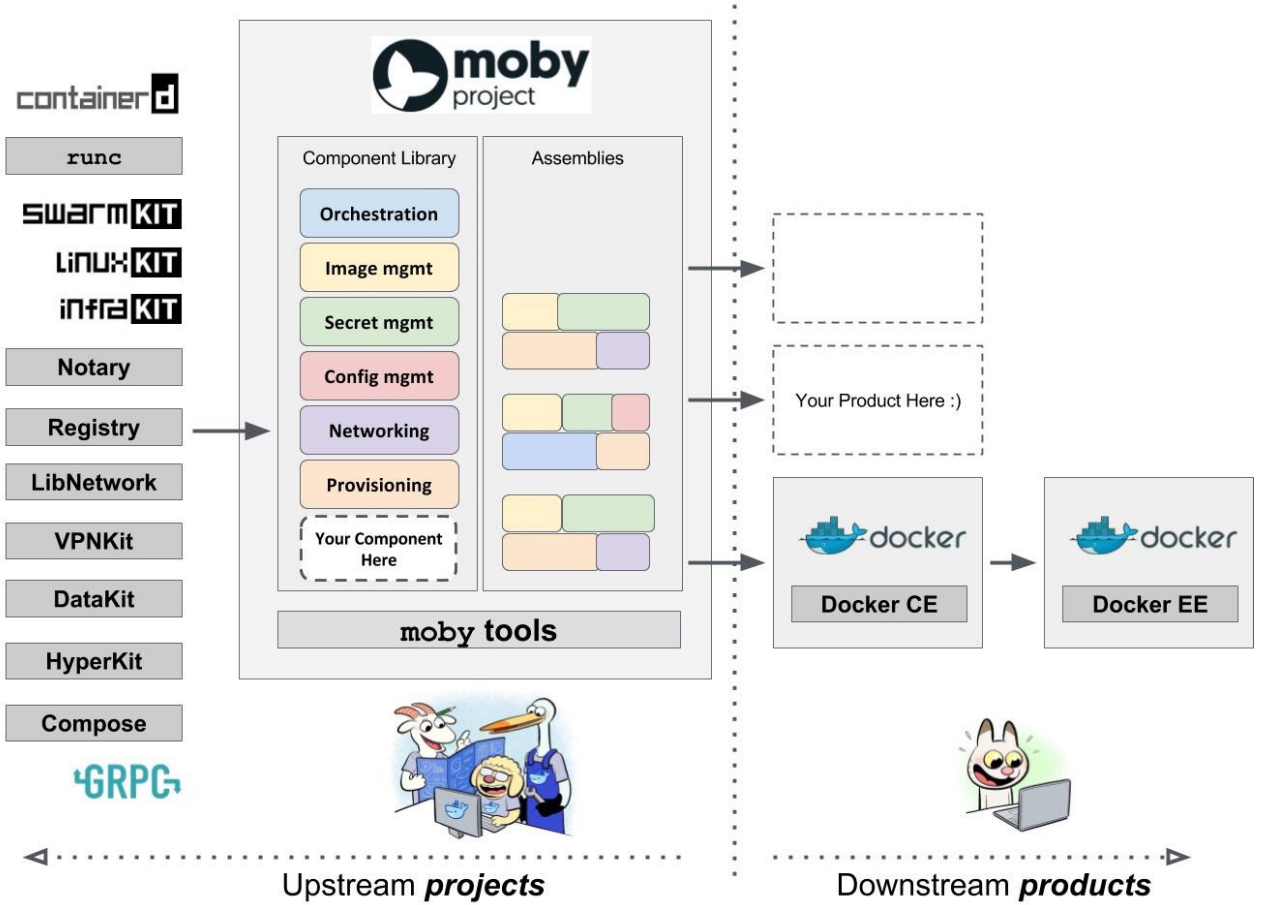
# Introductions

- Name
- Job Role
- Your experience with Docker (scale 1 - 5)
- Experience using a command-line text editor
- Expectations for course (please be specific)

# Docker Overview

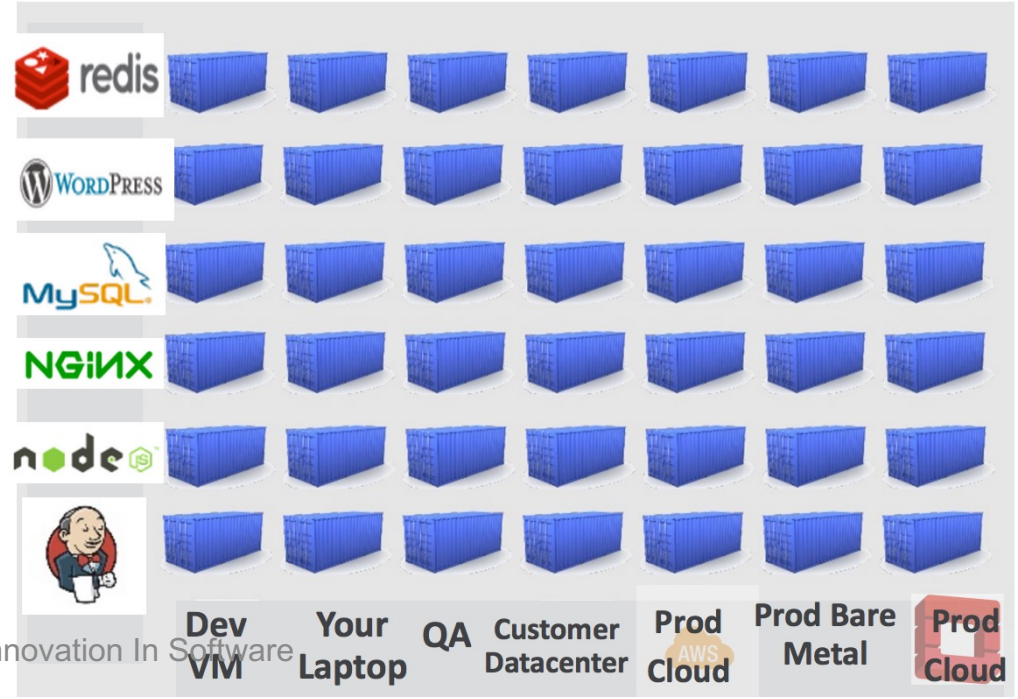


# What is Docker?



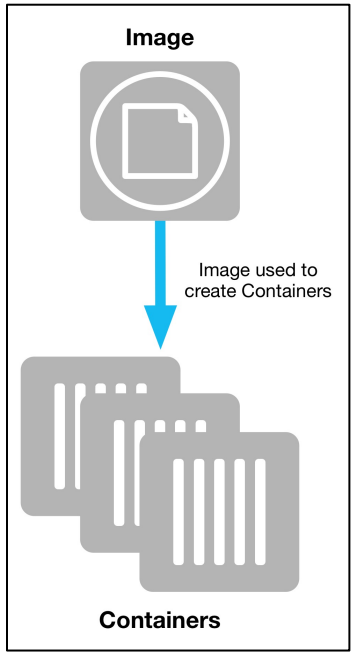
# What is Docker?


Docker allows you to package an application with all of its dependencies into a standardized unit for software development.



# Terminology

## Image




Read only template used to create containers
Built by you or other Docker users
Stored in Docker Hub, Docker Trusted Registry or your own Registry

# Terminology

*Image*

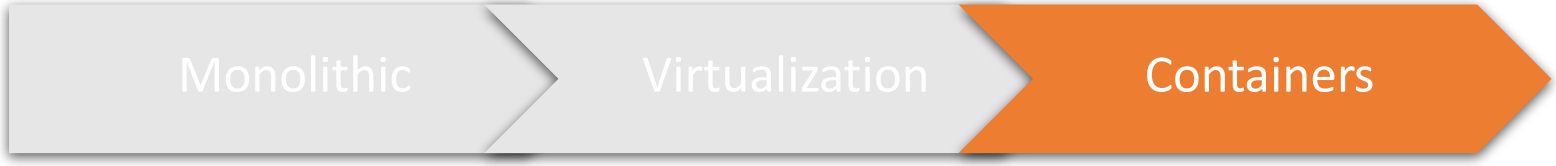
Read only template used to create containers
Built by you or other Docker users
Stored in Docker Hub, Docker Trusted Registry or your own Registry

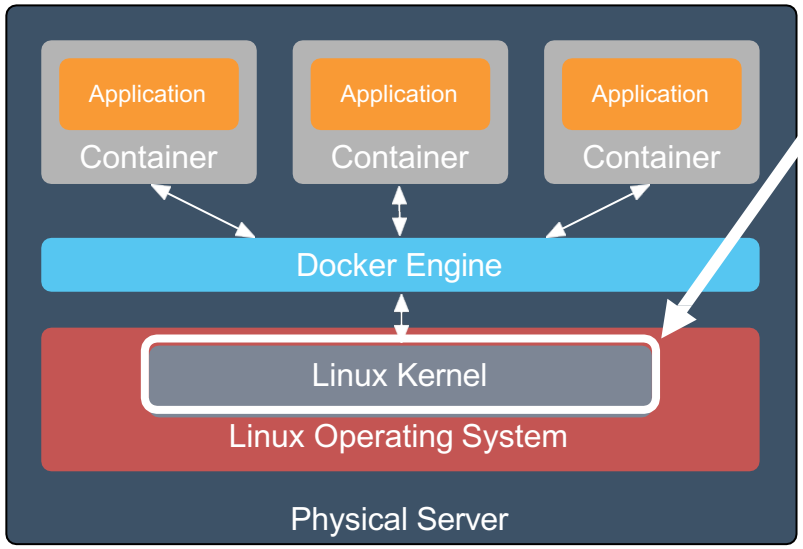
**Container**

- Isolated application platform
- Contains everything needed to run your application
- Based on one or more images

# Containers

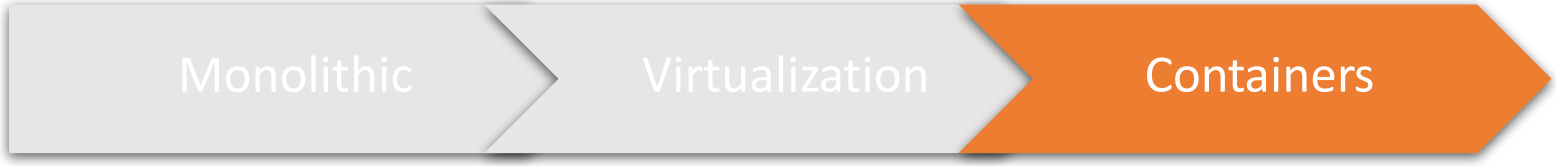
# Containers

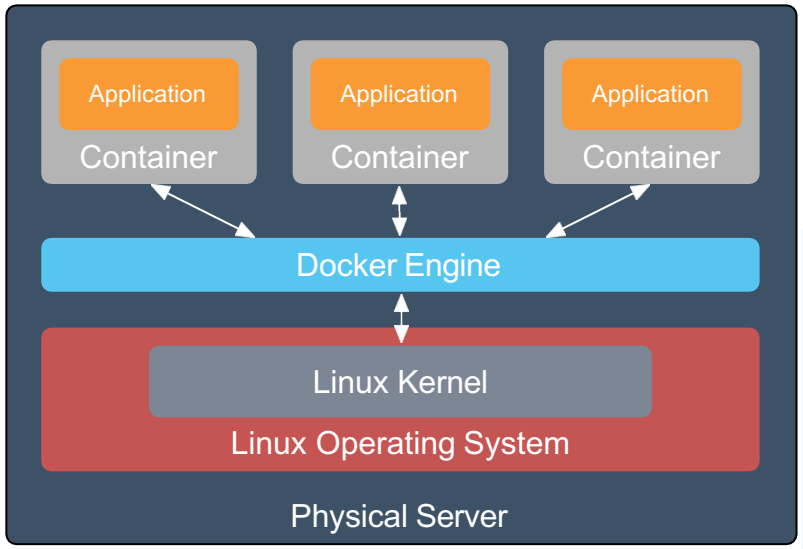



Shared kernel on the host to run multiple guest applications

# Containers - Advantages



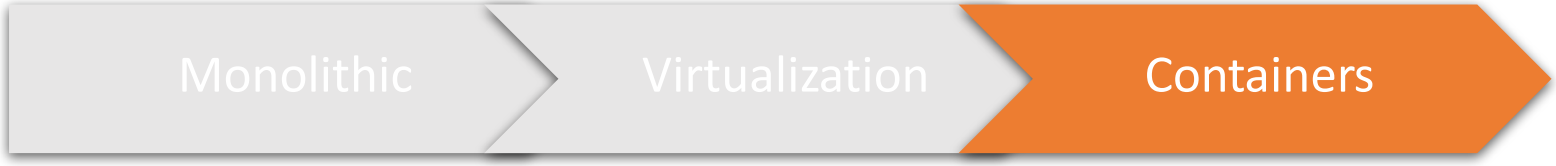



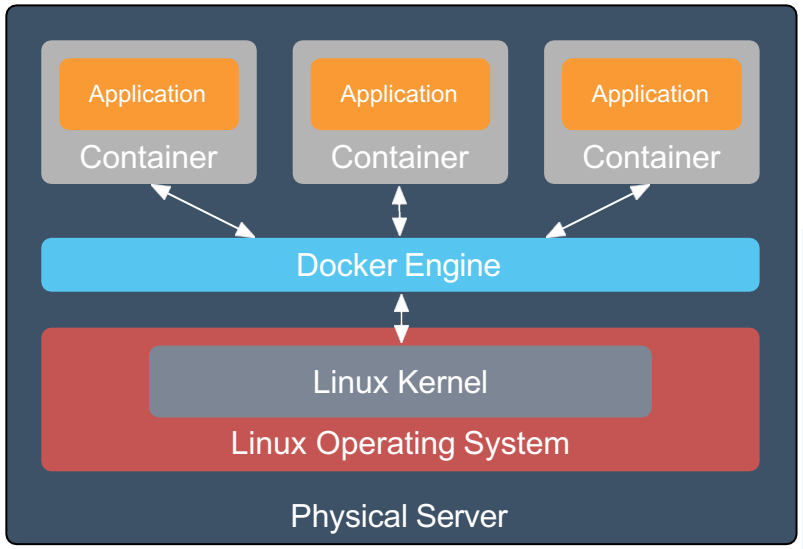

Shared kernel on the host to run multiple guest applications

## Advantages over VMs

- Containers are more lightweight
- No need to install a guest Operating System
- Less CPU, RAM, storage overhead
- More containers per machine
- Greater portability

# Containers - Challenges




Shared kernel on the host to run multiple guest applications

## Container Challenges

- Early Docker focused on single-node operations
- Up to user to cluster Docker hosts and manage deployment of containers on cluster
- User solves for automatic scale out of applications
- User solves for service discovery between application components (microservices)

# Container – Concept of Operations

# Container based virtualization

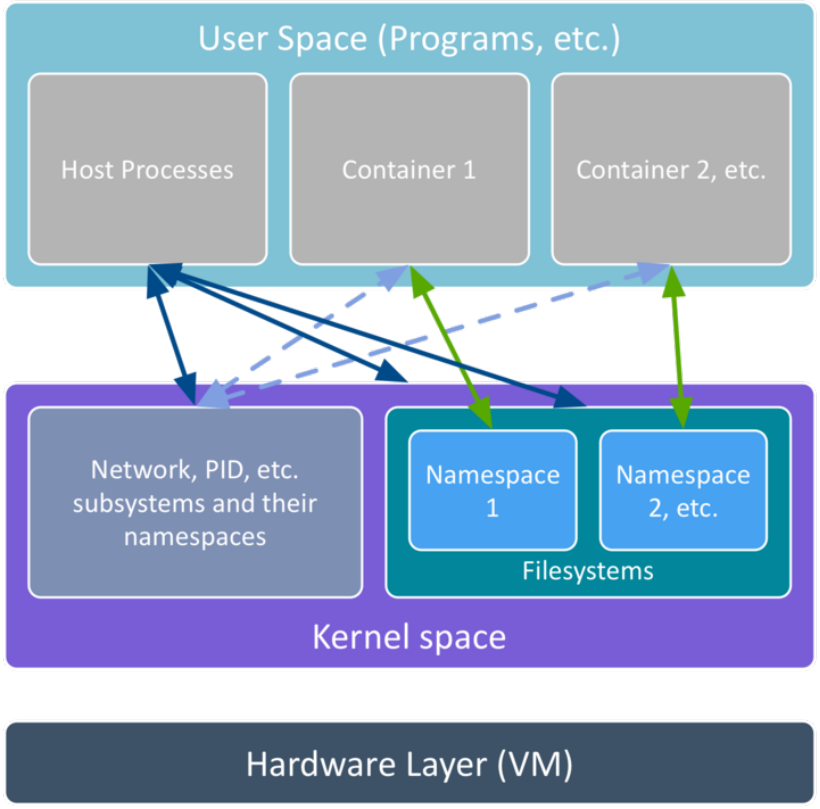
*Uses the kernel on the host operating system to run multiple guest instances*

- Each guest instance is a container
- Each container has its own
  - Root filesystem
  - Processes
  - Memory
  - Devices
  - Network Ports

# Isolation with Namespaces

*Namespaces - Limits what a container can see (and therefore use)*

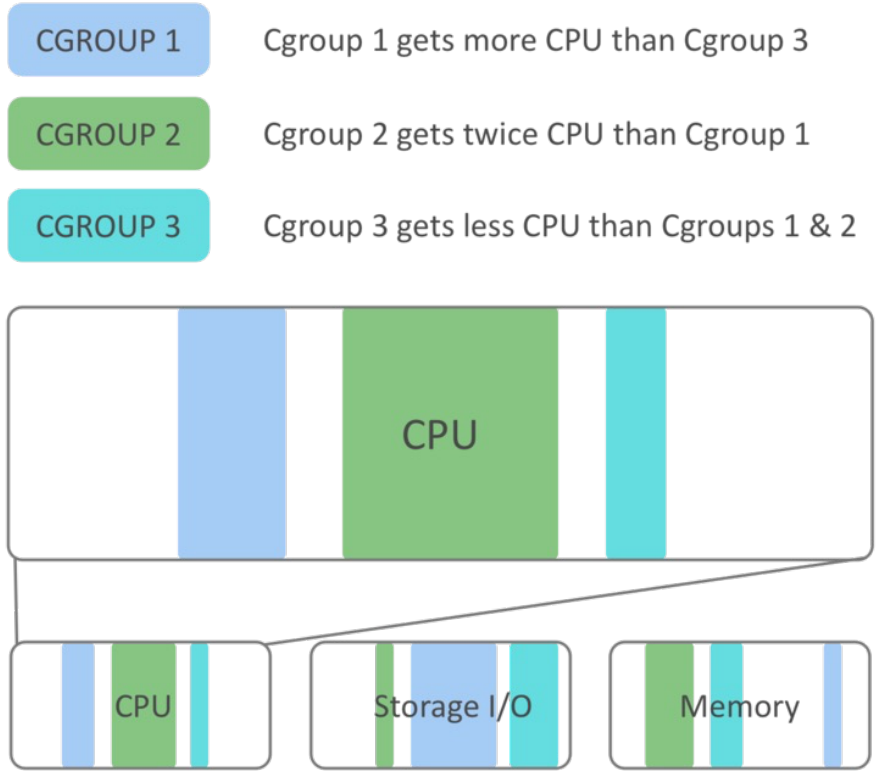
- Namespace wrap a global system resource in an abstraction layer
- Processes running in that namespace think they have their own, isolated resource
- Isolation includes:
  - Network stack
  - Process space
  - Filesystem mount points
  - etc.



# Isolation with Control group (Cgroups)

*Cgroups - Limits what a container can use*

- Resource metering and limiting
  - CPU
  - MEM
  - Block/IO
  - Network
- Device node (/dev/*) access control



# Container Use Cases



# DevOps

Developers

Focus on applications inside the container

Operations

Focus on orchestrating and maintaining containers in production

# Container Use Cases

**Development**

- Allows the ability to define the entire project configuration and tear-down/recreate it easily
- Supports multiple versions of application simultaneously

# Container Use Cases

**Development**

- Allows the ability to define the entire project configuration and tear-down/recreate it easily
- Supports multiple versions of application simultaneously

**Test Environments**

- Same container that developers run is the container that runs in test lab and production – includes all dependencies
- Well formed API allows for automated building and testing of new containers

# Container Use Cases

**Development**

- Allows the ability to define the entire project configuration and tear-down/recreate it easily
- Supports multiple versions of application simultaneously

**Test Environments**

- Same container that developers run is the container that runs in test lab and production – includes all dependencies
- Well formed API allows for automated building and testing of new containers

**Micro-Services**

- Design applications as suites of services, each written in the best language for the task
- Better resource allocation
- One container per microservice vs. one VM per microservice
- Can define all interdependencies of services with templates

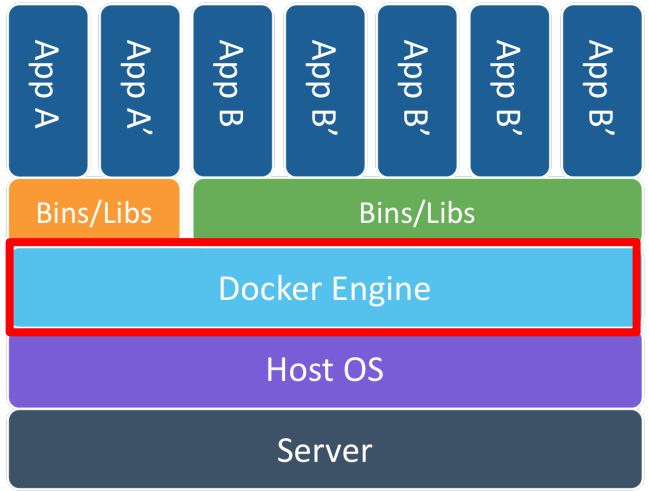
# The Docker Platform Components

# Docker Engine

Docker Engine | *Docker Registry* | Docker Compose | Docker Swarm

*Lightweight runtime program to **build, ship, and run Docker containers***

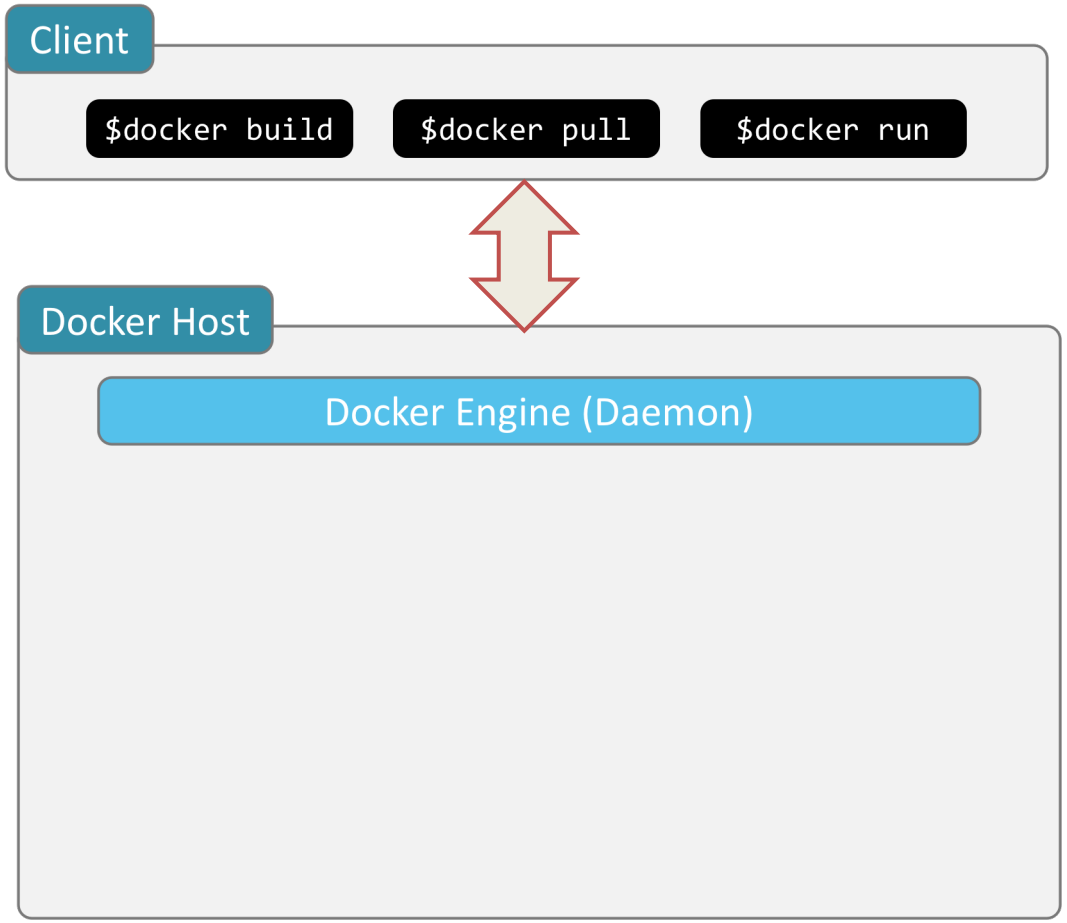
- Also known as **Docker Daemon**
- Uses Linux Kernel namespaces and control groups
  - **Linux Kernel (>= 3.10)**
- Namespaces provide an isolated workspace



# Docker Engine

Docker Engine | *Docker Registry* | Docker Compose | Docker Swarm

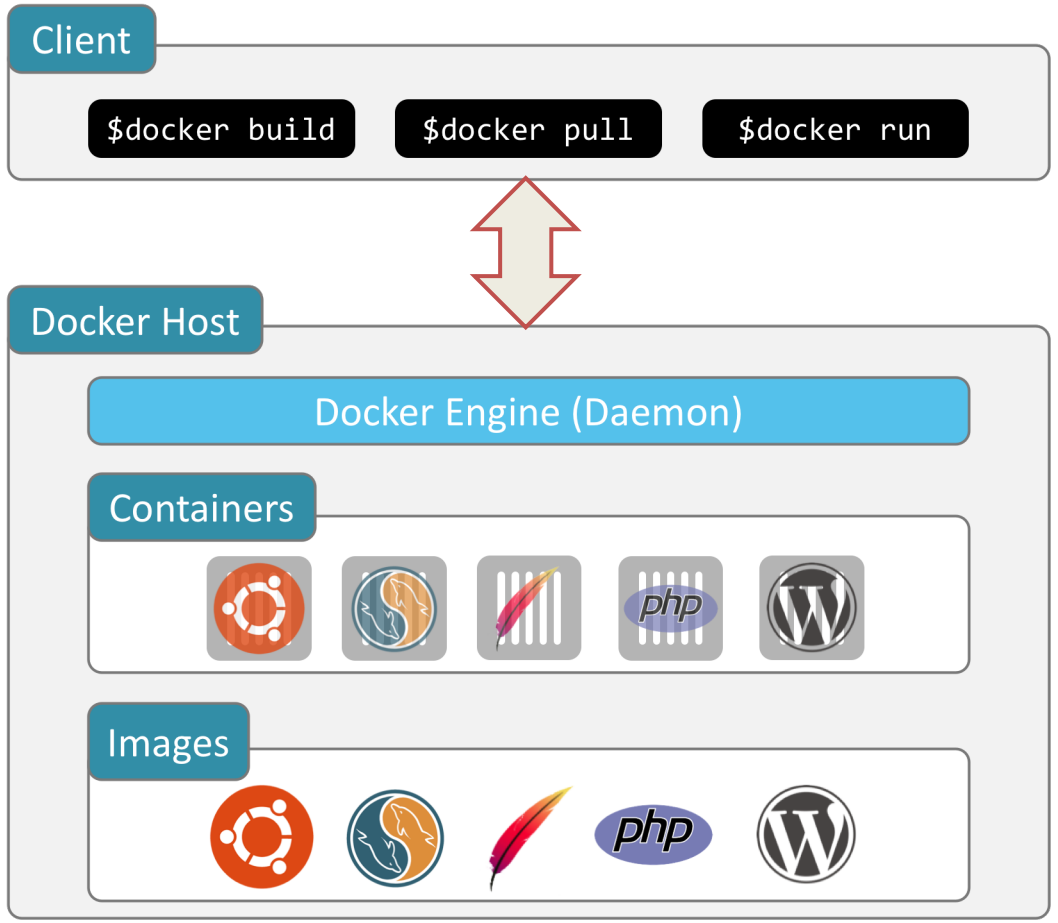
- The Docker Client is the `docker` binary
  - Primary interface to the Docker Host
- Accepts commands and communicates with the Docker Engine (Daemon)



# Docker Engine

Docker Engine | *Docker Registry* | Docker Compose | Docker Swarm

- Lives on a Docker host
- Creates and manages containers on the host



# Docker Registry

Docker Engine | *Docker Registry* | Docker Compose | Docker Swarm

*Image Storage & Retrieval System*

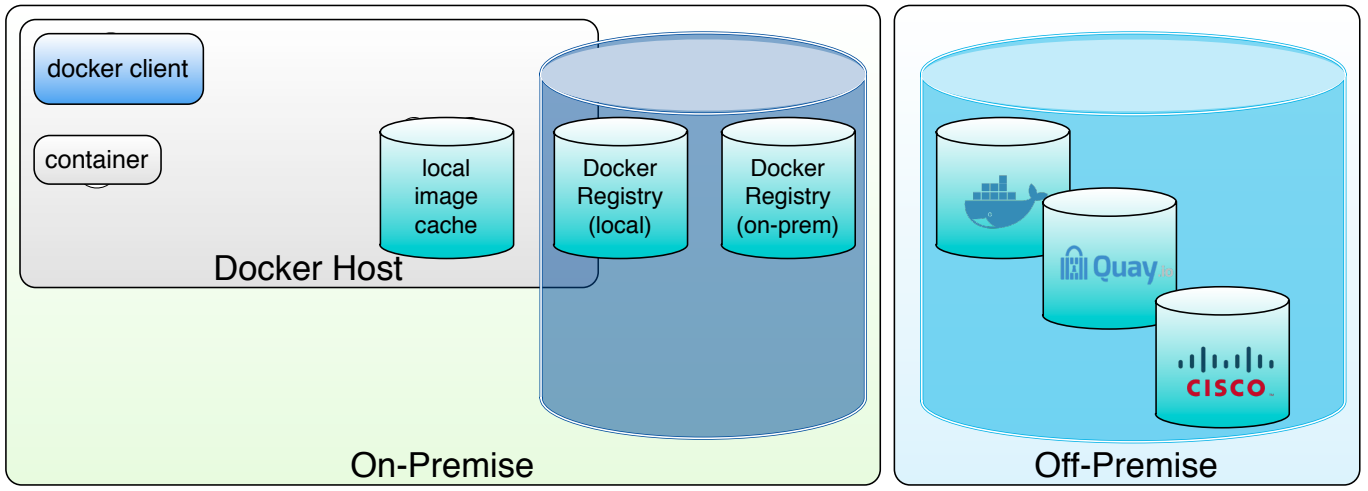

- Docker Engine **Pushes** Images to a Registry
- Version Control
- Docker Engine **Pulls** Images to Run

# Docker Registry

Docker Engine | *Docker Registry* | Docker Compose | Docker Swarm

**Types of Docker Registries**

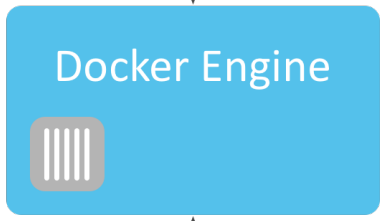
- Local Docker Registry
  (On Docker Host)
- Remote Docker Registry
  (On-Premise/Off-Premise)
- Docker Hub
  (Off-Premise)



# Docker Engine and Registry

Docker Engine | *Docker Registry* | Docker Compose | Docker Swarm

Manage & Monitor




Push/Pull


## Docker Engine:

- **Pushes** Images to a Registry
- **Pulls** Images to Run

# Docker Registry

Docker Engine | *Docker Registry* | Docker Compose | Docker Swarm

**The registry and engine both present APIs**

- All of Docker's functionality will utilize these APIs

- RESTFUL API
- Commands presented with Docker's CLI tools can also be used with curl and other tools

# Docker Compose

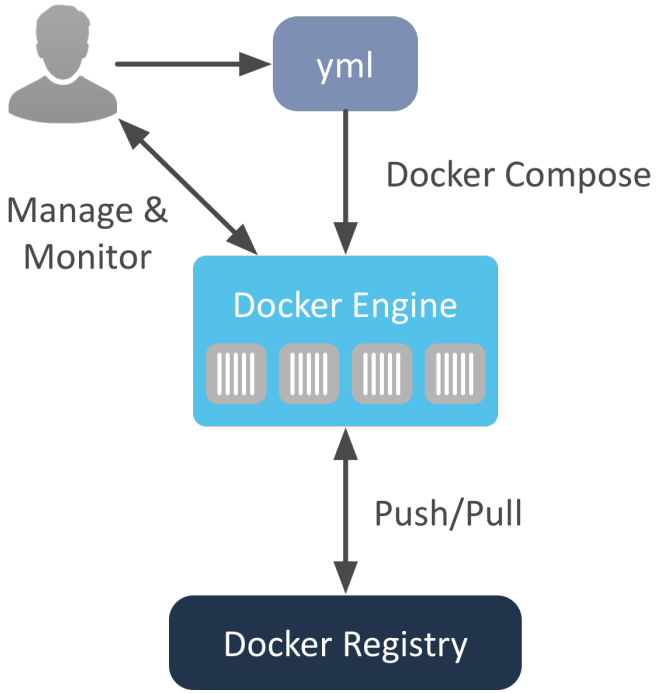
Docker Engine | *Docker Registry* | **Docker Compose** | Docker Swarm

*Tool to create and manage multi-container applications*

- Applications defined in a single file:

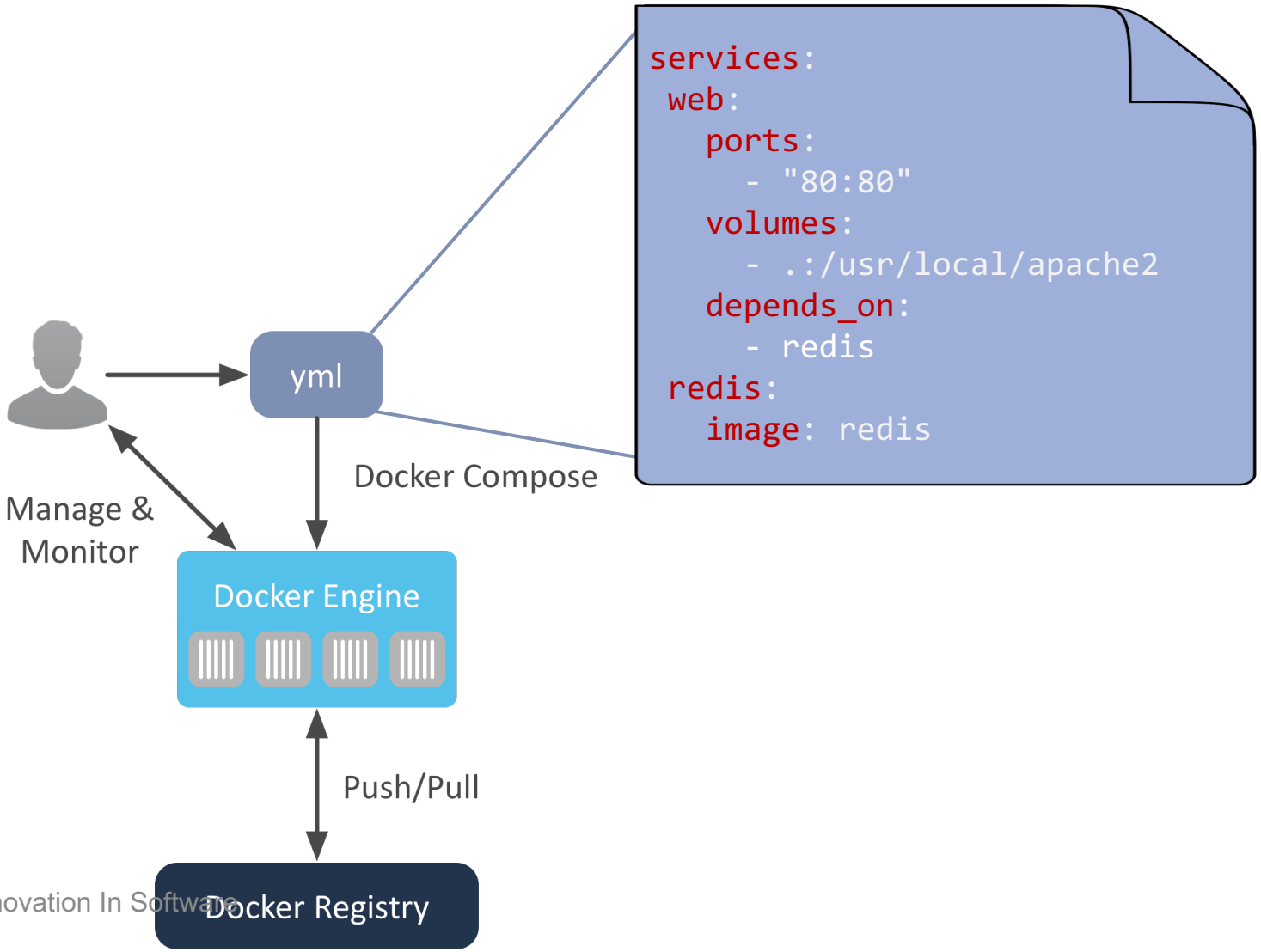
  **docker-compose.yml**

- Transforms applications into individual containers that are linked together
- Compose will start all containers in a single command



# Docker Compose

Docker Engine | *Docker Registry* | Docker Compose | Docker Swarm

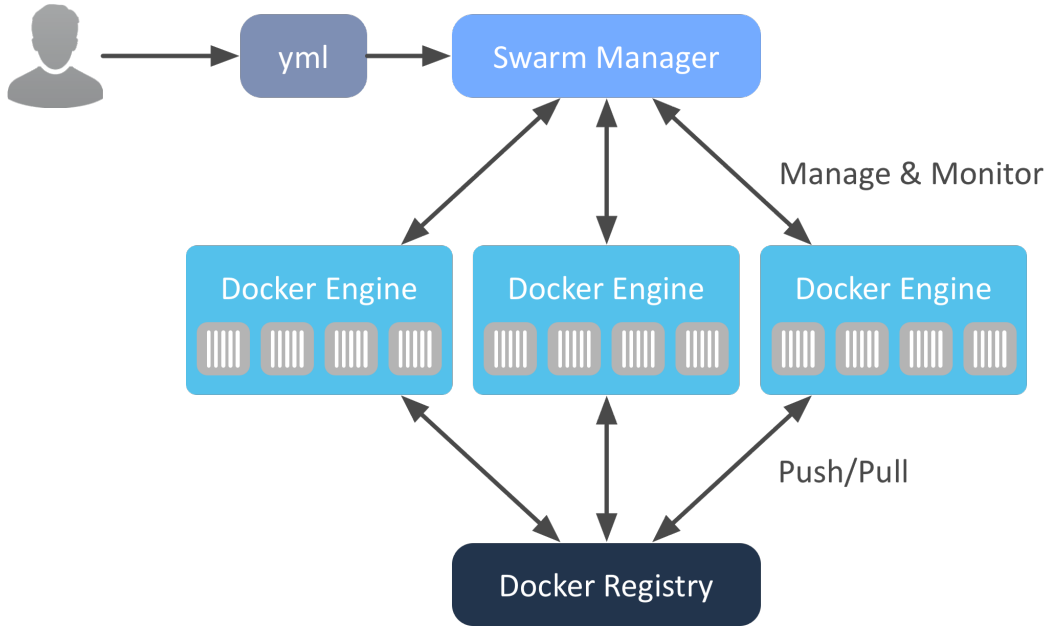


# Docker Swarm

Docker Engine | *Docker Registry* | Docker Compose | **Docker Swarm**

*Clusters Docker hosts and schedules containers*

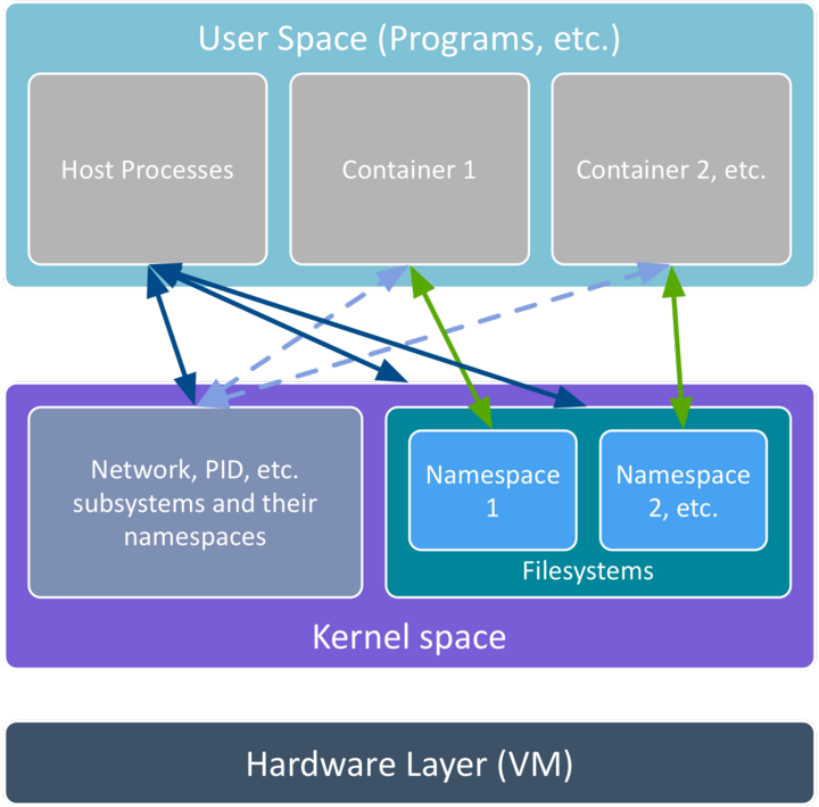
- Native Clustering for Docker
  - Turn a pool of Docker hosts into a single, virtual host
  - Serves the standard Docker API



# Isolation with Namespaces

*Namespaces - Limits what a container can see (and therefore use)*

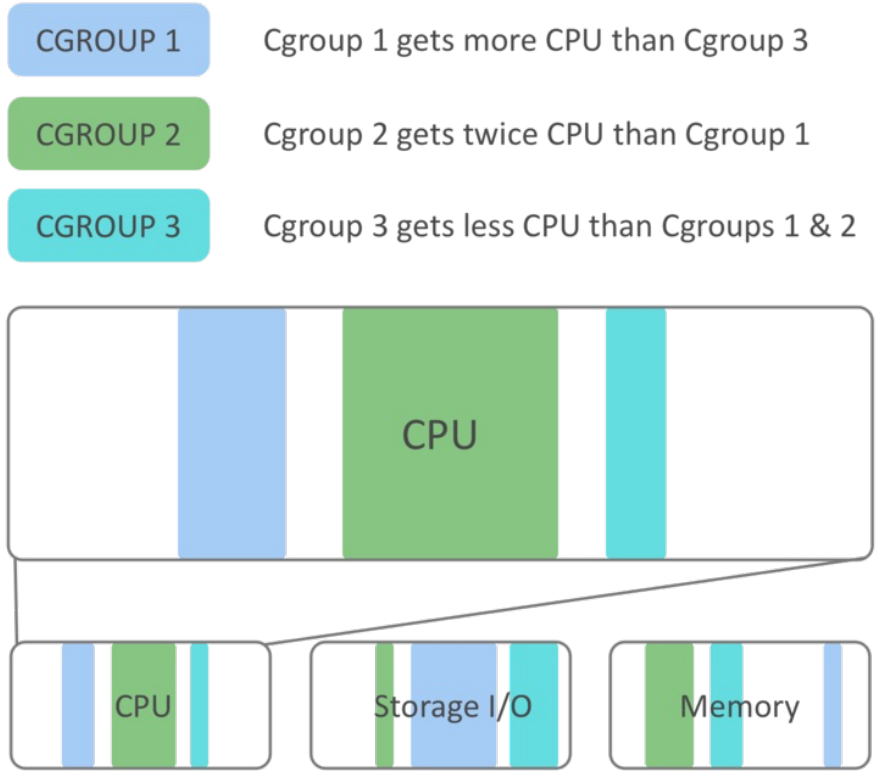
- Namespace wrap a global system resource in an abstraction layer
- Processes running in that namespace think they have their own, isolated resource
- Isolation includes:
  - Network stack
  - Process space
  - Filesystem mount points
  - etc.



# Isolation with Control group (Cgroups)

*Cgroups - Limits what a container can use*

- Resource metering and limiting
  - CPU
  - MEM
  - Block/IO
  - Network
- Device node (/dev/*) access control



# Docker Lab Setup

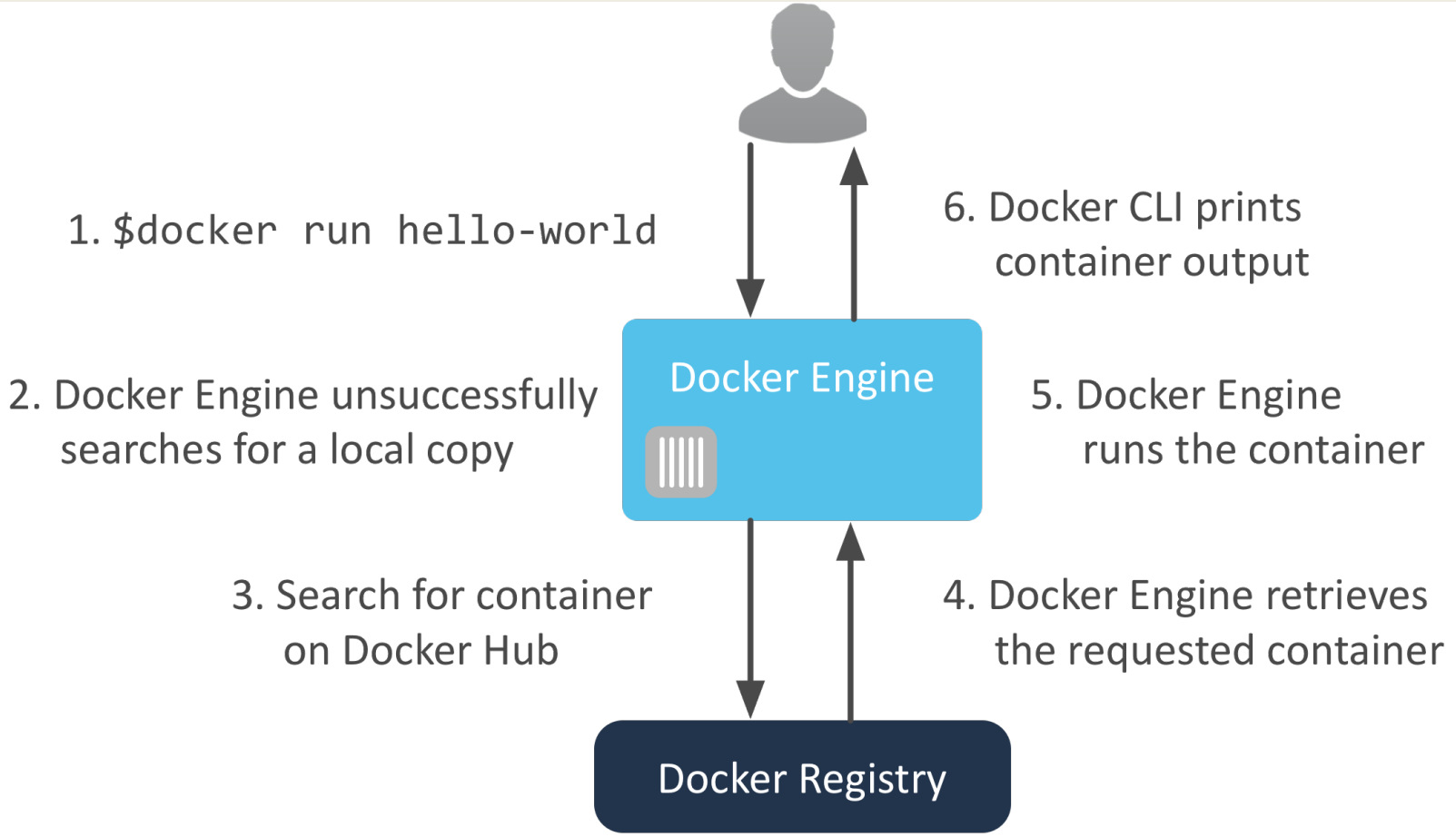


# Run a Container



# Container Review

What does running a container require?



# Run a Container

In your Lab Environment, execute the following command: `$ docker run hello-world`

The container will run and display the following:

```
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
ad01a741289f: Pull complete
Digest:
sha256:af451bc5c4d6e87b64303f707f7bd98debd7fd3175a676c08f4bf46cb48813d4
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working
correctly.
```

# Run a Container

Containers can accept additional arguments

In your Lab Environment, execute the following command:

```
$ docker run docker/whalesay \
          cowsay 'Go Docker'
```

The container will run and display the following:

```
Unable to find image 'docker/whalesay:latest' locally
latest: Pulling from docker/whalesay

e190868d63f8: Pull complete
909cd34c6fd7: Pull complete
0b9bfabab7c1: Pull complete
a3ed95caeb02: Pull complete
00bf65475aba: Pull complete
c57b6bcc83e3: Pull complete
8978f6879e2f: Pull complete
8eed3712d2cf: Pull complete
Digest:
sha256:178598e51a26abbc958b8a2e48825c90bc22e641de3d31e18aaf55f3258ba93b
Status: Downloaded newer image for docker/whalesay:latest
```

# Run a Container

***docker run*** also accepts additional arguments

In your Lab Environment, execute the following command:

```
$ docker run -d docker/whalesay \
          cowsay 'Detached'
```

The container will run and display the following:

```
3fa9d2d4f5ad3169d2e763f7ef48436c15ca4f8749a044e3017e6cb96cc60a2d
```

# Run a Container

What just happened?

```
$ docker run -d docker/whalesay \
          cowsay 'Detached'
```

# Run a Container

```
$ docker run -d docker/whalesay \
          cowsay 'Detached'
```

The *–d [detach]* flag instructs the Docker Engine to run the container in the background.

```
docker run –help

Usage:  docker run [OPTIONS] IMAGE [COMMAND] [ARG...]
Run a command in a new container

Options:
-d, --detach           Run container in background and print container ID
```

# Run a Container

## Why Run a Container in the Background?

# Discussion

## Why Run a Container in the Background?

- Containers that do not require interaction can be started and left to run in their own process

# Monitor Docker – Native Tools



# Native Monitoring Tools

These following tools provide real-time metrics on a container or a set of containers.

**`docker ps <switch>`**

- list containers

**`docker logs <container>`**

- displays console output of the container

**`docker top <container>`**

- lists all the processes running in a container

**`docker stats <container>`**

- streams real time stats of containers

**`docker inspect <container>`**

- displays detailed container configuration

# Docker ps (list containers)

Show running Docker containers

```
$ docker ps

CONTAINER ID        IMAGE                          COMMAND
ece82d202fd6        grafana/grafana:2.0.2          "/usr/sbin/grafana-se“
2be9d338d8e4        google/cadvisor                "/usr/bin/cadvisor -s"
ffa57323dad8        tutum/influxdb:0.8.8           "/run.sh"
```

Show Docker containers id

```
$ docker ps -q
ece82d202fd6
2be9d338d8e4
ffa57323dad8
c5817eea595e
```

Show all Docker containers

```
$ docker ps –a

CONTAINER ID        IMAGE                          COMMAND
ece82d202fd6        grafana/grafana:2.0.2          "/usr/sbin/grafana-se“
2be9d338d8e4        google/cadvisor                "/usr/bin/cadvisor -s"
ffa57323dad8        tutum/influxdb:0.8.8           "/run.sh"
c5817eea595e        nginx/nginx-php                "/usr/local/bin/run"
```

# Docker ps (list containers)

In your Lab Environment, execute the following command:

```
$ docker ps
```

The command will display the output:

```
CONTAINER ID   IMAGE   COMMAND   CREATED   STATUS   PORTS   NAMES
```

This is expected since we have no running containers…

# Docker ps (list containers)

What about all containers?

In your Lab Environment, execute the following command:

```
$ docker ps -a
```

The command will display output similar to:

```
CONTAINER ID   IMAGE              COMMAND                  CREATED              STATUS                          PORTS     NAMES
f774f8618226   hello-world        "/hello"                 7 seconds ago        Exited (0) 6 seconds ago                  timely_terapin
3fa9d2d4f5ad   docker/whalesay    "cowsay 'Go Docker'"     50 minutes ago       Exited (0) 50 minutes ago                 ecstatic_feynman
8f5e737d15bf   docker/whalesay    "cowsay 'Detached'"      About an hour ago    Exited (0) 59 minutes ago                 happy_leavitt
```

The process we executed in these containers finished, so the containers exited

Make note (copy) a CONTAINER ID

# Docker logs <container>

## Display the Console Output of a Container

```
$ docker logs f774f8618226

Hello from Docker!
This message shows that your installation appears to be working correctly.
To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.
To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash
Share images, automate workflows, and more with a free Docker Hub account:
 https://hub.docker.com
For more examples and ideas, visit:
 https://docs.docker.com/engine/userguide/
```

# Docker logs <container>

Try it!

In your Lab Environment, execute the following command:

```
$ docker logs <container_id>
```

The command will display the output:

# Docker top <container>

Running the **`docker top`** command on all host container

- Flags used with regular top can be used **`–aux -faux`**

```
$ docker top –help
Usage:  docker top [OPTIONS] CONTAINER [ps OPTIONS]
Display the running processes of a container
--help=false       Print usage
```

# Docker top <container>

Try it!

In your Lab Environment, execute the following command:

```
$ docker run -d busybox top
```

This will pull and run the latest build of busybox and run the ‘top’ command.

Once the container has started, the console will display the Container ID:

```
7ed45bb572d0886686c5d9ac7484094716cf0c581087d6499193a799a3ed6345
```

# Docker top <container>

Now that we have a running container:
In your Lab Environment, execute the following command: `$ docker top <container_id>`

The console will display the running process in the container:

```
PID                 USER                TIME                COMMAND
3372                root                0:03                top
```

# Docker stats <container>

**docker stats** command

```
$ docker stats --help
Usage:  docker stats [OPTIONS] CONTAINER [CONTAINER...]
Display a live stream of container(s) resource usage statistics
--help=false         Print usage
--no-stream=false    Disable streaming stats and only pull the first
result
```

**docker stats** command on a single container

```
$ docker stats <container_id>

CONTAINER           CPU %               MEM USAGE / LIMIT
cdfd04b5aeef        0.15%               29.52 MB / 2.098 GB

MEM %               NET I/O             BLOCK I/O
1.41%               6.684 kB / 648 B    0 B / 49.15 kB
```

# Docker stats <container>

Try it!

In your Lab Environment, execute the following command:

```
$ docker stats <container_id>
```

The console will display the real-time stats of the container:

```
CONTAINER
CPU %               MEM USAGE / LIMIT     MEM %               NET I/O
BLOCK I/O           PIDS
7ed45bb572d0886686c5d9ac7484094716cf0c581087d6499193a799a3ed6345
0.25%               147 MiB / 1.954 GiB   7.35%               648 B /
648 B       0 B / 0 B           24
```

# Docker inspect <container>

## **docker inspect** command

```
$ docker inspect --help
Usage:    docker inspect [OPTIONS] CONTAINER|IMAGE|TASK
Return low-level information on a container, image or task
  -f, --format       Format the output using the given go template
  --help             Print usage
  -s, --size         Display total file sizes if the type is container
  --type             Return JSON for specified type, (e.g image, container or task)
```

## **docker inspect** command on a single container

```
$ docker inspect 506fe8cf11bb
[
    {
        "Id": "506fe8cf11bbbb74bf9db26f6561f218ec437cc2b8267d214affb31493e4c251",
        "Created": "2016-07-11T15:51:34.148392848Z",
```

# Docker inspect <container>

Try it!

In your Lab Environment, execute the following command:

```
$ docker inspect <container_id>
```

The console will display the configuration details of the container

```
[
    {
        "Id":
"8404dd7fcb18ba49e6d839bf87ce877297566cfac6ca9a6f0c93846e2bf5e366",
        "Created": "2016-09-09T14:46:28.911819016Z",
        "Path": "top",
```

# Start, Stop & Remove Containers



# Manage Containers

The following tools allow management of containers

**`docker stop <container>`**

- Stop a container

**`docker start <container>`**

- Start a container

**`docker rm <container>`**

- Remove a container

# docker stop <container>

Stop a Docker container

```
$ docker stop --help

Usage:  docker stop [OPTIONS] CONTAINER [CONTAINER...]


Stop one or more running containers


Options:
      --help       Print usage
  -t, --time int   Seconds to wait for stop before killing it (default
10)
```

# docker stop <container>

Try it!

In your Lab Environment, execute the following command:

```
$ docker stop <container_id>
```

This will stop the container.  The console will acknowledge with the Container ID

```
7ed45bb572d0
```

# docker start <container>

Start a Docker container

```
$ docker start --help

Usage:  docker start [OPTIONS] CONTAINER [CONTAINER...]


Start one or more stopped containers


Options:
  -a, --attach               Attach STDOUT/STDERR and forward signals
      --detach-keys string   Override the key sequence for detaching a
container
      --help                 Print usage
  -i, --interactive          Attach container's STDIN
```

# docker start <container>

Try it!
In your Lab Environment, execute the following command:

```
$ docker start <container_id>
```

This will start the container.

The console will acknowledge with the Container ID.

```
7ed45bb572d0
```

# docker rm <container>

Remove a Docker container

```
$ docker rm --help

Usage:  docker rm [OPTIONS] CONTAINER [CONTAINER...]


Remove one or more containers


Options:
  -f, --force     Force the removal of a running container (uses
SIGKILL)
      --help      Print usage
  -l, --link      Remove the specified link
  -v, --volumes   Remove the volumes associated with the container
```

# docker rm <container>

Try it!
In your Lab Environment, execute the following command:

```
$ docker rm <container_id>
```

This will remove the container.

The console will acknowledge with the Container ID.

```
7ed45bb572d0
```

But what if I wanted to delete ALL containers?
In your Lab Environment, execute the following command: `$ docker rm $(docker ps -a -q)`

This will remove the container. The console will acknowledge with all the Container IDs.

```
1028e737f3ab
b72a77b7a22f
f774f8618226
3fa9d2d4f5ad
8f5e737d15bf
```

***Use with Caution***

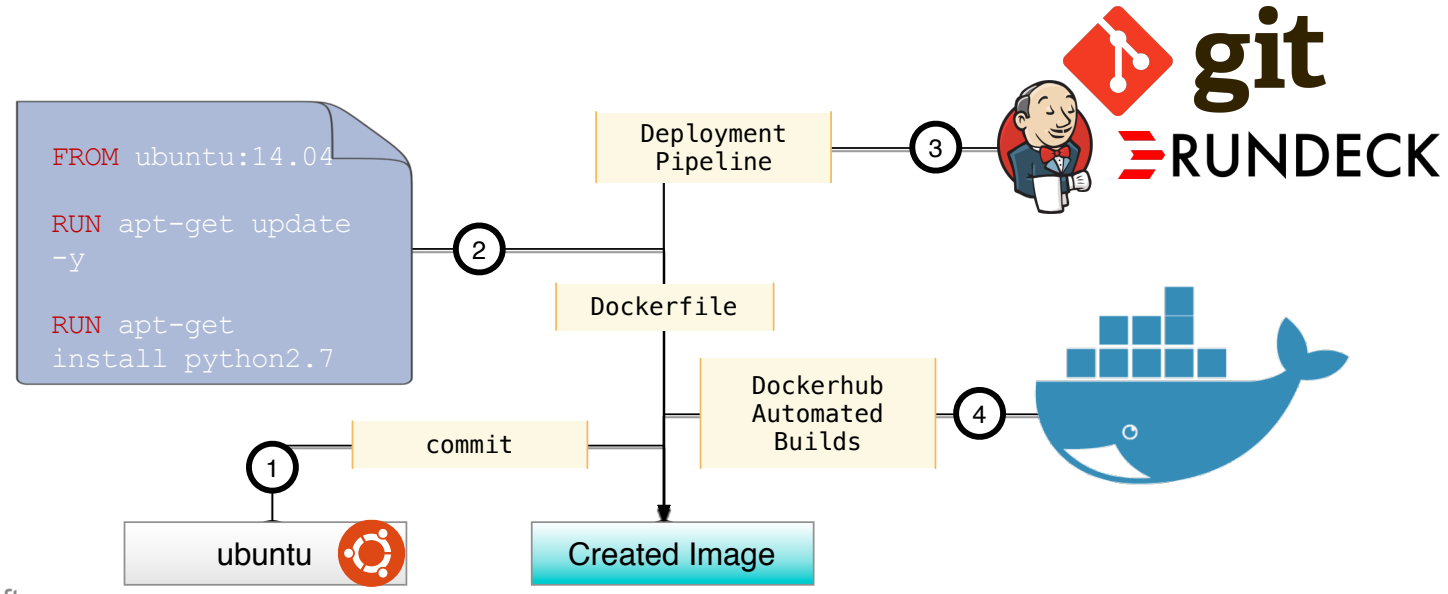
# Docker Images



# Methods to Create an Image

Images can be created from:

1. The command-line
2. An image from a Dockerfile
3. An image from a deployment pipeline
4. An image from Docker automated builds



# Anatomy of Docker Image

# Docker Image Terminology

Image

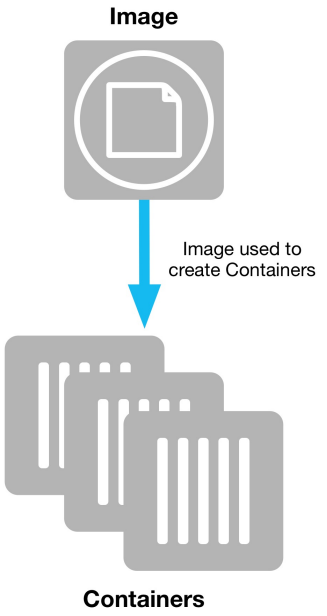
Union Filesystem

Dockerfile

Container

Hierarchy of files, with meta-data for how to create/run a container

- Read only template used to create containers
- Can be exported or modified to new images
- Created manually or through automated processes
- Stored in a Registry (Docker Hub, Docker Trusted Registry, etc.)



# Docker Image Terminology

Image

Union Filesystem

Dockerfile

Container

**UnionFS –** Used by Docker to layer images

- Not a distributed File System

# Docker Image Terminology

Image

Union Filesystem

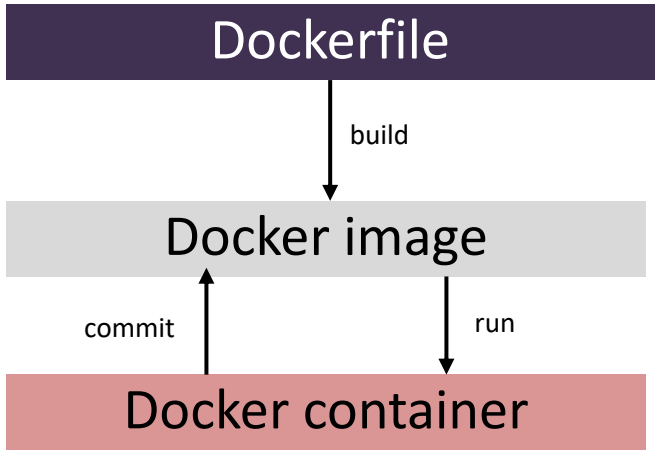
Dockerfile

Container

Configuration File (script) for creating images.

Defines:

- Existing image to be the starting point
- Set of instructions to augment that image (each of which results in a new layer of the file system)
- Meta-data such as ports exposed
- The command to execute when the image is run



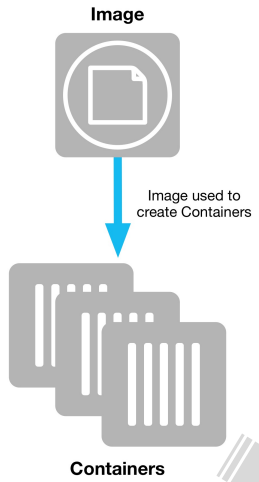
# Docker Image Terminology

Image

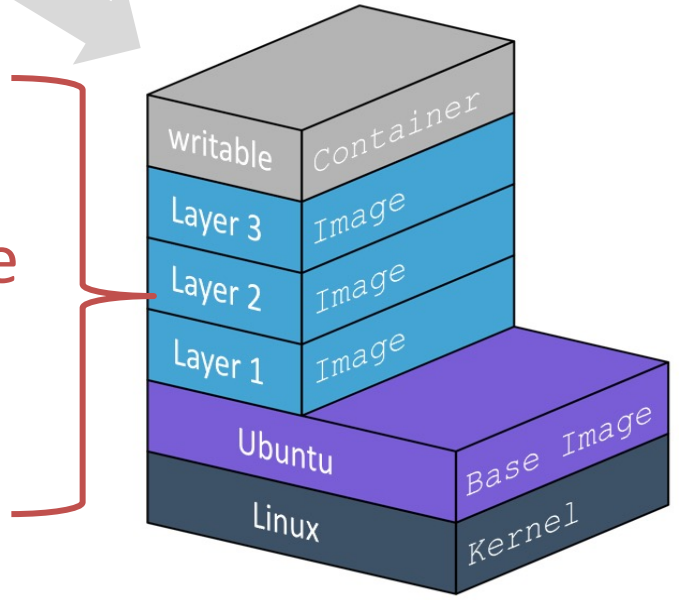
Union Filesystem

Dockerfile

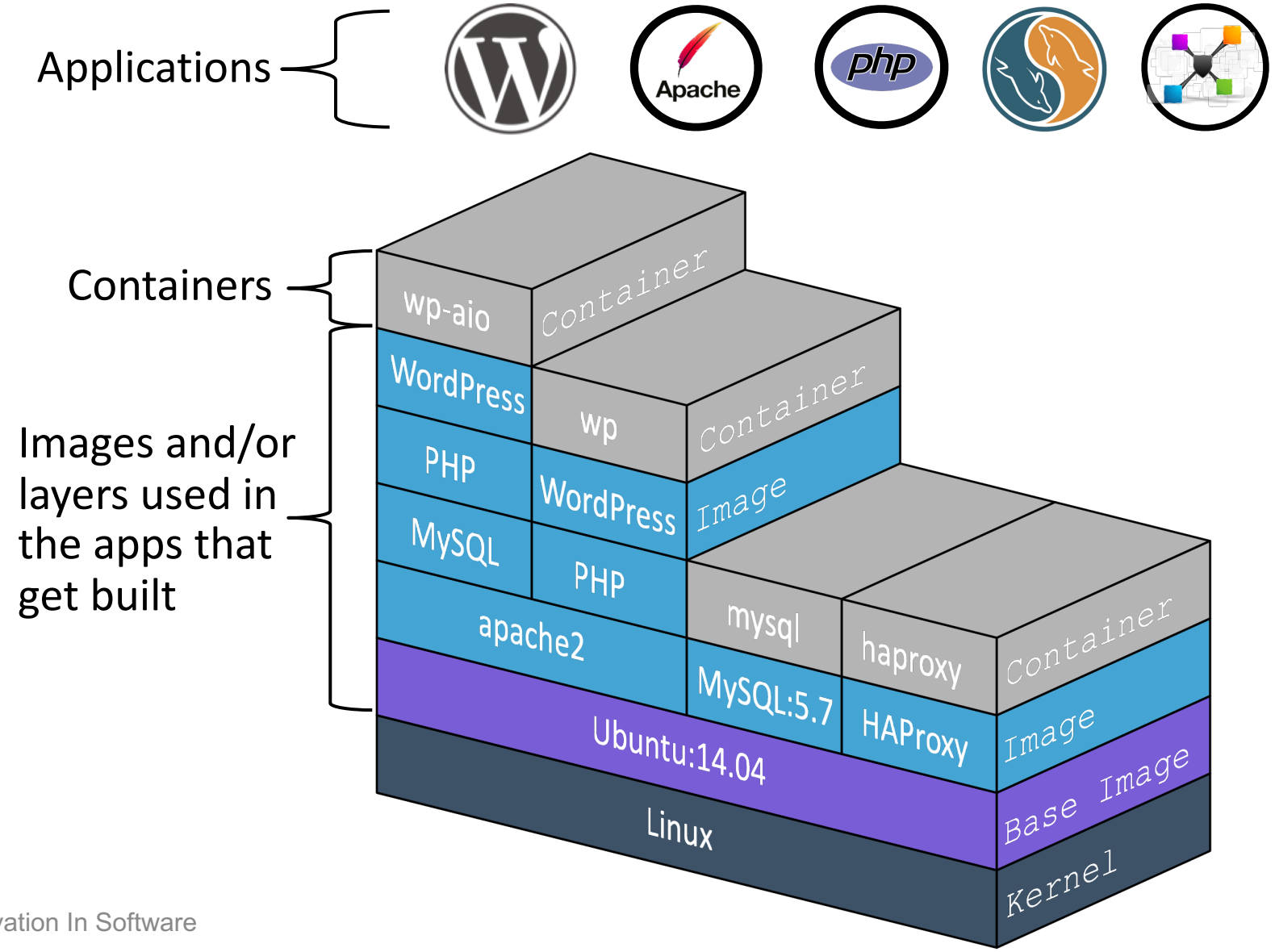
Container




Runtime instance of an image plus a read/write layer



# Applications, Containers, and Images



# Union Files System



# Image/Container Interactions

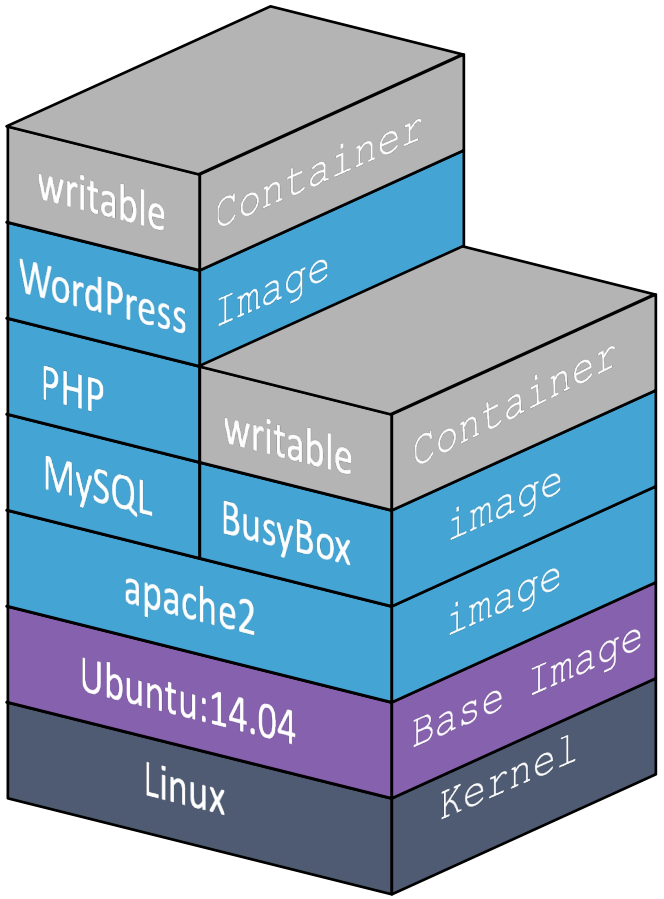
**Pull** an image from a Docker Registry

**Run** a container from an image

**Add** a file to a running container – Example, the image requires an additional file called index.html

**Change** to a running container – Example, the image requires an update of the index.php file

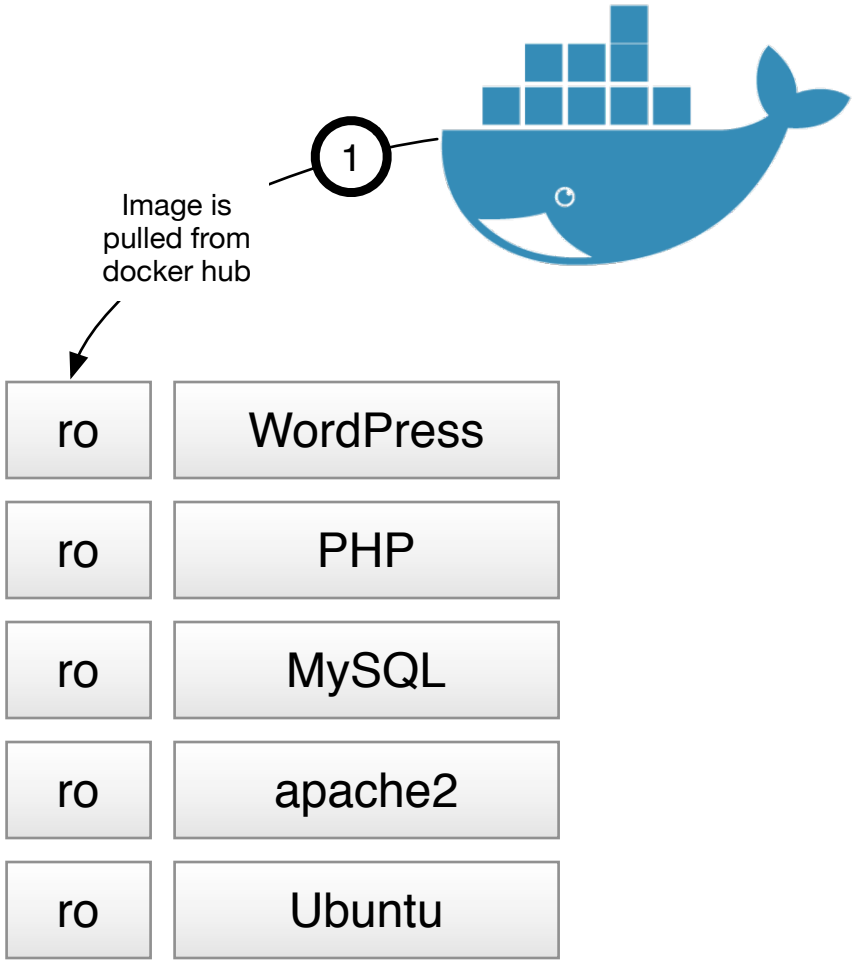
**Delete** files from a running container – All containers share the same host kernel



# Pull the WordPress AIO

The WordPress All-In-One image is pulled from Docker Hub

```
$docker inspect wordpress-aio
```



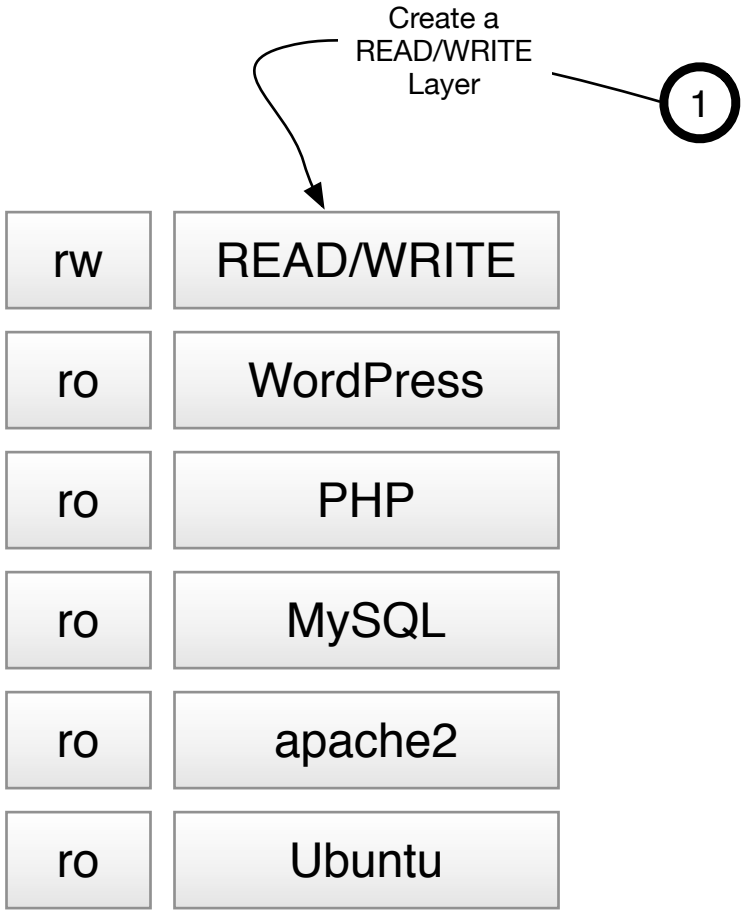

## Docker images are:

- Pulled or created
- Stored in a local image repo
- READ-ONLY
- The basis of all containers

# Run the WordPress AIO

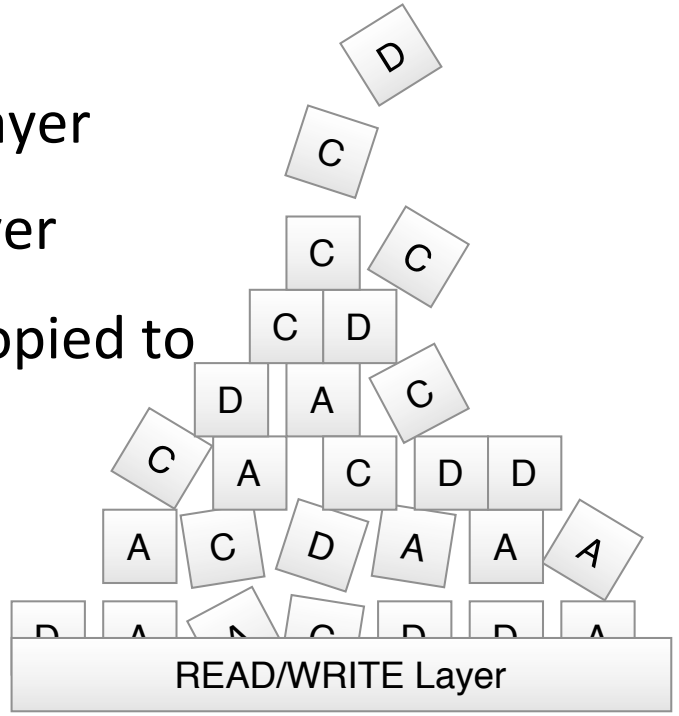
A container is made from the previously pulled WordPress AIO image

1. `$docker run wordpress-aio`
2. Container created
3. READ/WRITE layer created
4. All **ADD**, **CHANGE**, **DELETE** are committed to the READ/WRITE Layer



# Key Takeaways

- **Images** are:
  - Highly portable and readily available
  - Reusable for many deployments
- **ADD** – ADD DATA to READ/WRITE Layer
- **CHANGE –** ADD DATA to READ/WRITE Layer
- **DELETE –** ADD DATA to READ/WRITE Layer
- **ALL** ADDs, CHANGEs, and DELETEs are copied to the READ/WRITE Layer



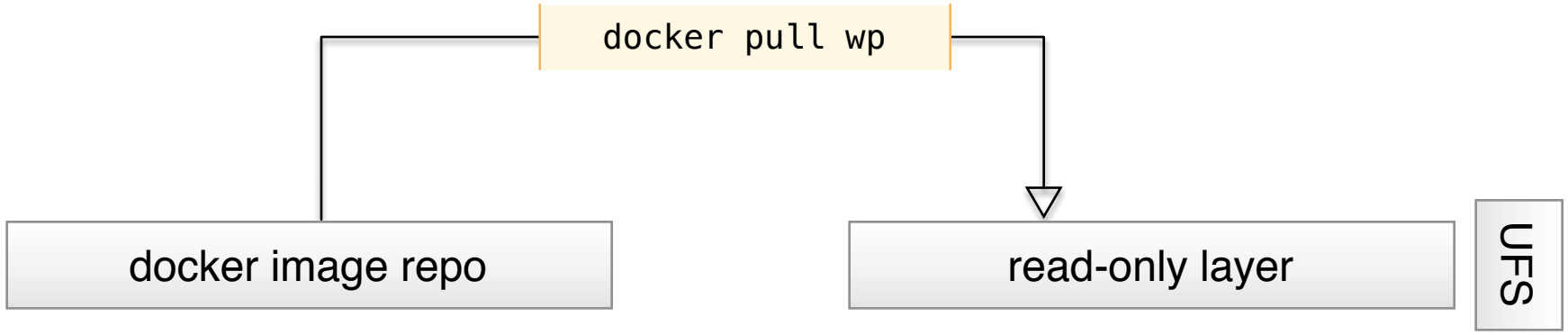
# Commits on Containers



# Union Filesystem

**docker pull**– The image is pulled from a Docker registry

```
$ docker image pull <image>
```

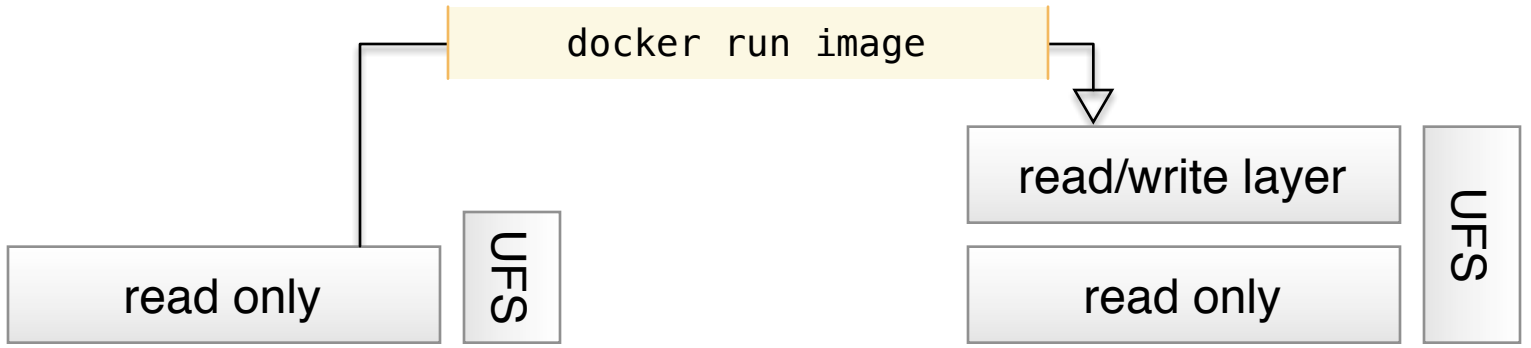


# Union Filesystem

**RUN** – Creates a read/write layer for the container to use

**ADDS, CHANGES, DELETES** are written to the read/write layer, the COPY ON WRITE layer.

```
$ docker container run --options <image>
```



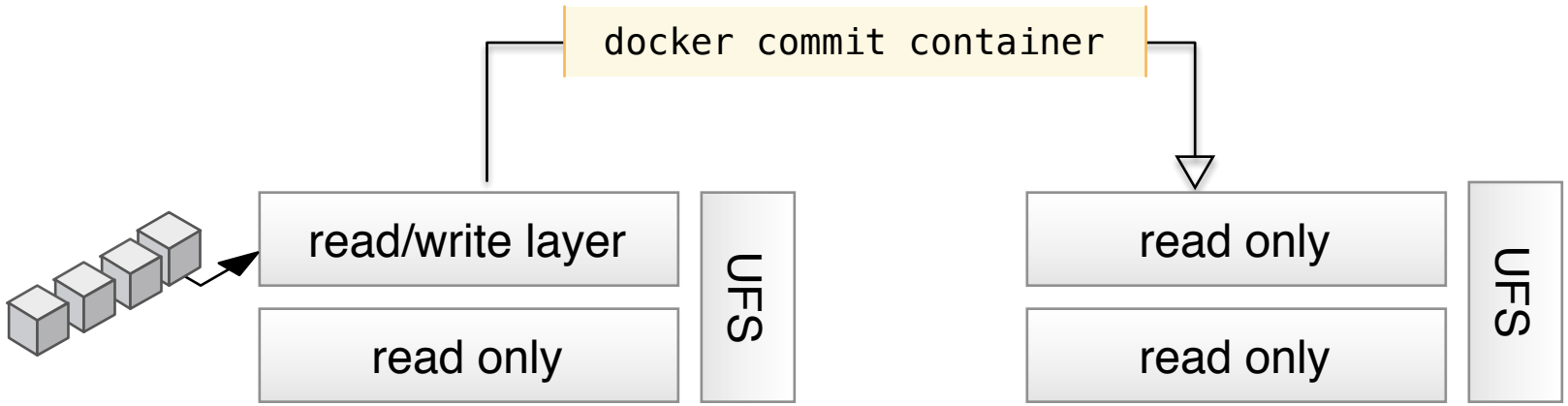
# Union Filesystem

## Container is committed

- When a container is committed, the READ/WRITE layer becomes a READ ONLY layer

```
docker commit -m “comment” -a “author” demo
author/demo:new
```

```
$ docker commit -m “comment” -a “author” wp <repo>/wp:tag
```

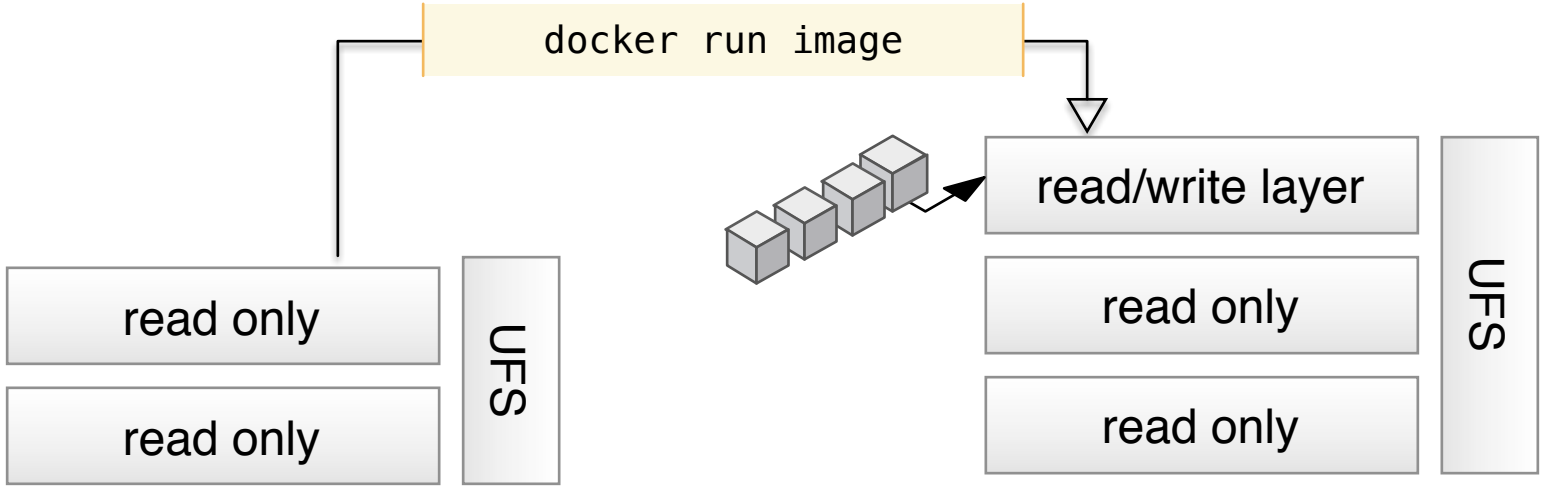


# Union Filesystem

**A Container run from the newly created image**

- The UFS READ/WRITE READ-ONLY process starts again
- A new READ/WRITE layer is created
- Only when changes are made to to R/W UnionFS is data written

```
$ docker container run --options <repo>/wp:tag
```



# Lab: Build Docker Images (Manually)



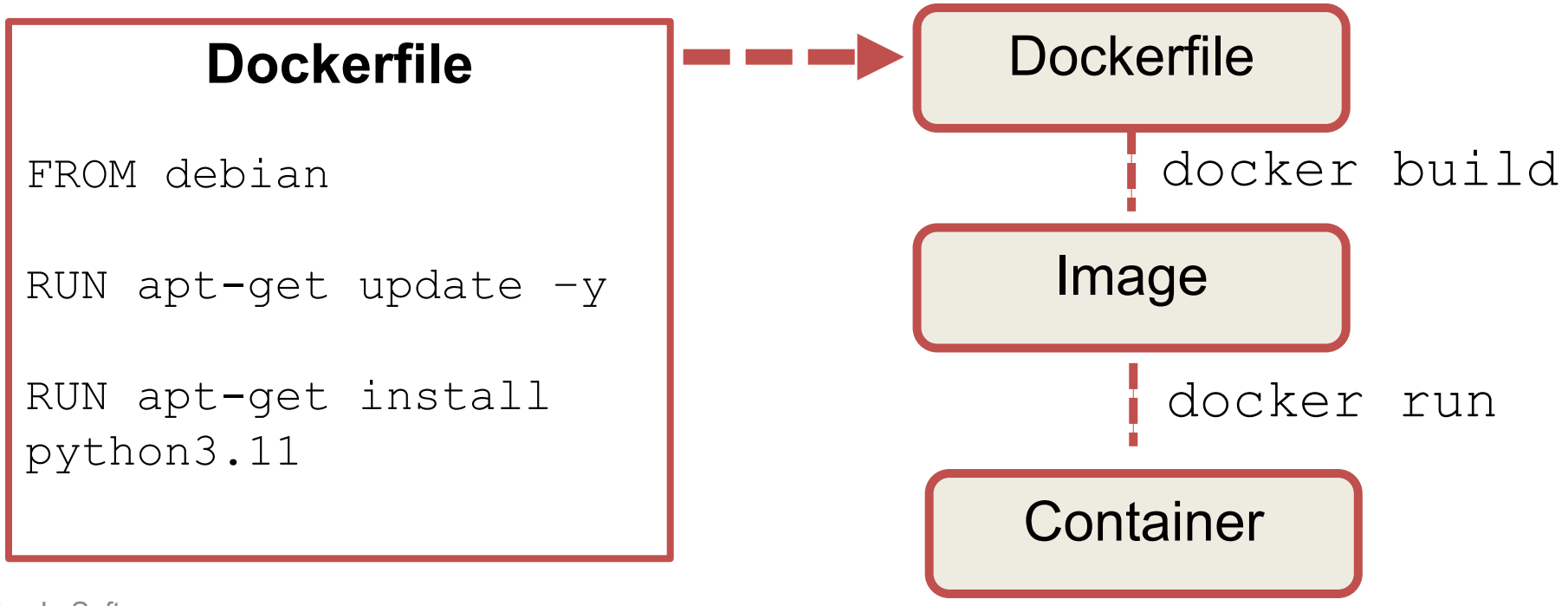
# Build Docker Images (Dockerfile)



# Dockerfile

Docker can build images automatically by reading the instructions from a **Dockerfile**

E.g. dockerfile that installs python 3.11 on top of debian image



# Dockerfile Syntax

Dockerfiles have an easily readable format

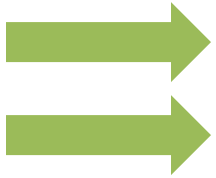
# comments
INSTRUCTIONS
arguments

```
#My first Dockerfile
FROM ubuntu:20.04
RUN apt-get update
```

Lines starting with the “#” are comments
Comments can be placed anywhere
Comments help tell others what is intended

# Dockerfile Syntax

```
# comments
INSTRUCTIONS
arguments
```



```
#My first Dockerfile
FROM ubuntu:20.04
RUN apt-get update
```

INSTRUCTIONS should be CAPITALIZED

INSTRUCTIONS are run during image build

The first INSTRUCTION is always FROM

# Dockerfile Syntax

# comments

INSTRUCTIONS

arguments

```
#My first Dockerfile
FROM ubuntu:20.04
RUN apt-get update
```

`arguments` are what gets fed to the builder

`arguments` can be run sequentially in the same `INSTRUCTION` with an escape

# Escape Parser Directive

- form # directive=value
- Windows uses “\” for directories

```
# escape=`
FROM microsoft/windowsservercode
SHELL [“powershell”,”-command”]
RUN New-Item -ItemType Directory `
    C:\Example
ADD Execute-MyCmdlet.psl c:\example
```

# Dockerfile Example

- Dockerfile takes the latest mongo image
- Updates it
- Installs an entrypoint script to streamline execution

```
FROM mongo:latest
RUN apt-get update && apt-get install -y dos2unix
EXPOSE 27017
COPY docker-entrypoint-init.sh /entrypoint-init.sh
RUN dos2unix /entrypoint-init.sh && \
  apt-get --purge remove -y dos2unix && \
  rm -rf /var/lib/apt/lists/*
RUN chmod ugo+x /entrypoint-init.sh
ENTRYPOINT ["/entrypoint-init.sh"]
CMD ["mongod"]
```

# Dockerfile Instructions

| Command | Description |
|---|---|
| # | Comment line |
| MAINTAINER | Provides name and contact info of image creator |
| FROM | Tells Docker which base image to build on top of (e.g. centos7) |
| COPY | Copies a file or directory from the build host into the build container |
| RUN | Runs a shell command inside the build container |
| CMD | Provides a default command for the container to run. May be overridden or changed |
| ADD | Copies new files, directories or remote file URLs |
| LABEL | Adds metadata to an image |

source: https://docs.docker.com/v1.8/reference/builder/

# Dockerfile Instructions

| Command | Description |
|---|---|
| VOLUME | Exposes any database storage area, configuration storage, or files/folders created by your Docker container |
| USER | Change to a non-root user |
| WORKDIR | Sets the working directory for any RUN, CMD, ENTRYPOINT, COPY or ADD instruction |
| ONBUILD | Executes after the current Dockerfile build completes. ONBUILD executes in any child image derived FROM the current image |
| EXPOSE | Informs Docker that the container will listen on the specified network ports at runtime |
| ENTRYPOINT | Allows you to configure a container that will run as an executable |
| ENV | Sets the environment variable <key> to the value |

source: https://docs.docker.com/v1.8/reference/builder/

# ADD – Not Recommended

```
ADD http://example.com/big.tar.xz /usr/src/things/
RUN tar -xJf /usr/src/things/big.tar.xz -C
/usr/src/things
RUN make -C /usr/src/things all
```

- Image size matters
- Using `ADD` to fetch packages from remote URLs INCREASES images sizes
  - Not recommended
- Use `curl` or `wget`
  - Gain ability to delete the files you no longer need after they've been extracted
  - Reduces unnecessary layers in your image

# Best Practice: Use Curl

```
RUN mkdir -p /usr/src/things \
&& curl -SL http://example.com/big.tar.xz \   | tar
-xJC /usr/src/things \
&& make -C /usr/src/things all
```

- Use `curl` instead of `ADD`
  - Allows for cleaning up the tar file after it's been extracted and application installed

# Build Docker Images (Docker build)

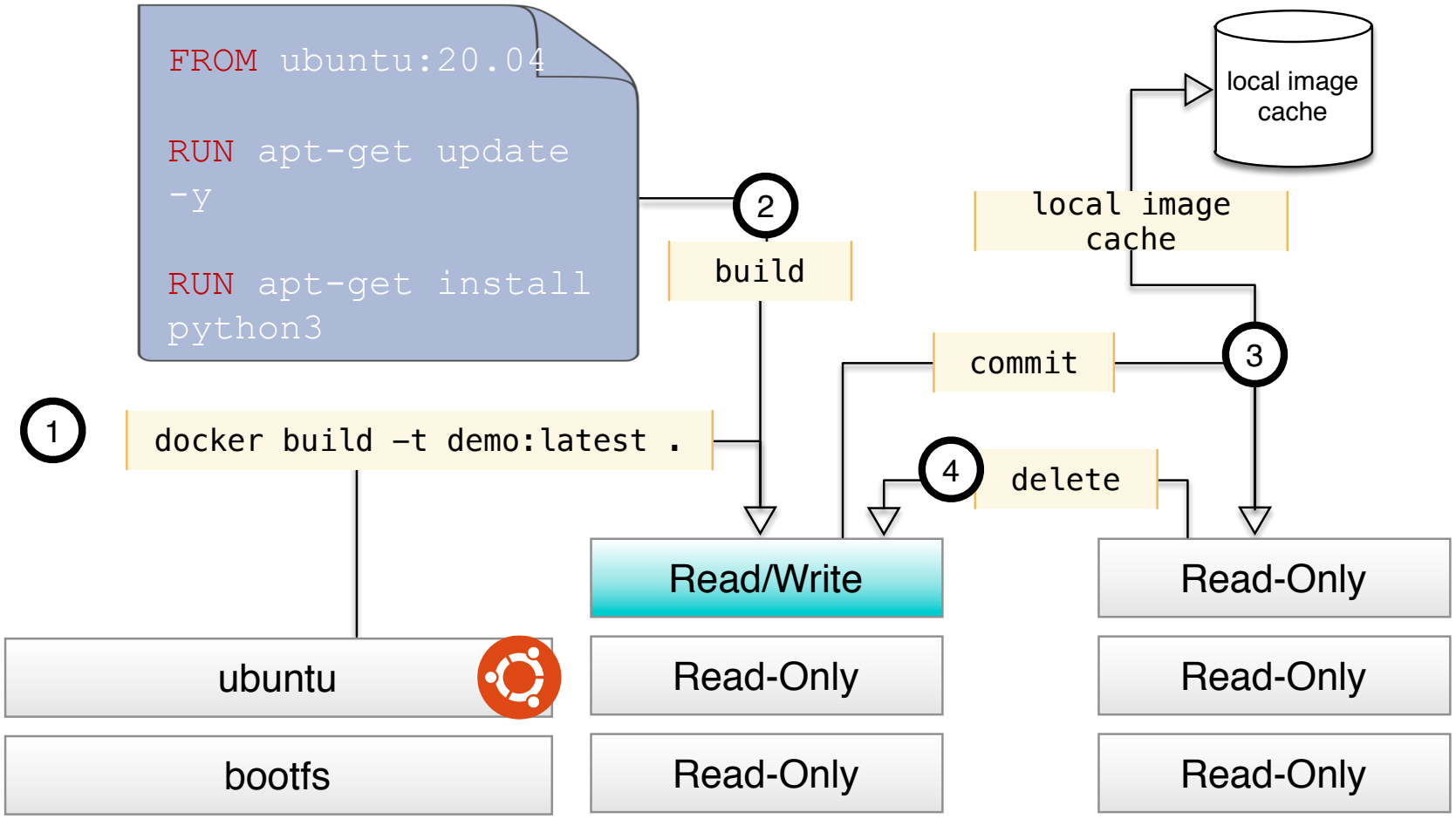
# Build a Docker Image (Dockerfile)

1. The `docker build` command is used to ”bake” an image

- Uses Dockerfile
- Most common flag is `-t` which names the image and (optionally) tags it

| `build` flags | Description |
|---|---|
| -- pull | Pull new version of the image |
| -m, --memory | Memory limit |
| --no-cache | Do not use cache when building the image |
| -q, --quiet | Suppress the verbose output generated by the containers |

# Build a Docker Image (Dockerfile)



# Image Tags

Version tags simplify image identification and tracking

- Common - especially with base operating systems (i.e. ubuntu:20.04)
- Allows iteration on known good versions of images, as well as enable us to know exactly what version of our image we are running
- Use multiple tags on an image for easy handling of version for dev, test, etc.
  - Ex. myimage:dev, myimage:1.1, myimage:latest

# Docker stop & start signals

- ENTRYPOINT/CMD starts as PID1
- SIGTERM sent for graceful shutdown
  - --time=30 (30 seconds until SIGKILL sent)
- SIGKILL sent for immediate shutdown
- docker kill --signal=SIGQUIT nginx
- docker kill --signal=SIGWINCH apache

# Dockerfile kill signals

- "lazy" syntax
  - "python /myapp/server.py"
    - Shell initialized →
      - /bin/bash = PID1 →
        - server.py
- Exec syntax
  - ["python", "/myapp/server.py"]
    - server.py = PID1

- Script used to call app MUST use 'exec python /myapp/server.py'

# Lab: Build Docker Images (Dockerfile)



# Advanced Docker builds

- A great docker image is small e.g. < 20MiB
- The ideal image contains only the application*
- A great image has very few layers*
- The ideal image has 1 layer*
- The ideal build is 100% reproducible
- The ideal build is as fast as possible


*except when it doesn't

# Q: When does performance matter?

ALWAYS

# Q: When does performance not matter?

NEVER

# Typical order of operations

1. Correctness and Integrity
2. Readability and Usability
3. Performance

That said, let’s take performance where it’s free

# Why care?

Smaller Images Mean:

- Faster builds
- Faster pulls
- Faster pushes
- Faster development
- Faster testing
- Faster deployment
- Faster feedback

# Minimal base images

- scratch
- alpine (<5MiB)

## Application Base Images

- redis:4-alpine
- mysql:5.7

## Stack Base Images

- gcr.io/distroless (python, node, java, dotnet, cc)
- python:3.6-alpine
- php:7.2-alpine3.7
- golang

# Layer review

- Images are stacks of layers
- Each Dockerfile line is a layer

```
#Layer n
FROM nixos/nix

#Layer n+1
ENV build=dev

#Layer n+2
CMD /app/bin/start

#Layer n+3
COPY app /app

#Layer n+4
RUN /app/bin/build.sh
```

# Add small layers

Might be small, might be HUGE

```
FROM alpine

COPY . .

```

Probably smaller

```
FROM alpine

COPY app /app
```

# Independent RUNs at top

RUNs are cacheable

```
FROM alpine

RUN dd if=/dev/random of=/tmp/seed bs=1 count=1

# More stuff ...
```

# Dependent RUNs immediately after the file they depend on

```
#Layer n
FROM nixos/nix

#Layer n+1
ENV build=dev

#Layer n+2
CMD /app/bin/start

#Layer n+3
COPY app /app

#Layer n+4
RUN /app/bin/build.sh
```

# Chain your RUNs

Chained commands happen in the same layer

```
FROM node:6-alpine

CMD ["node", "/www/src/index.js"]

COPY package.json /www/

RUN apk add --update git && \
    npm --prefix=/www install && \
    apk del git && \
    rm -rf /var/cache/apk/*

COPY src/ /www/src
```

# Cleanup your RUNs

This build requires git, but the runtime doesn't. Let's cleanup after the build.

```
FROM node:6-alpine

CMD ["node", "/www/src/index.js"]

COPY package.json /www/

RUN apk add --update git && \
    npm --prefix=/www install && \
    apk del git && \
    rm -rf /var/cache/apk/*

COPY src/ /www/src
```

# Add layers sparingly

Each line should have a good reason to exist

Common extraneous layers:
RUN followed by RUN
MAINTAINER <in+version+control@example.com>
EXPOSE 80
WORKDIR

Sometimes extraneous:
VOLUME /app/data

# Order matters

Anti-Example:

```
COPY . /tmp/
RUN pip install --requirement /tmp/requirements.txt

```

Not bad?:

```
COPY requirements.txt /tmp/
RUN pip install --requirement /tmp/requirements.txt
COPY . /tmp/

```

# Order matters

Good:

```
COPY LICENSE /tmp/
COPY requirements.txt /tmp/
RUN pip install --requirement /tmp/requirements.txt
COPY *.py /tmp/
```

Best:

```
COPY LICENSE /tmp/
COPY requirements.txt /tmp/
RUN pip install --requirement /tmp/requirements.txt
COPY constants.py /tmp/
COPY main.py /tmp/
```

# Layers are immutable, nth reminder

Bad*:

```
COPY . .
```

Probably Worse:

```
COPY . .
RUN rm –rf secrets/
```


*potentially *okay* with multi-stage builds

# .dockerignore - reduce build context

you can use .dockerignore in root of build context to exclude files from the build

.dockerignore makes these better:

```
COPY . .
COPY src/ /app/src
```

When simple, explicit is still preferable:

```
COPY foo /app/foo
COPY bar /app/bar
```

# Cacheability vs Layers and Readability

We see a tradeoff between values:
minimizing number of layers and readability
maximizing cacheability

```
COPY *.py /app/
¬
# VS¬
¬
COPY constants.py /app/¬
COPY main.py /app/¬
```

```
COPY *.py /app/¬
¬
# VS¬
¬
COPY a.py /app/¬
COPY b.py /app/¬
COPY c.py /app/¬
COPY d.py /app/¬
# ...
```

# When to **not** minimize an image

When it impacts usability or readability:

- run a debugger inside the container (copy in the debugger dependencies)
- run a profiler inside the container (copy in the profiler dependencies)
- run a supporting application inside the container (e.g. redis-cli)
- open a shell inside a container

# Multi-stage Docker builds

- Introduced in Docker 17.05
- Multiple FROM statements can be used to decompose a docker build into “stages”
- Conditionally build a single stage, or all stages

# Common multi-stage patterns

- Create a “build” layer that performs the actual compilation of your application
- Create a “test” layer that contains canned dummy data for your application to consume or expose for testing
- Create a “production” layer that is stripped down to only a lightweight base image (like alpine) and your binary

# Benefits of multi-stage builds

Portability

- Utilizing a ”build” layer in your Dockerfile allows a docker build to be ported to any build system (Jenkins, TravisCI, CircleCI, etc) that can build a container. No consideration has to be made as to whether the build system "supports" builds with your preferred runtime.

Security

- By only including explicitly what you want in the final image, you decrease your container attack surface

Simplicity

- Removes any requirement to have “test” and “production” Dockerfiles to facilitate testing.
- The number and size of layers in intermediate stages does not factor into the final deployable image.

# Multi-stage build example

Create a named “build” stage

Build stage contains secrets that should not exist in the final image

Create an unnamed runtime stage

Copy artifacts from the “build” stage to the runtime stage

```
FROM node:8-alpine AS build
COPY .npmrc /home/user/.npmrc
COPY package.json /app/
RUN npm –prefix=/app install
COPY src /app/src

FROM node:8-alpine
CMD [“node”, ”/app/src/index.js”]
COPY --from=build /app /app
```

# Trusting Upstream

```
FROM nginx:1.13.9
```

Does project use semver?
If yes are they consistent?

What version of semver?

What happens if upstream breaks my app?
Will tests catch it?

# Reliable builds

```
#nginx:1.13.9
FROM nginx@sha256:4771d09578c7c6a65299e110b3ee1c0a2592f5ea2618d23e4ffe7a4cab1ce5de
COPY index.html /www/

```

+ Reliable base image for builds

- No automatic upstream updates
- Worse readability


okay, maybe not 100%

# Discussion: Optimize Dockerfiles



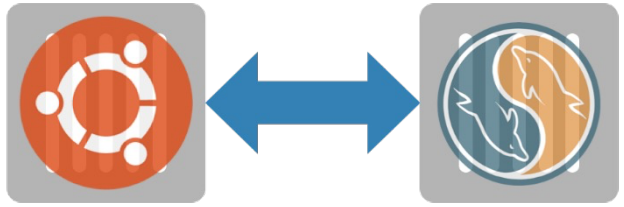
# Docker Networking



# Docker Networking Overview

Docker Engine provides network access for containers that need:

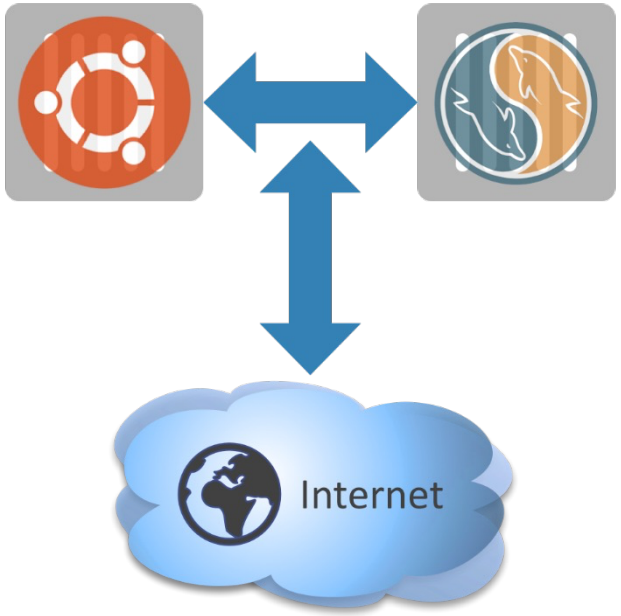
- Other Containers



# Docker Networking Overview

Docker Engine provides network access for containers that need:

- Other Containers
- Networks



# Docker Networking Overview

Docker Engine provides network access for containers that need:

- Other Containers
- Networks

Accomplished through:

1. IP Address Management (IPAM)
2. Service Port exposure
3. Manual Port Mapping
4. Dynamic Port Mapping

# Docker Networking Architecture



# Network Interfaces

**Docker Container**

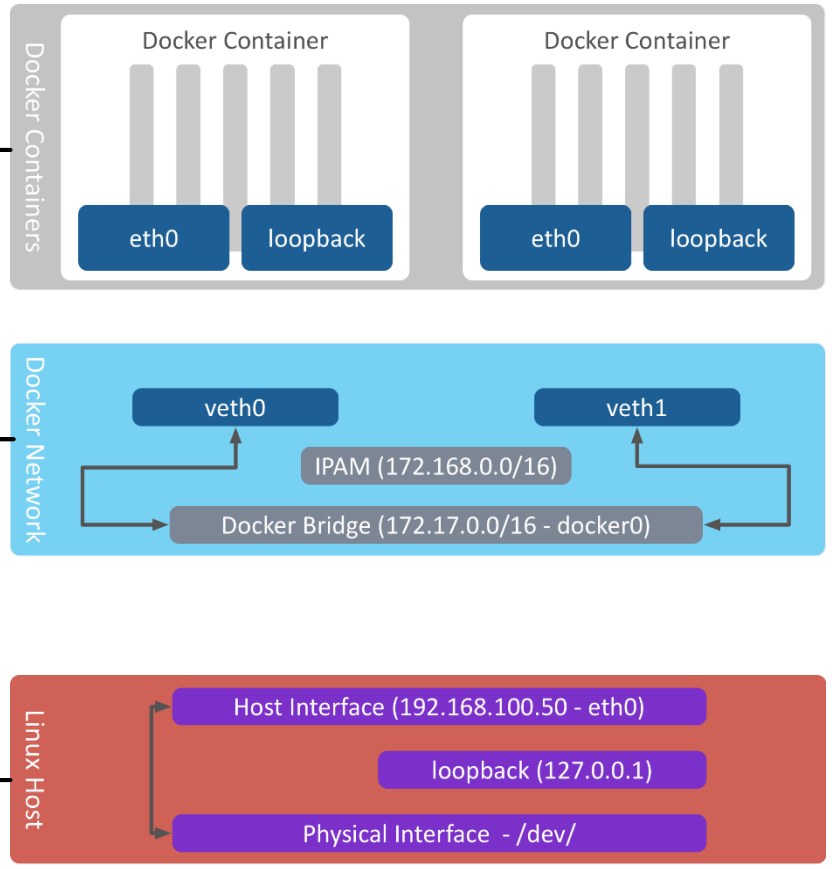
- eth0 Interface
- Loopback Interface

**Docker Network**

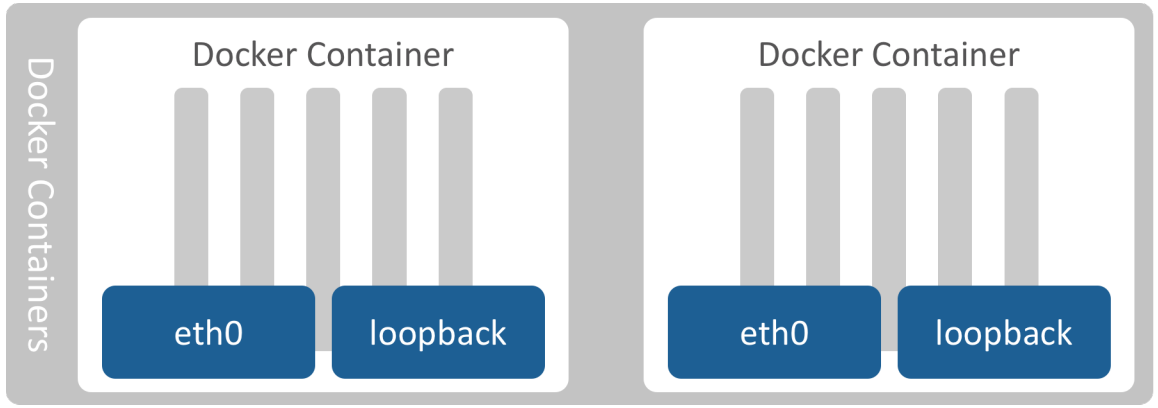
- Docker Bridge
- IP Address Management

**Linux Host Network**

- Host Interfaces
- Host Loopback Interface
- Physical Interfaces



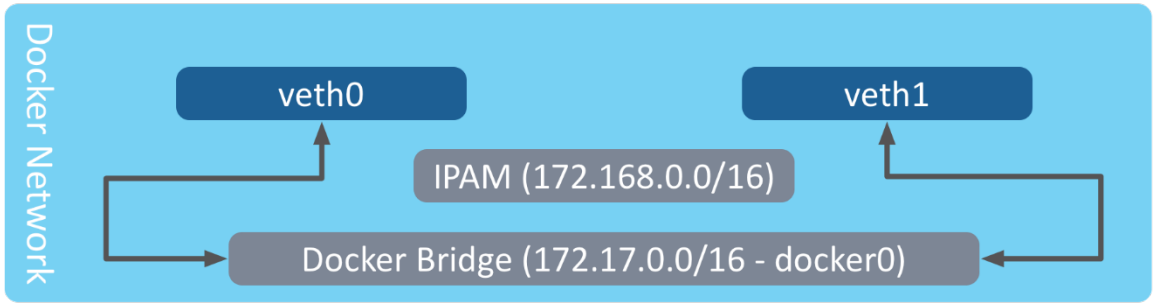
# Docker Container




## Docker Container

- **Loopback Interface** - used for internal communication
- **eth0 Interface** - used for external communication
- Container TCP Port Exposure

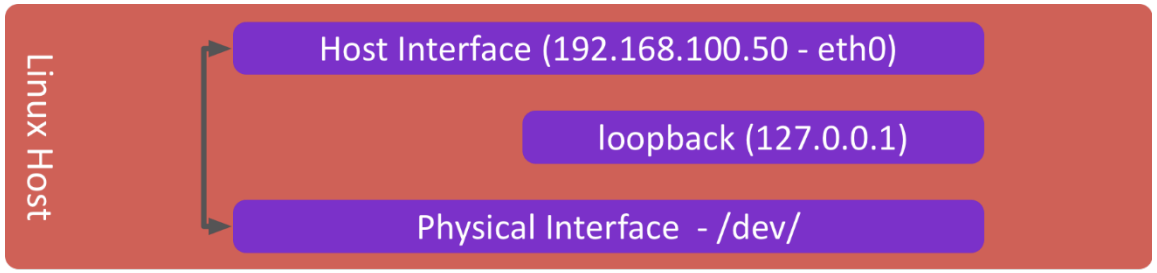
# Docker Container




## Docker Network

- **Docker Bridge** (e.g. docker0) - the bridge which manages traffic between Docker Engine and the Linux Host
- **veth Interfaces** - interfaces on the bridge to the containers
- **IP Address Management** - assigns IP addresses to container that require them
- Container to Host TCP Port Mapping

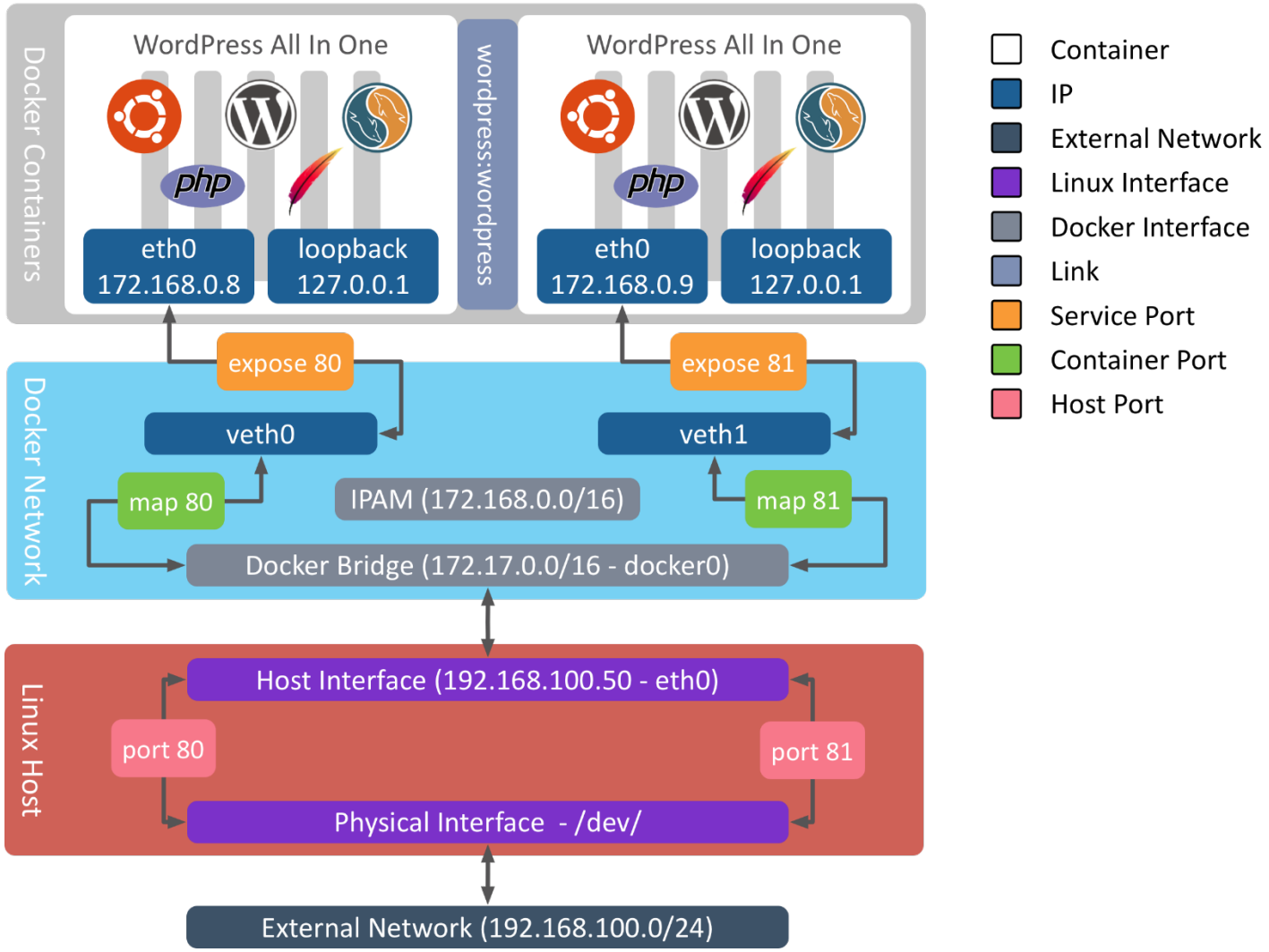
# Docker Container




## Linux Host Network

- **Host Loopback Interface** - host internal communication
- **Host Interfaces** - internal and external communication
- **Physical Interfaces** - sends/receives external communication to/from the outside world
- Linux Host TCP Ports

# Docker Networking Architecture

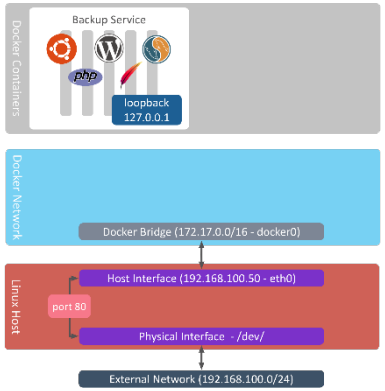


# Container Network Deployment Methods



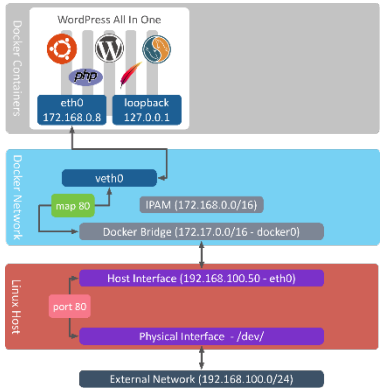
# Container Network Models

## Isolated Container Method



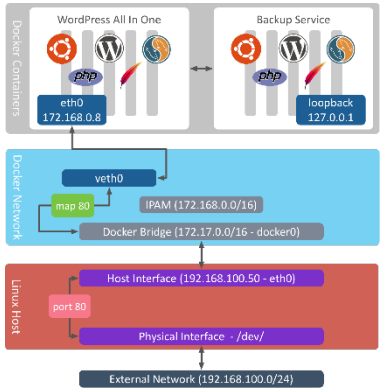

- Seldom used
- Provides the most security
- Has zero access to network resources
- Has no eth0 but has a loopback

## Bridged Container Method



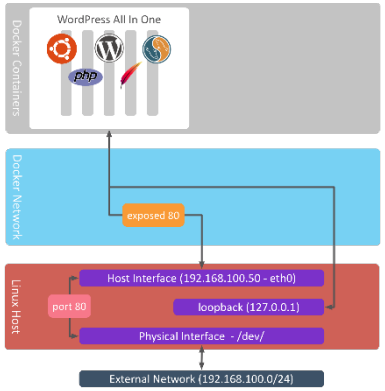

- Most commonly used
- Has limited access to network resources
- Has an eth0 and a loopback

## Shared Container Method



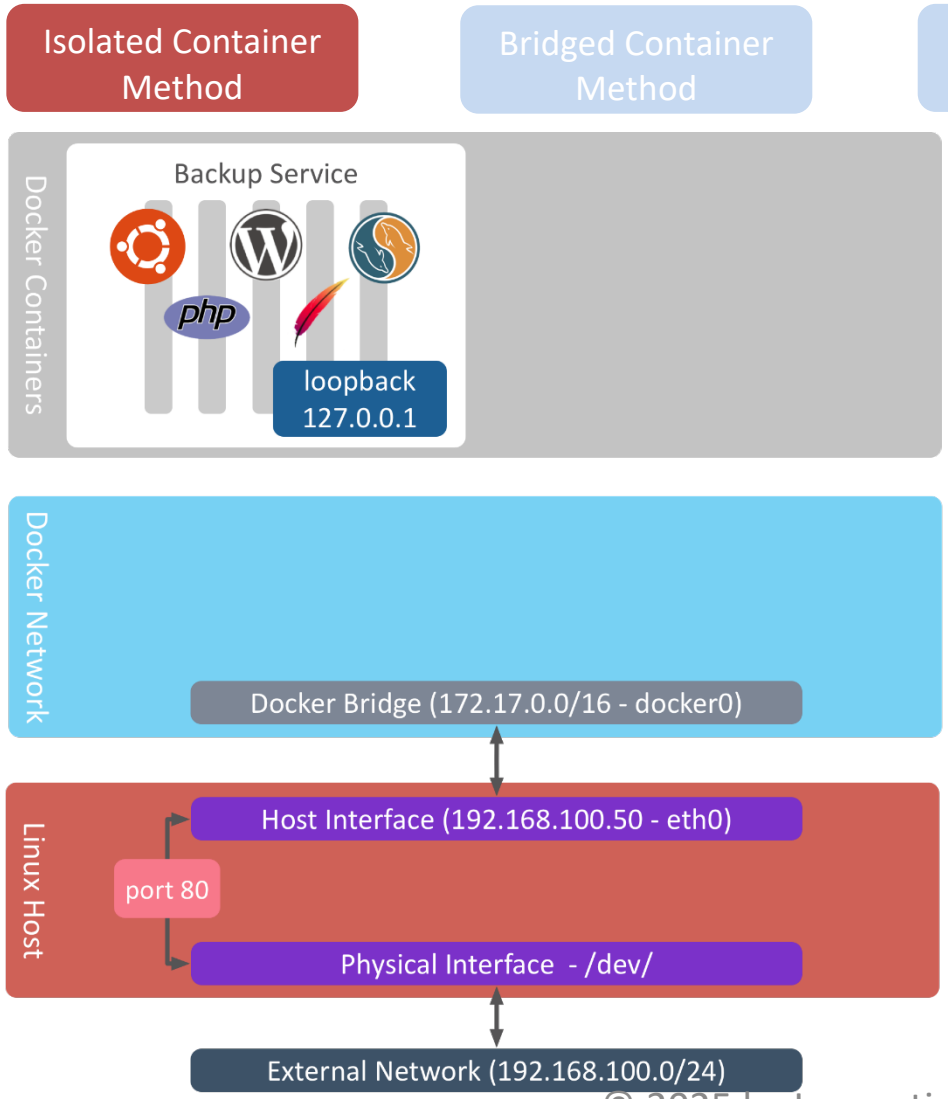

- Least used
- Has limited and no access to network resources
- Each container has either a loopback or eth0

## Open Container Method




- Used as a last resort and with caution
- Unrestricted access to host network
- Uses host network and loopback

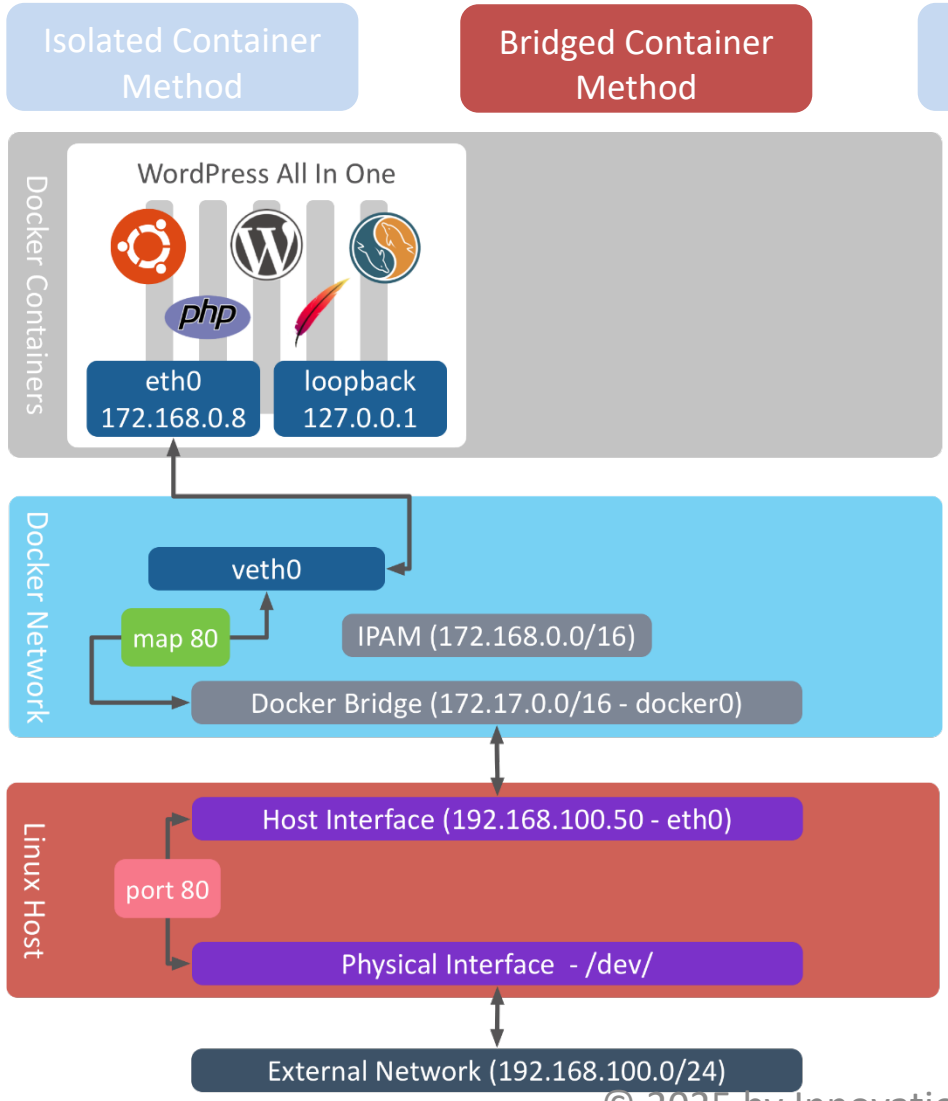
# Isolated Container Method




**Container has:**

- No network access to the outside OR other containers
- A loopback
- Access to storage resources but not NAS/iSCSI
- e.g. Backup, Bigdata analytics, Batch jobs

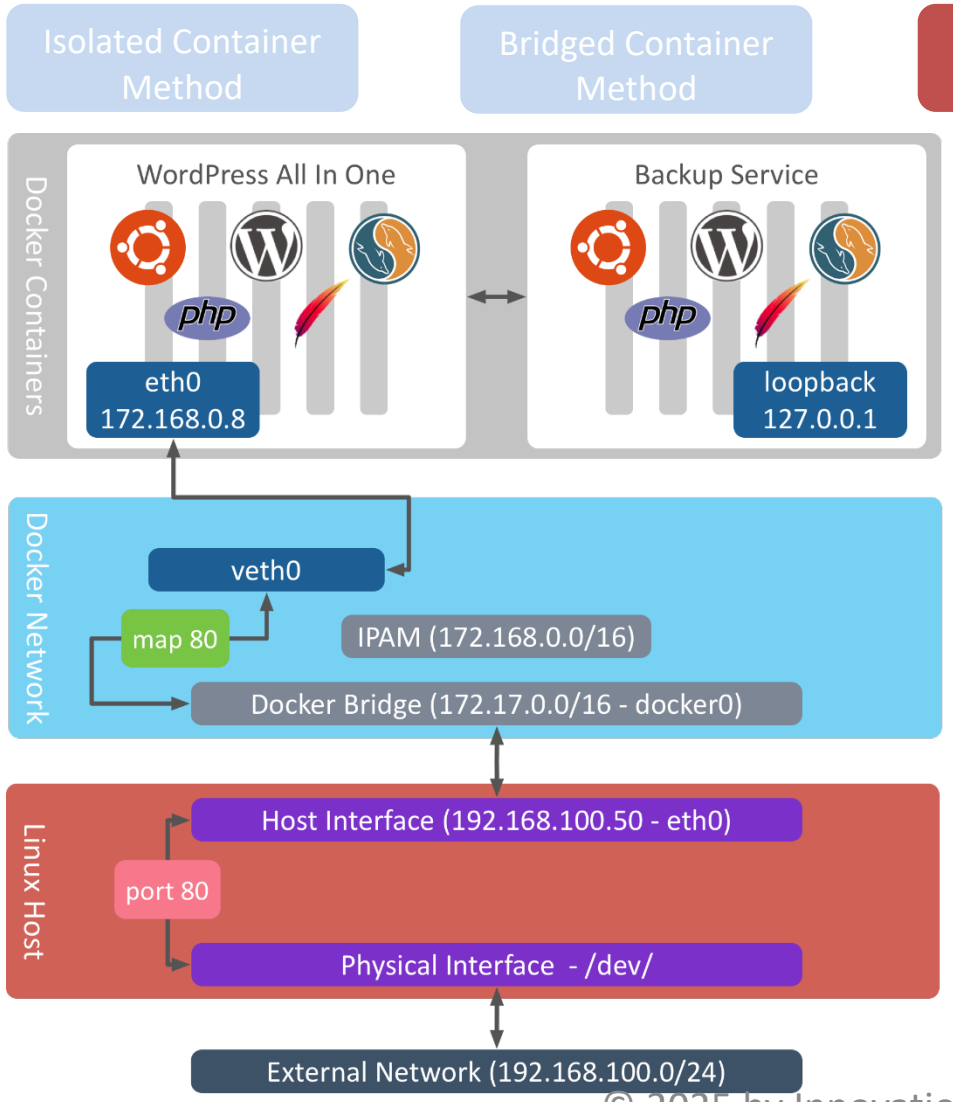
# Bridged Container Method (Default)




**Container has:**

- Network access to the outside and other containers
- IP address is assigned by IPAM services (dhcp default)
- A loopback
- Access to storage resources
- e.g. Most application workloads (web servers, load balancing, database servers)

# Shared Container Method



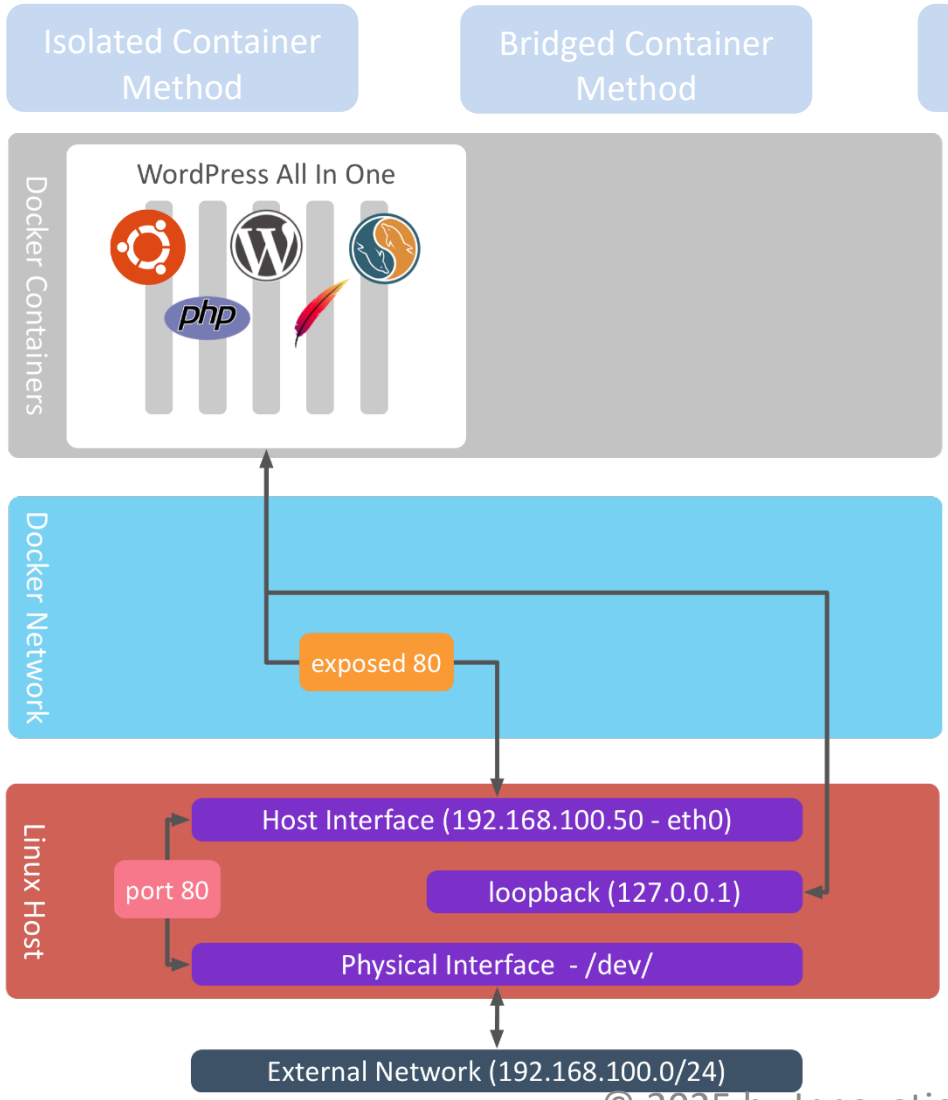

## Container One has:

- Access to bridge network through eth0
- Uses Container B's loopback

## Container Two has:

- A loopback network
- Uses Container A's eth0 to access network
- Example: "Black Box" processes

# Open Container Method




## Container:

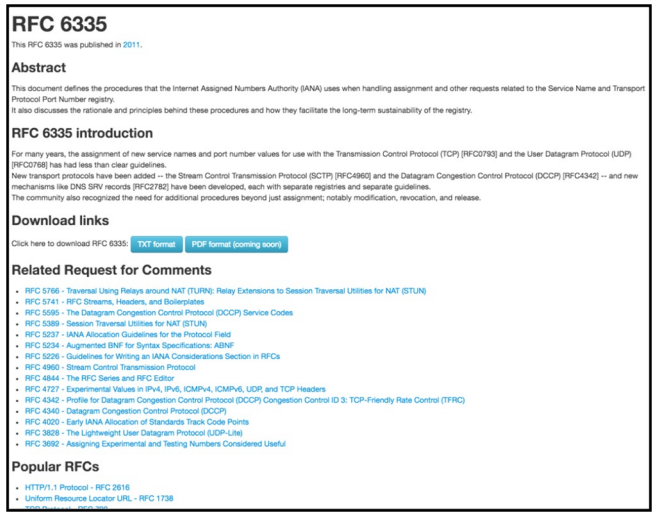
- Has no:
  - Ethernet interface (eth0)
  - Loopback interface (lo)
- Uses
  - Linux Host's network and loopback interfaces
  - Access the outside network through the host's network.
  - Example: Packet Capture

# Service Ports



# Service Ports

Service names and ports are used to distinguish between different services. Many are **defined in RFC6335**



RFC 6335

This RFC 6335 was published in 2011.

Abstract

This document defines the procedures that the Internet Assigned Numbers Authority (IANA) uses when handling assignment and other requests related to the Service Name and Transport Protocol Port Number registry.
It also discusses the rationale and principles behind these procedures and how they facilitate the long-term sustainability of the registry.

RFC 6335 introduction

For many years, the assignment of new service names and port number values for use with the Transmission Control Protocol (TCP) [RFC0793] and the User Datagram Protocol (UDP) [RFC0768] has had less than clear guidelines.
New transport protocols have been added -- the Stream Control Transmission Protocol (SCTP) [RFC4960] and the Datagram Congestion Control Protocol (DCCP) [RFC4342] -- and new mechanisms like DNS SRV records [RFC2782] have been developed, each with separate registries and separate guidelines.
The community also recognized the need for additional procedures beyond just assignment; notably modification, revocation, and release.

Download links

Click here to download RFC 6335: TXT format | PDF format (coming soon)

Related Request for Comments

- RFC 5766 - Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN)
- RFC 5741 - RFC Streams, Headers, and Boilerplates
- RFC 5595 - The Datagram Congestion Control Protocol (DCCP) Service Codes
- RFC 5389 - Session Traversal Utilities for NAT (STUN)
- RFC 5237 - IANA Allocation Guidelines for the Protocol Field
- RFC 5234 - Augmented BNF for Syntax Specifications: ABNF
- RFC 5226 - Guidelines for Writing an IANA Considerations Section in RFCs
- RFC 4960 - Stream Control Transmission Protocol
- RFC 4844 - The RFC Series and RFC Editor
- RFC 4727 - Experimental Values in IPv4, IPv6, ICMPv4, ICMPv6, UDP, and TCP Headers
- RFC 4342 - Profile for Datagram Congestion Control Protocol (DCCP) Congestion Control ID 3: TCP-Friendly Rate Control (TFRC)
- RFC 4340 - Datagram Congestion Control Protocol (DCCP)
- RFC 4020 - Early IANA Allocation of Standards Track Code Points
- RFC 3828 - The Lightweight User Datagram Protocol (UDP-Lite)
- RFC 3692 - Assigning Experimental and Testing Numbers Considered Useful

Popular RFCs

- HTTP/1.1 Protocol - RFC 2616
- Uniform Resource Locator URL - RFC 1738

- Some Common Service Ports:
  - FTP TCP Port 21
  - SSH TCP Port 22
  - HTTP Web Server TCP Port 80
  - HTTP SSL Web Server TCP Port 443
  - LDAP TCP Port 389

# Service Ports and Applications

## Common TCP Service and Application Ports:

| Service | Service Port |
|---|---|
| FTP | 21 |
| SSH | 22 |
| HTTP | 80 |
| Apache | 80 |
| LDAP | 389 |
| HTTPS | 443 |
| MariaDB | 3306 |
| MySQL | 3307 |
| redis | 6379 |
| tomcat | 8080 |

# Expose Service Ports



# Expose Service Ports

Docker Engine provides a mechanism to expose required service ports

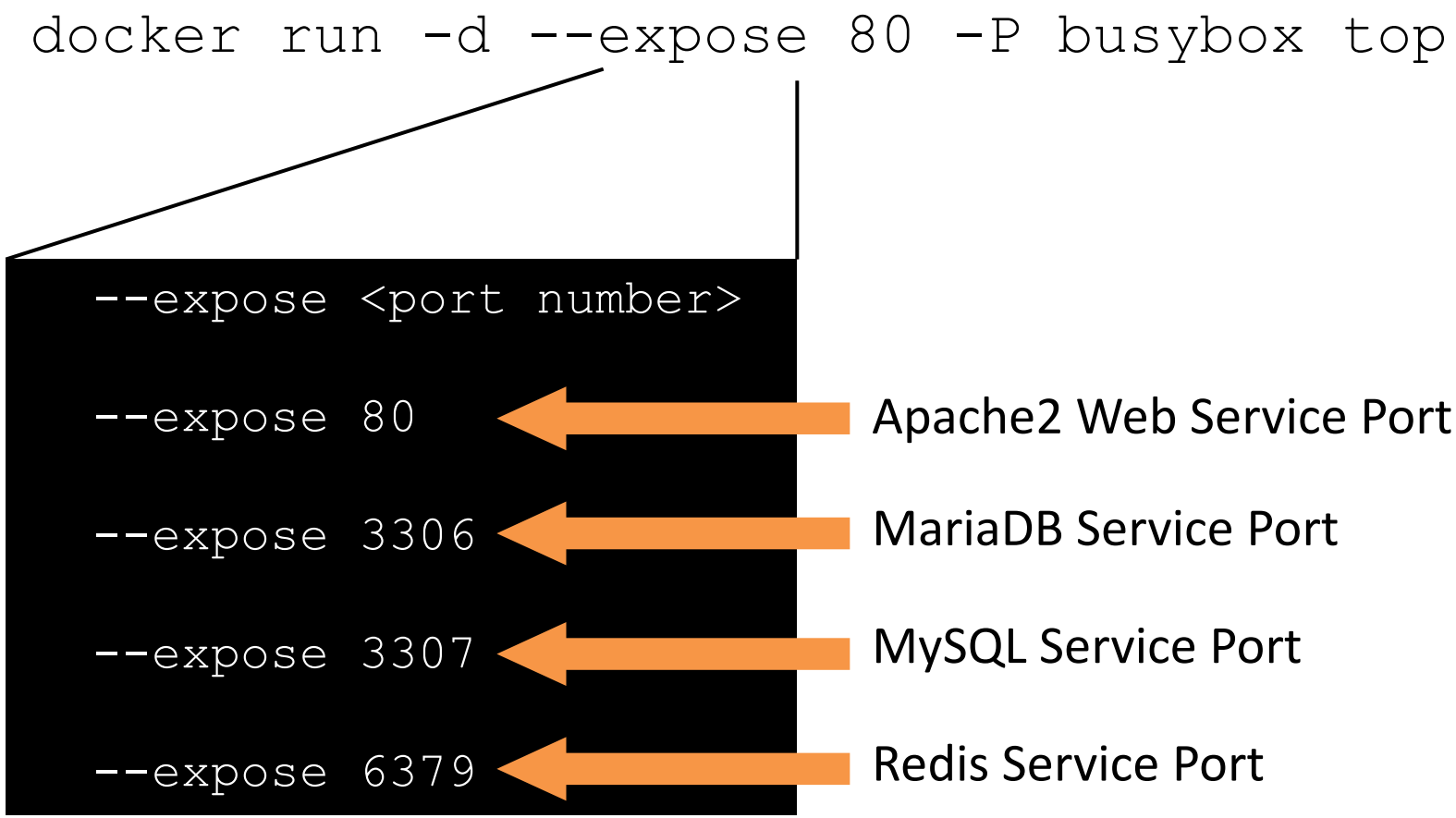
- Exposure is by port, not service name
- Service port exposure is required for communication with the container
- Service ports are exposed:
  - Docker Container creation
  - Docker Image creation

# Expose Service Ports

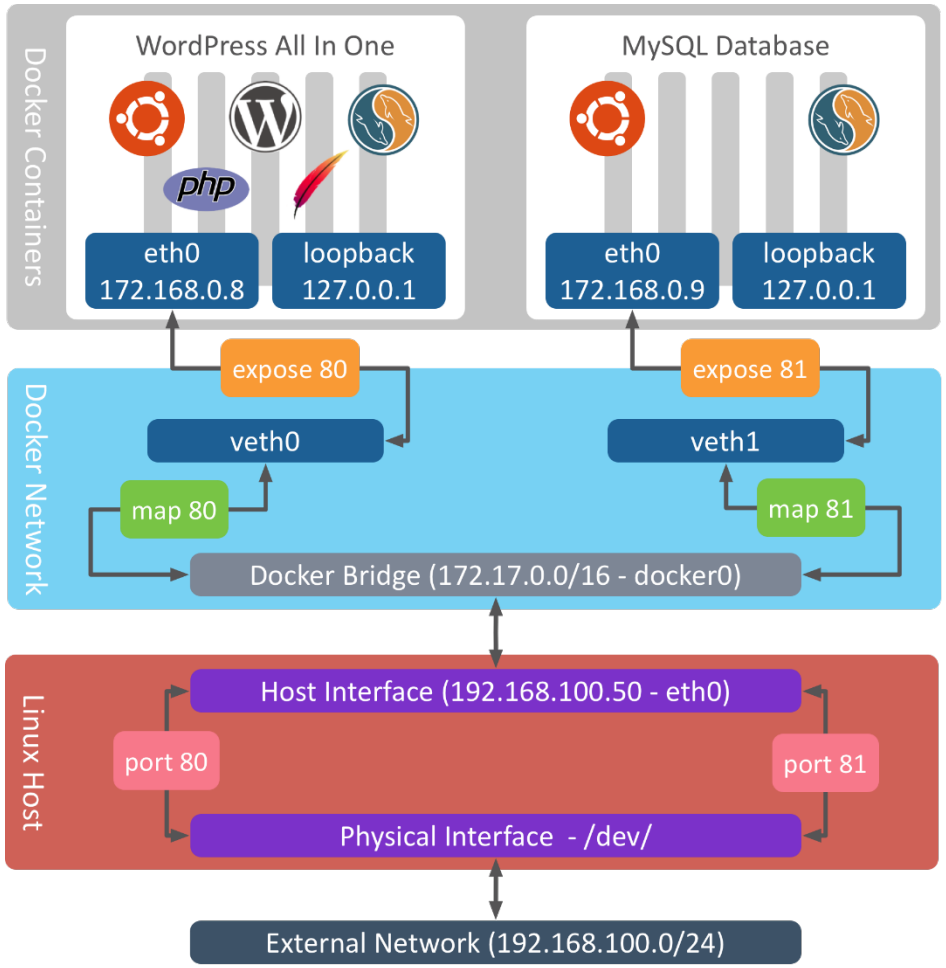
## Common TCP Service Ports:

| Service | Service Port | Exposed Port |
|---|---|---|
| FTP | 21 | 21 |
| SSH | 22 | 22 |
| HTTP | 80 | 80 |
| Apache | 80 | 80 |
| LDAP | 389 | 389 |
| HTTPS | 443 | 443 |
| MariaDB | 3306 | 3306 |
| MySQL | 3307 | 3307 |
| redis | 6379 | 6379 |
| tomcat | 8080 | 8080 |

# Map Ports on a Bridged Container



# Expose Container Service Ports




**Container:**

- Service ports are defined
- Service ports are exposed from the container
- Ports are not accessible outside Docker network

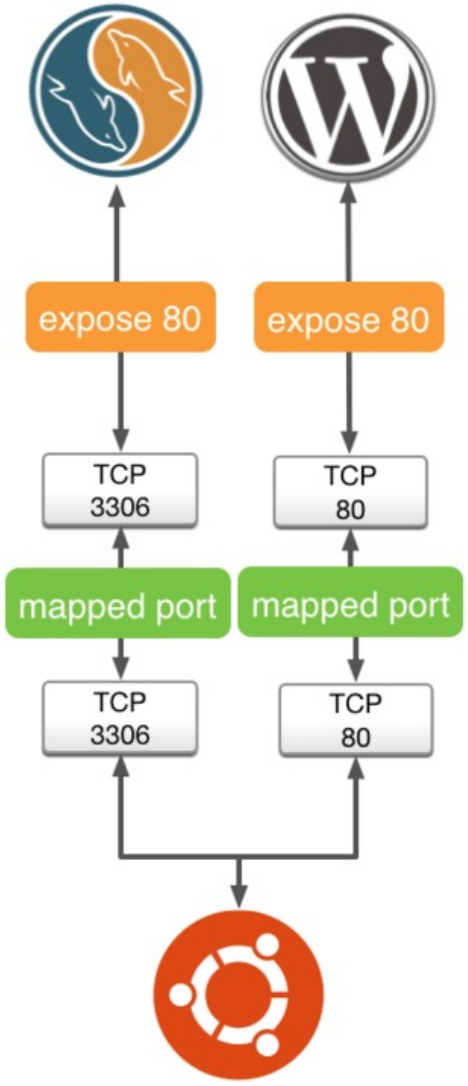
# Port Mapping Containers



# Map Exposed Service Ports

Docker provides a mechanism to map exposed service ports

- TCP ports can be mapped:
  - Dynamically
  - On all host interfaces
  - On user defined ports and interfaces
- Exposed service ports are only mapped during container creation:
  - Docker Container creation



# Expose Service Ports

## Common Service Ports:

| Service | Service Port | Expose port | Mapped Port | Host Port |
|---|---|---|---|---|
| Apache2 | 80 | 80 | 80 | 33731 |
| HTTPD | 80 | 80 | 80 | 33732 |
| LDAP | 389 | 389 | 389 | 33733 |
| MariaDB | 3306 | 3306 | 3306 | 33734 |
| MySQL | 3307 | 3307 | 3307 | 33735 |
| redis | 6379 | 6379 | 6379 | 33736 |

Docker provides a mechanism to:

- map exposed service ports → mapped service ports → host TCP ports

- Note the two service ports using TCP Port 80

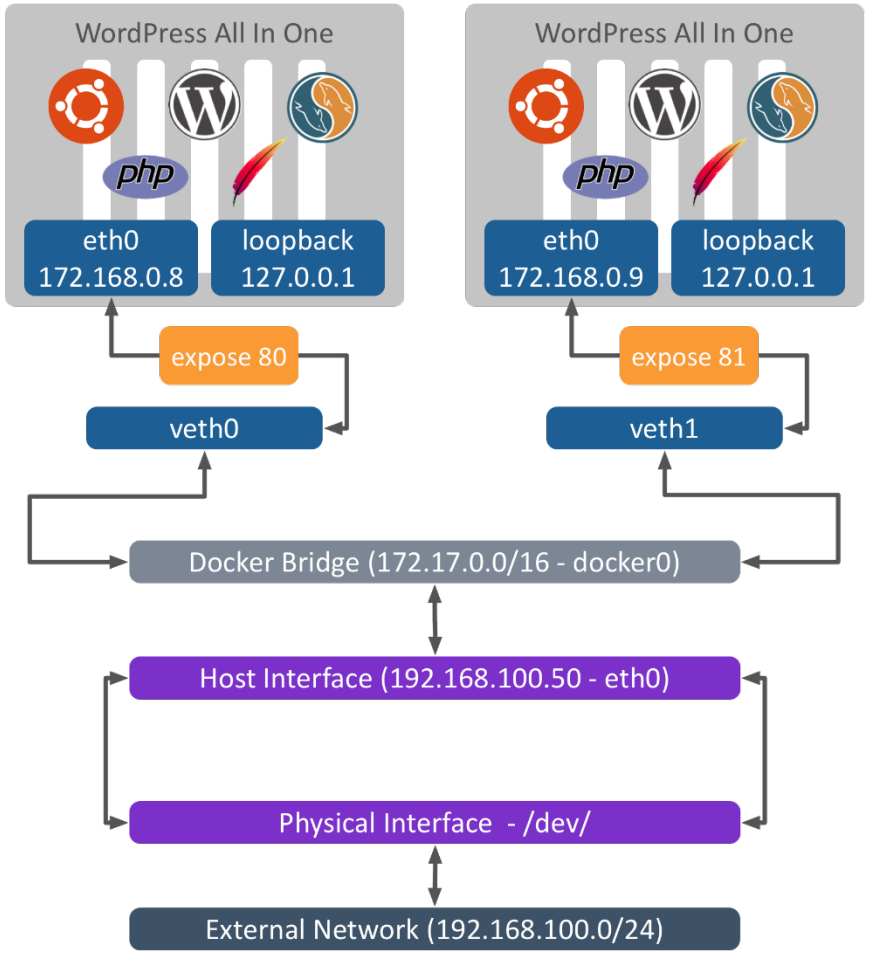
# WordPress Service Ports

## WordPress AIO Service Ports:

| Service | Service Port | Exposed Port | Mapped Port | Host Port |
|---|---|---|---|---|
| WPAIO01 | 80 | 80 | 80 | 33731 |
| WPAIO02 | 80 | 80 | 80 | 33732 |
| WPAIO03 | 80 | 80 | 80 | 33733 |
| WPAIO04 | 80 | 80 | 80 | 33734 |
| WPAIO05 | 80 | 80 | 80 | 33735 |
| WPAIO06 | 80 | 80 | 80 | 33736 |

The above table depicts how a single Linux Host can host up six WordPress AIO(s) on container port 80, using dynamically assigned ports on the Linux Host

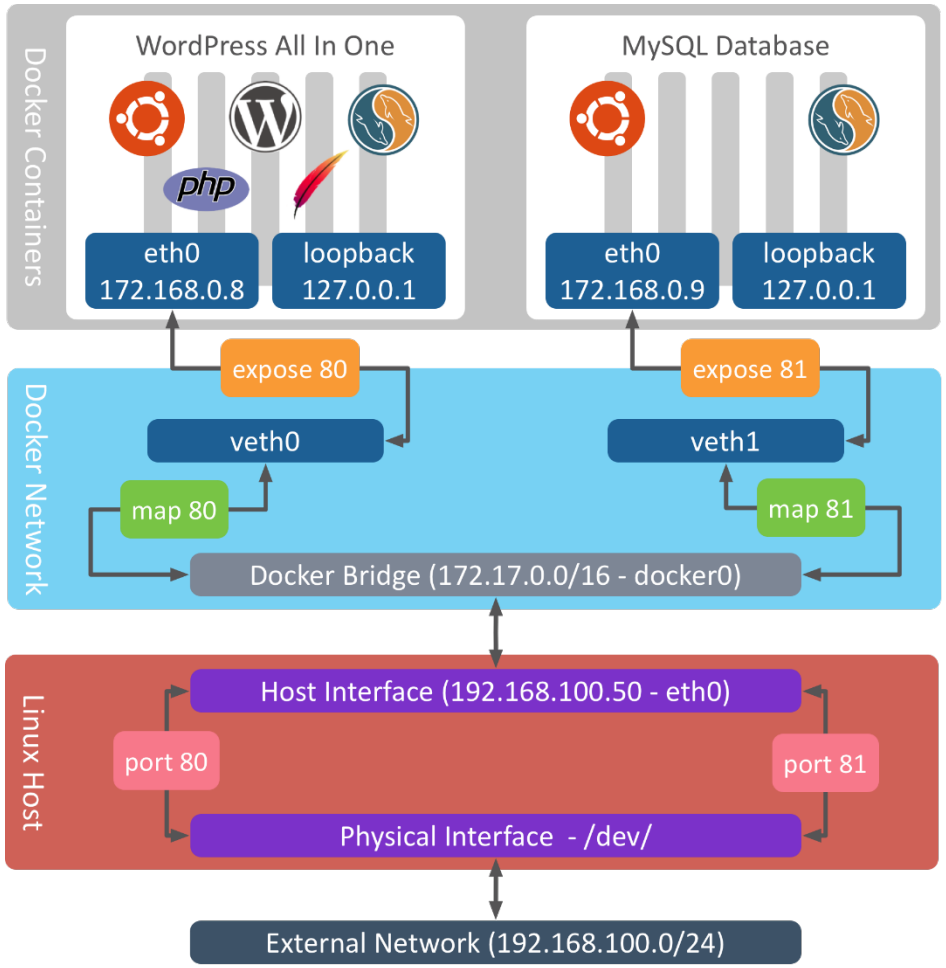
# Expose Container Service Ports




**Container:**

- Service ports are defined
- Service ports are exposed from the container
- Ports are not accessible

# Map Container Service Ports

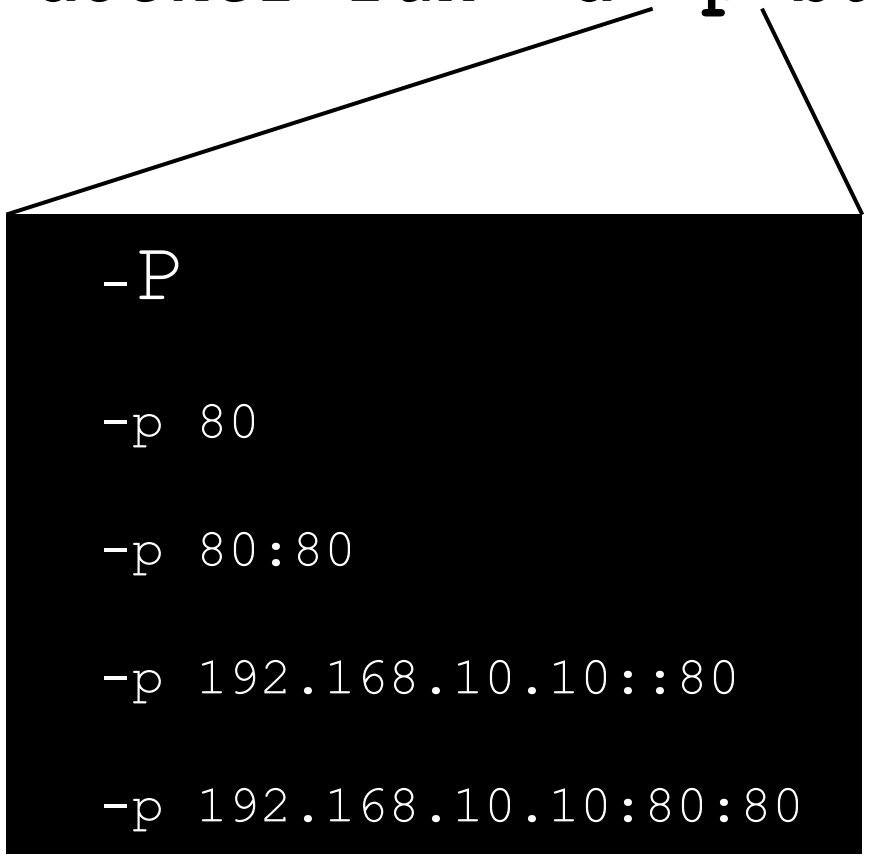



**Container:**

- Service Ports are known
- Ports are exposed from the container
- Host ports are mapped to container ports for accessibility

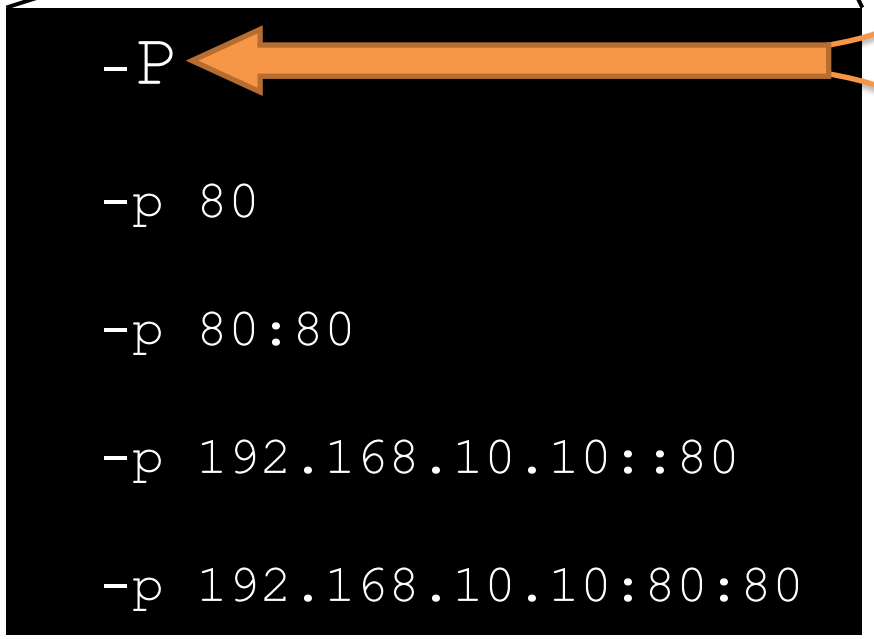
# Map Ports on a Bridged Container

```
docker run -d -p busybox:latest top
```



# Map Ports on a Bridged Container

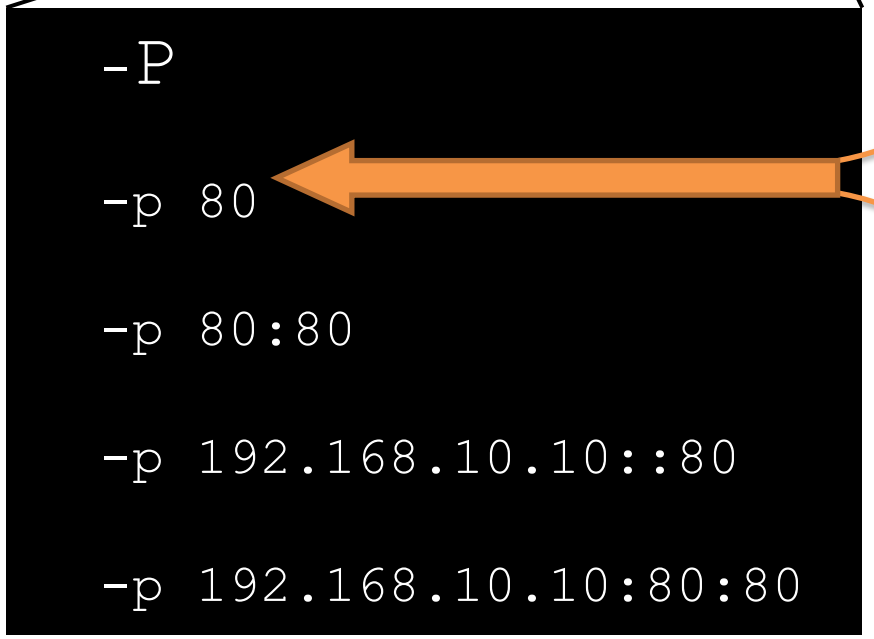
```
docker run -d -p busybox:latest top
```




- Map exposed ports to all host interfaces
- Random host ports
- Ports must be exposed
- Capital **P** not p

# Map Ports on a Bridged Container

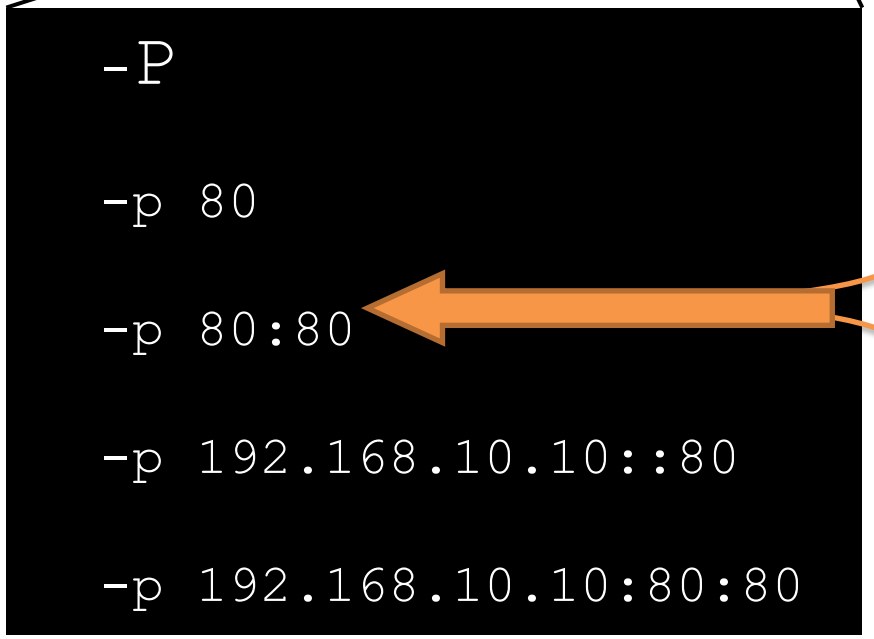
```
docker run -d -p busybox:latest top
```




- Maps container port 80 to random host port on all host interfaces

# Map Ports on a Bridged Container

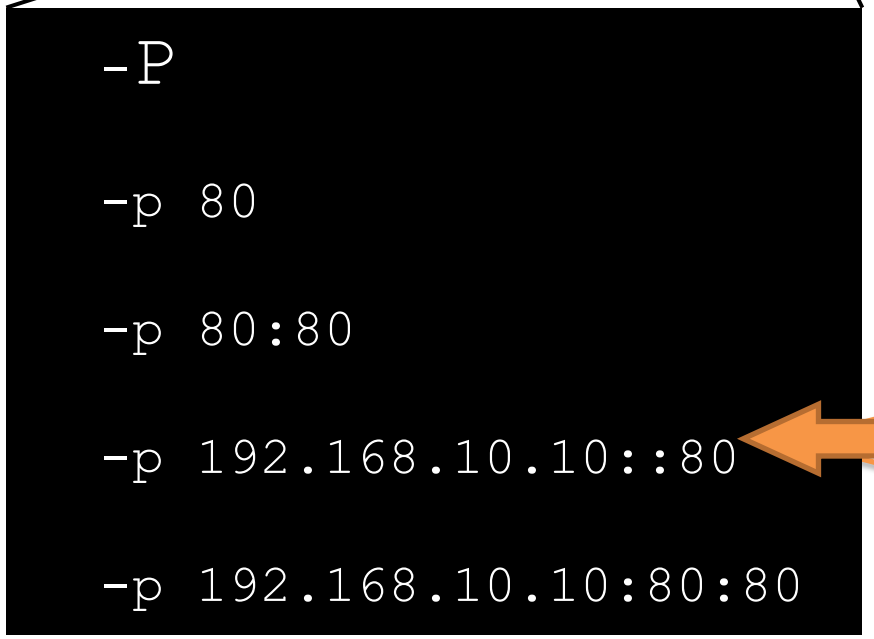
```
docker run -d -p busybox:latest top
```




- Maps container port 80 to host port 80 on all host interfaces

# Map Ports on a Bridged Container

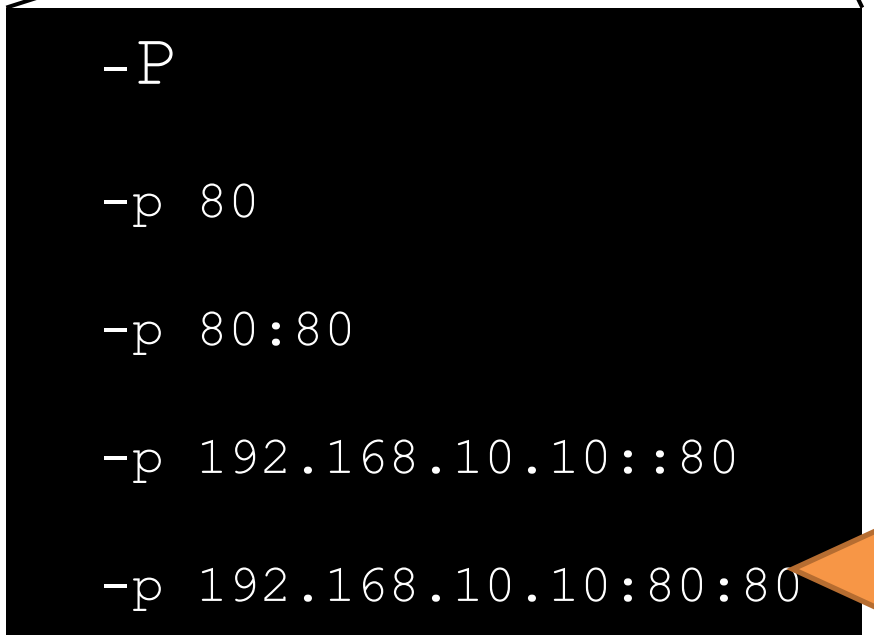
```
docker run -d -p busybox:latest top
```




- Maps container port 80 to random host port on specified host interface

# Map Ports on a Bridged Container

```
docker run -d -p busybox:latest top
```

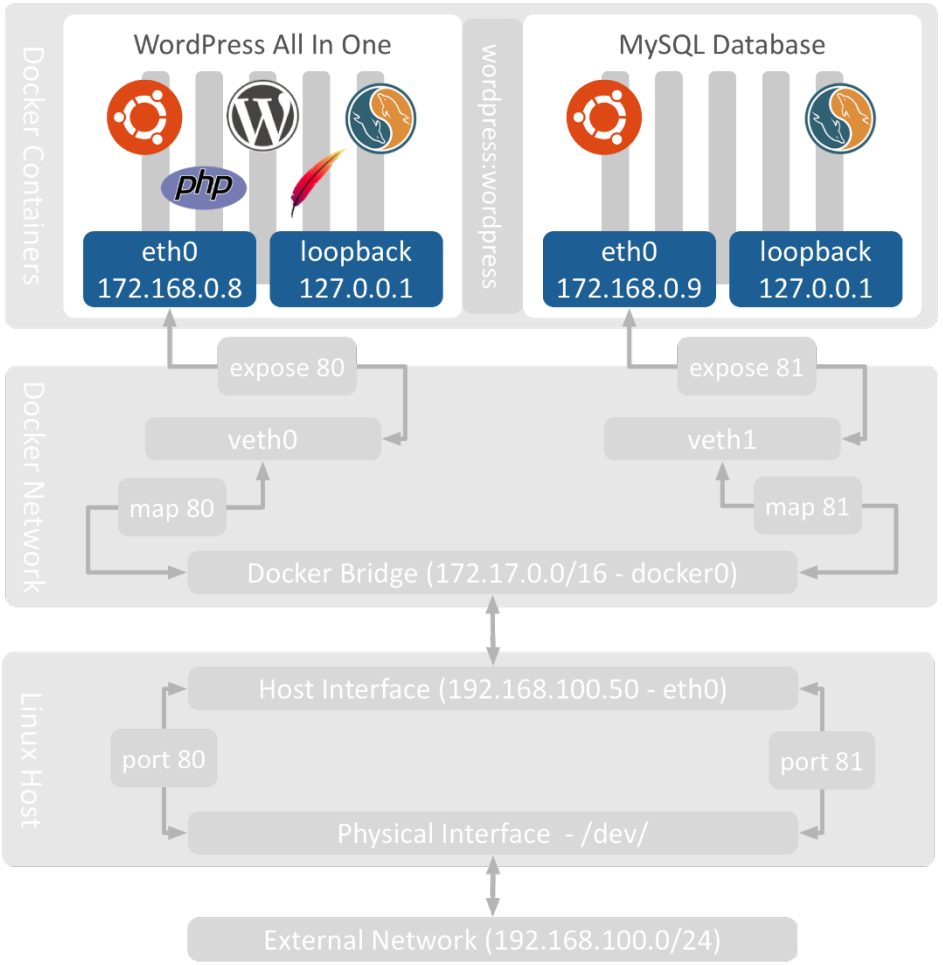



- Maps container port 80 to host port 80 on host specified host interface

# Link Containers



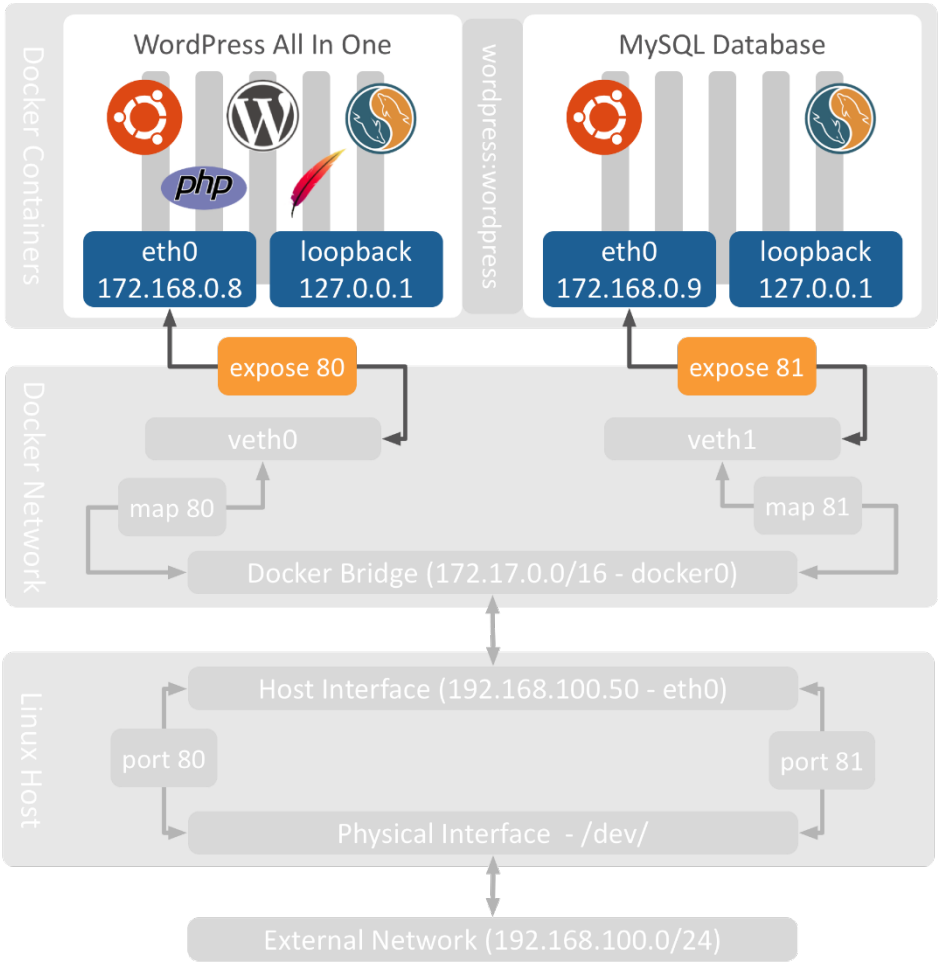
# Link containers (LEGACY)




**Linking:**

- Service Ports are known to all containers
- Ports are exposed from the container
- Host ports are mapped to container ports for accessibility
- Linking containers create internal routing rules
- Linking containers uses aliases to connect containers

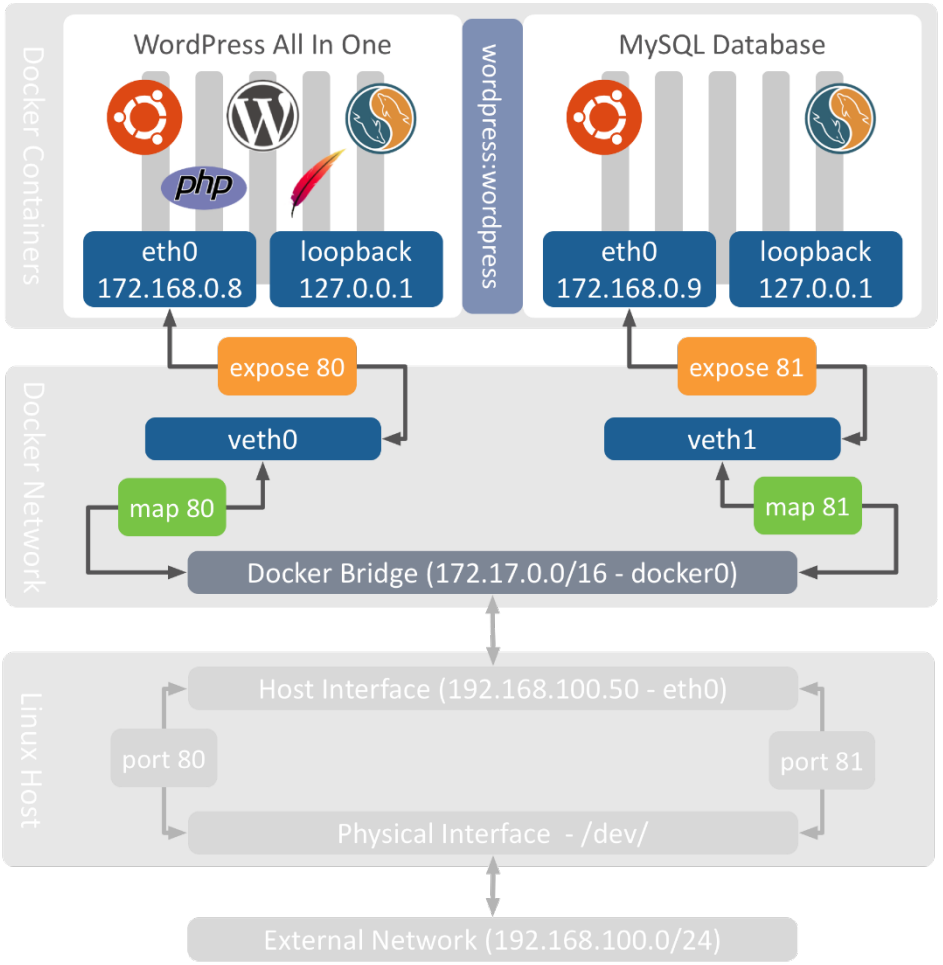
# Link containers (LEGACY)




**Linking:**

- Service Ports are known to all containers
- Ports are exposed from the container
- Host ports are mapped to container ports for accessibility
- Linking containers create internal routing rules
- Linking containers uses aliases to connect containers

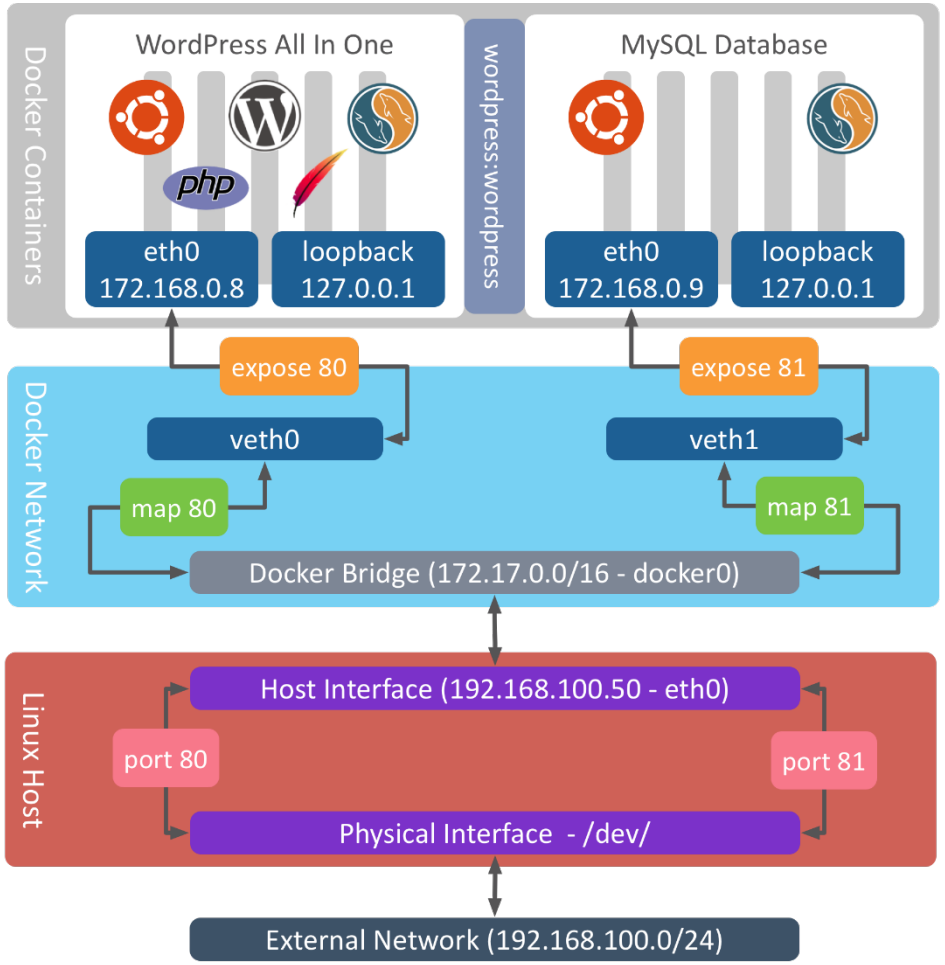
# Link containers (LEGACY)




**Linking:**

- Service Ports are known to all containers
- Ports are exposed from the container
- Host ports are mapped to container ports for accessibility
- Linking containers create internal routing rules
- Linking containers uses aliases to connect containers

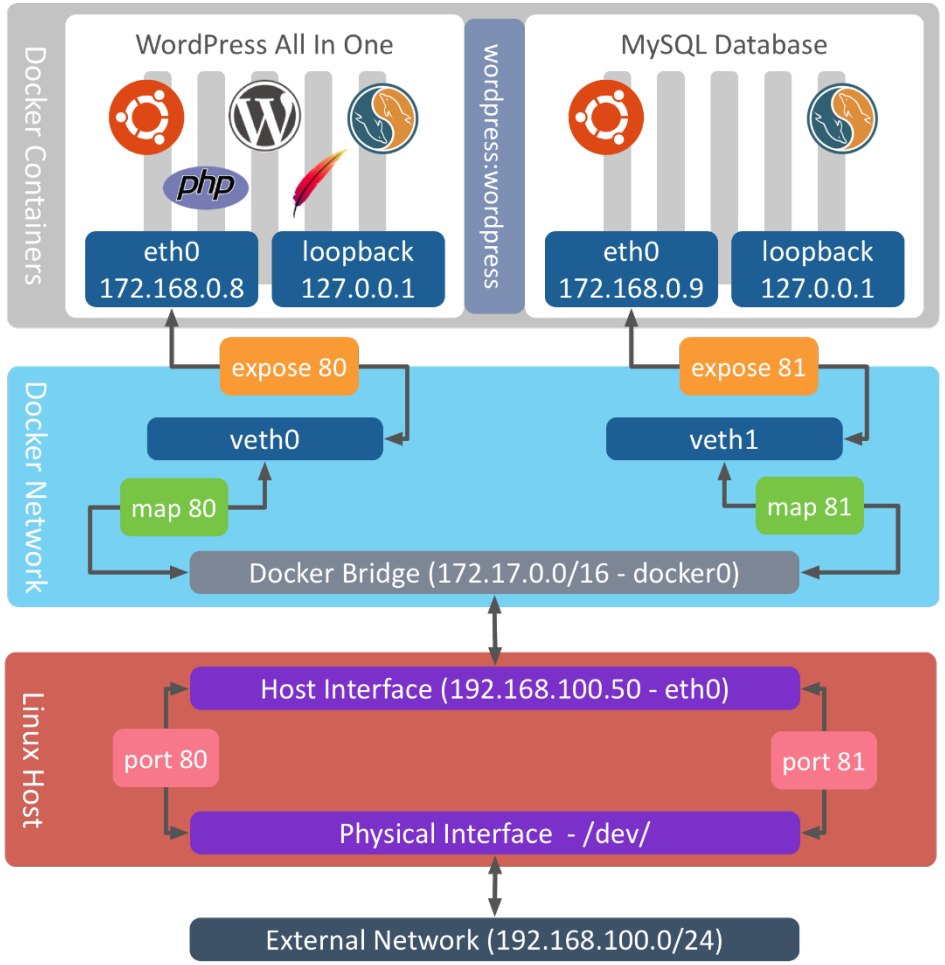
# Link containers (LEGACY)




**Linking:**

- Service Ports are known to all containers
- Ports are exposed from the container
- Host ports are mapped to container ports for accessibility
- Linking containers create internal routing rules
- Linking containers uses aliases to connect containers

# Link containers (LEGACY)




**Linking:**

- Service Ports are known to all containers
- Ports are exposed from the container
- Host ports are mapped to container ports for accessibility
- Linking containers create internal routing rules
- Linking containers uses aliases to connect containers

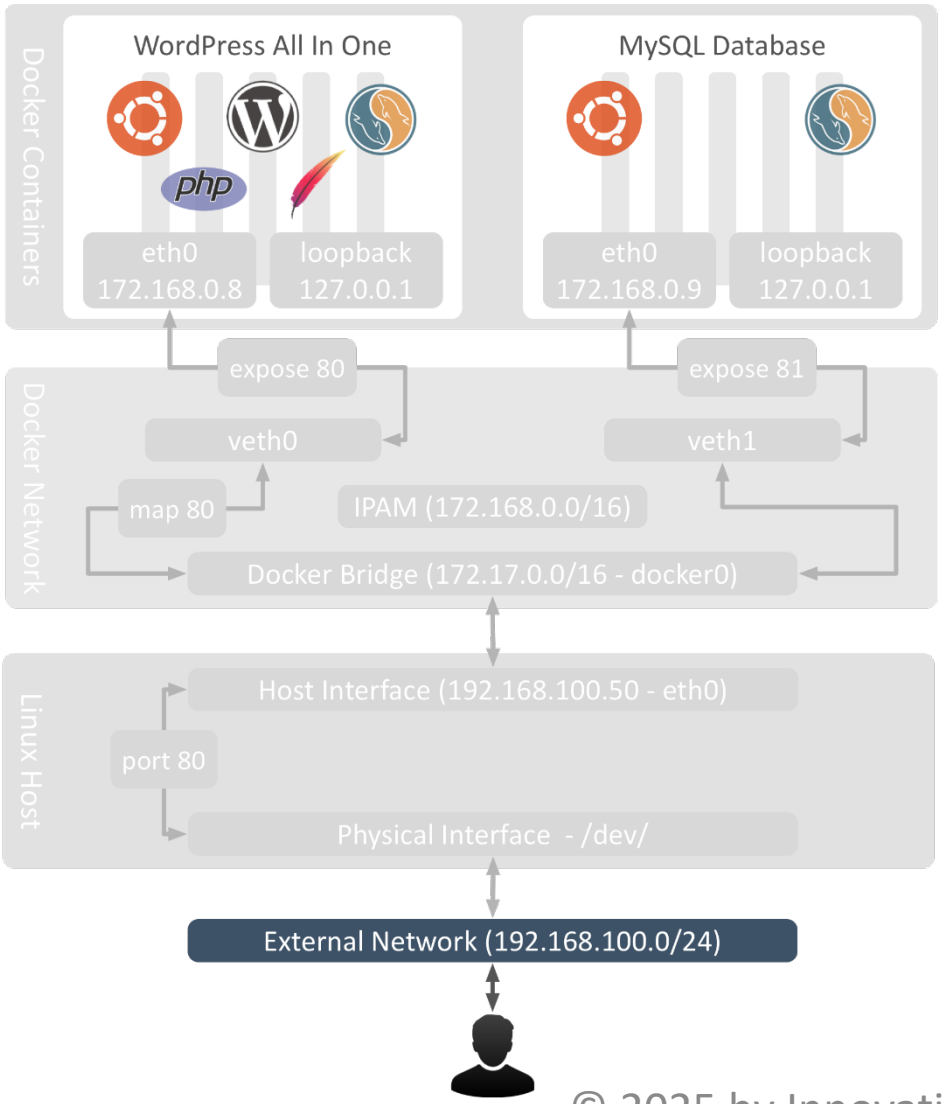
# User Traffic Flow



# User Traffic Flow

- **Applications provide:**
  - Service Ports that need to be exposed
- **Docker Engine provides:**
  - Exposure of required Service Ports
  - Mapping of required Service Ports
- **Linux Host provides:**
  - Internal connectivity between containers
  - Internal connectivity between host and containers
  - External connectivity between containers and users

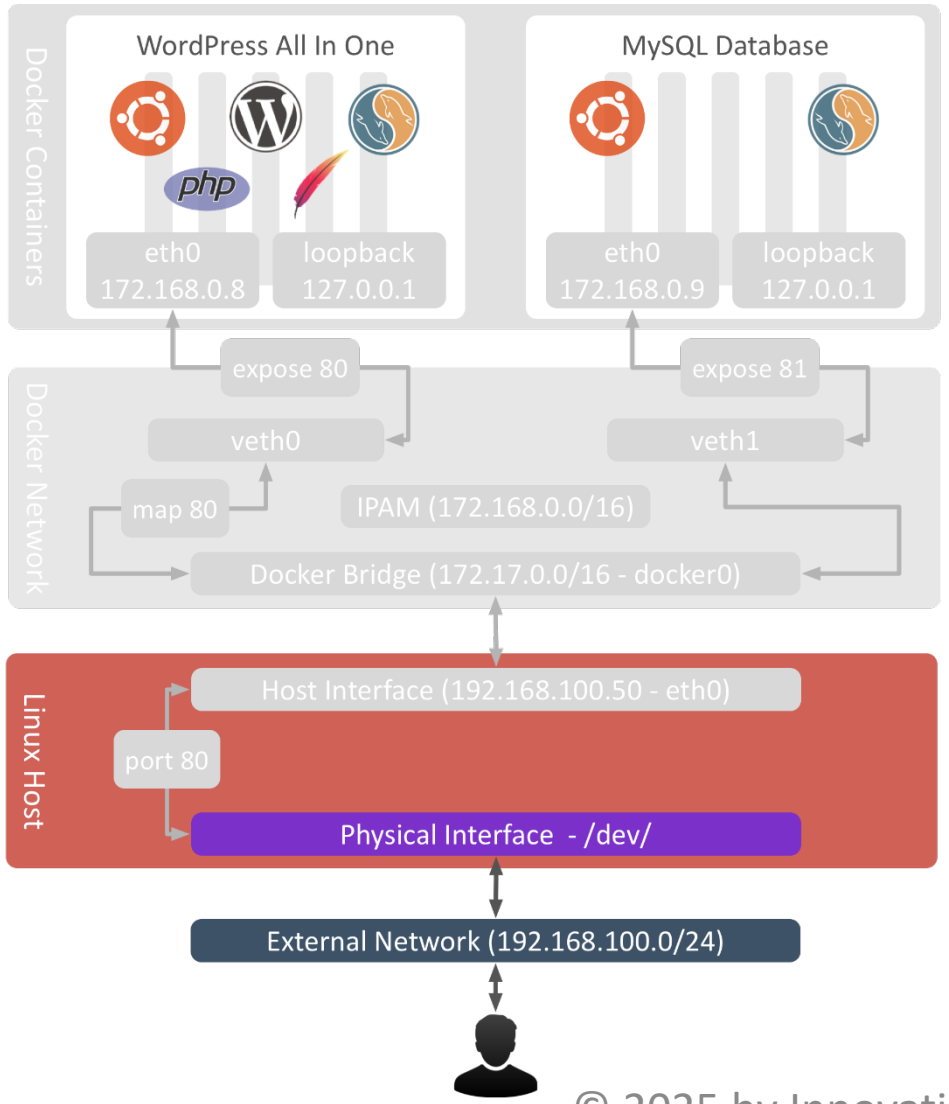
# User Traffic Flow




**User Traffic:**

- Enters from the external network (outside the Docker host)
- Enters the physical interface
- Is passed to the host interface (eth0)
- Is passed to the Docker Bridge
- Is passed to the veth interface
- Is passed to eth0 in the container

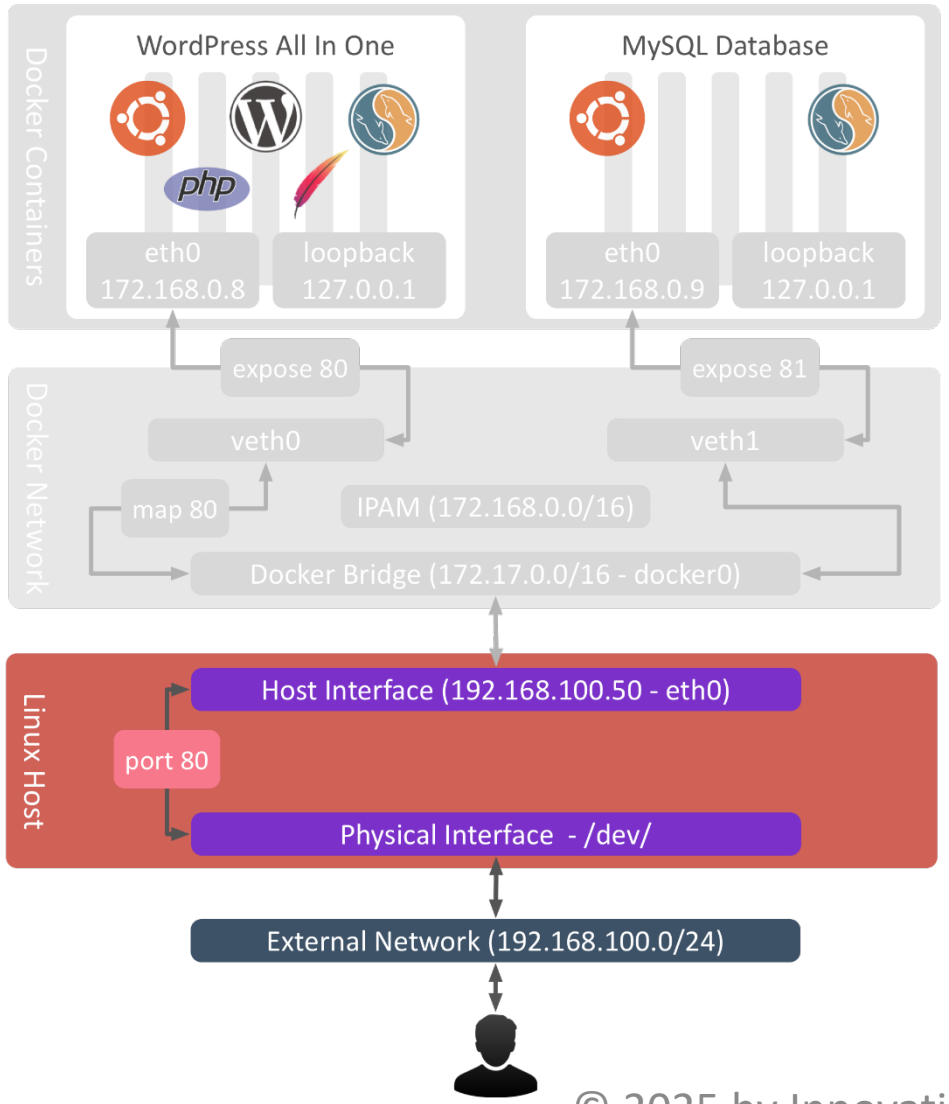
# User Traffic Flow




**User Traffic:**

- Enters from the external network (outside the Docker host)
- Enters the physical interface
- Is passed to the host interface (eth0)
- Is passed to the Docker Bridge
- Is passed to the veth interface
- Is passed to eth0 in the container

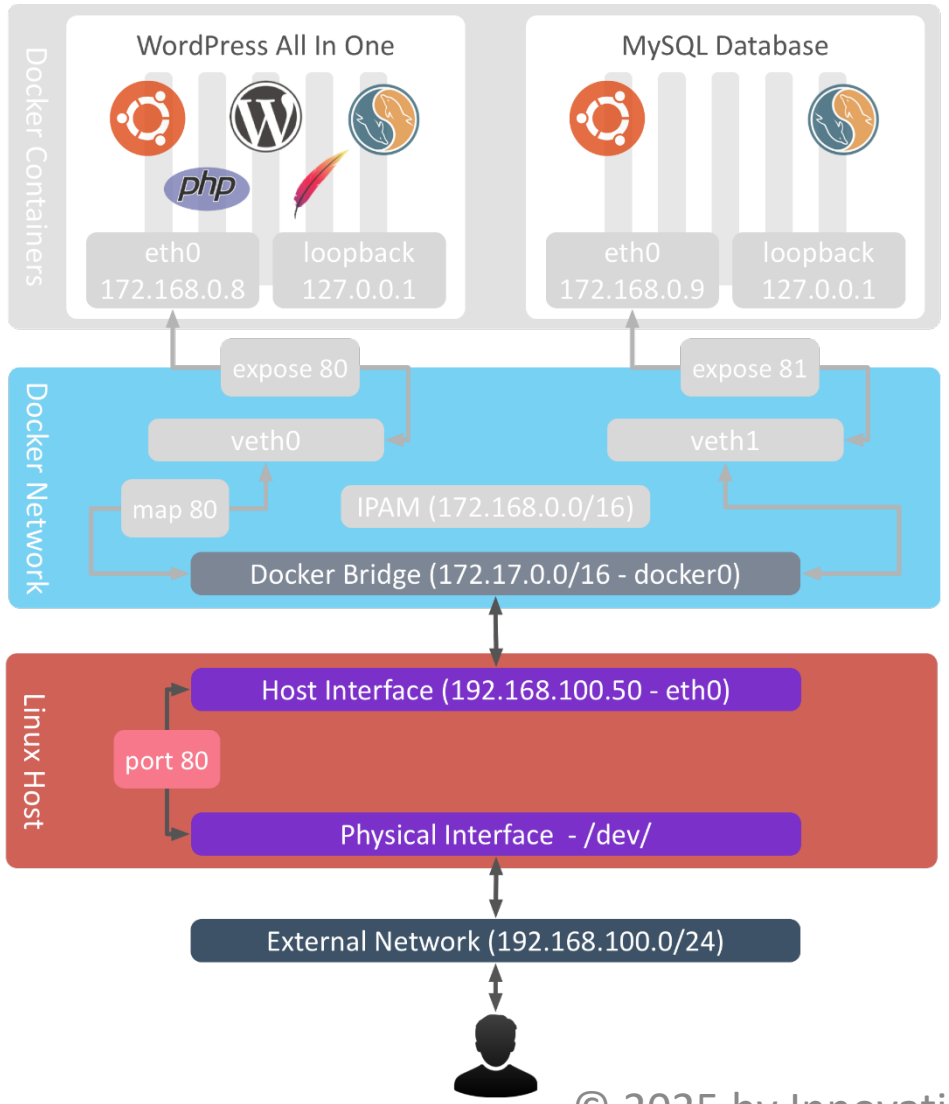
# User Traffic Flow




**User Traffic:**

- Enters from the external network (outside the Docker host)
- Enters the physical interface
- Is passed to the host interface (eth0)
- Is passed to the Docker Bridge
- Is passed to the veth interface
- Is passed to eth0 in the container

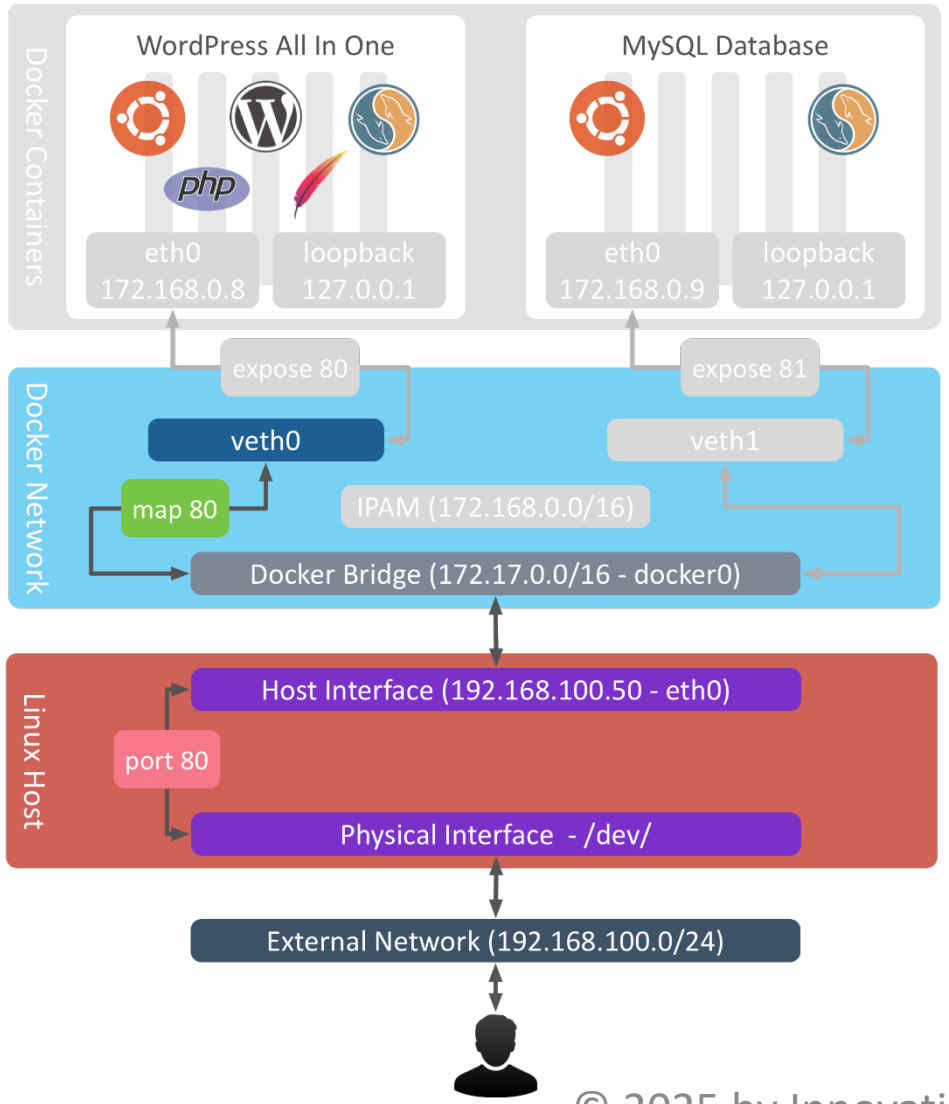
# User Traffic Flow




**User Traffic:**

- Enters from the external network (outside the Docker host)
- Enters the physical interface
- Is passed to the host interface (eth0)
- Is passed to the Docker Bridge
- Is passed to the veth interface
- Is passed to eth0 in the container

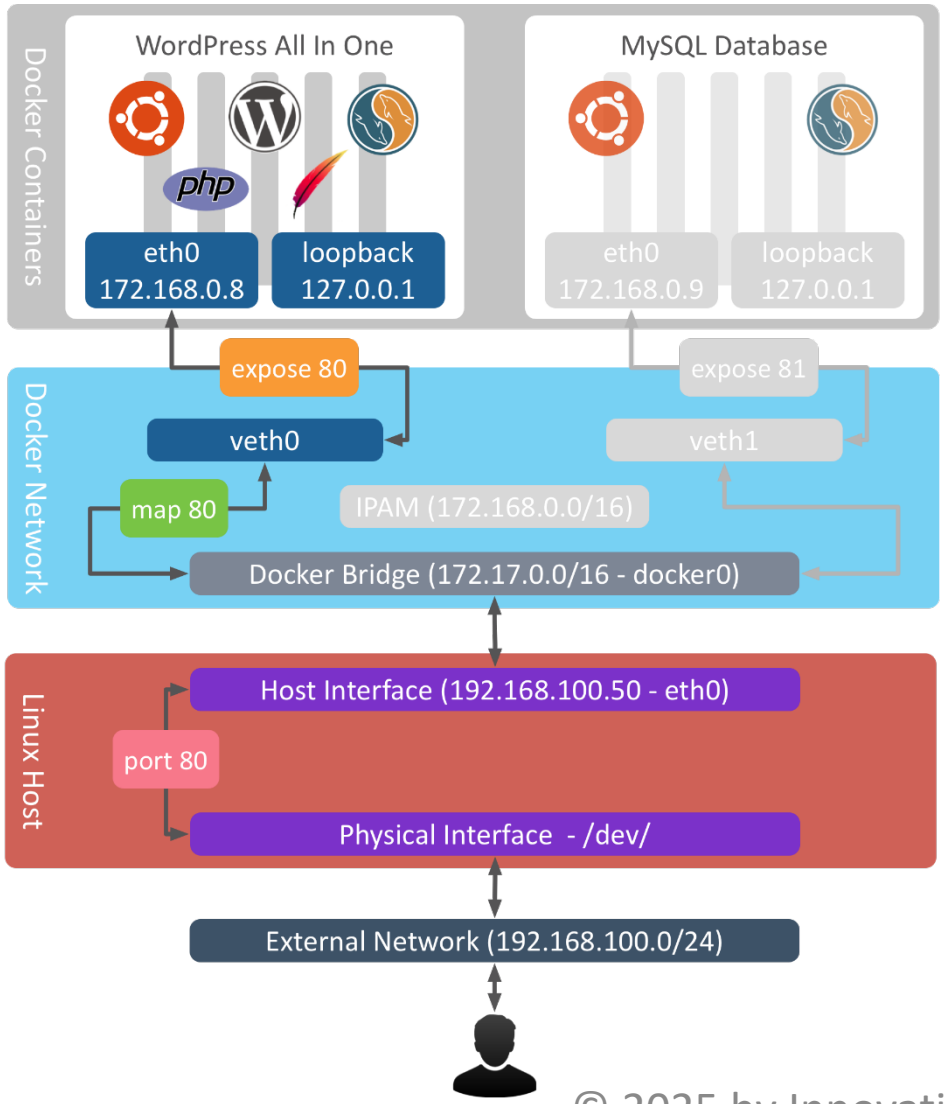
# User Traffic Flow




**User Traffic:**

- Enters from the external network (outside the Docker host)
- Enters the physical interface
- Is passed to the host interface (eth0)
- Is passed to the Docker Bridge
- Is passed to the veth interface
- Is passed to eth0 in the container

# User Traffic Flow




**User Traffic:**

- Enters from the external network (outside the Docker host)
- Enters the physical interface
- Is passed to the host interface (eth0)
- Is passed to the Docker Bridge
- Is passed to the veth interface
- Is passed to eth0 in the container

# Add Networks to Docker Bridge



# Add Networks to Docker Bridge

**Four types of Networks:**

1. **none** (no network access, most secure)
2. **bridge** (limited access to host network, moderately secure)
3. **host** (no restrictions to host networks, least secure)
4. **overlay** (networks with other Docker hosts)

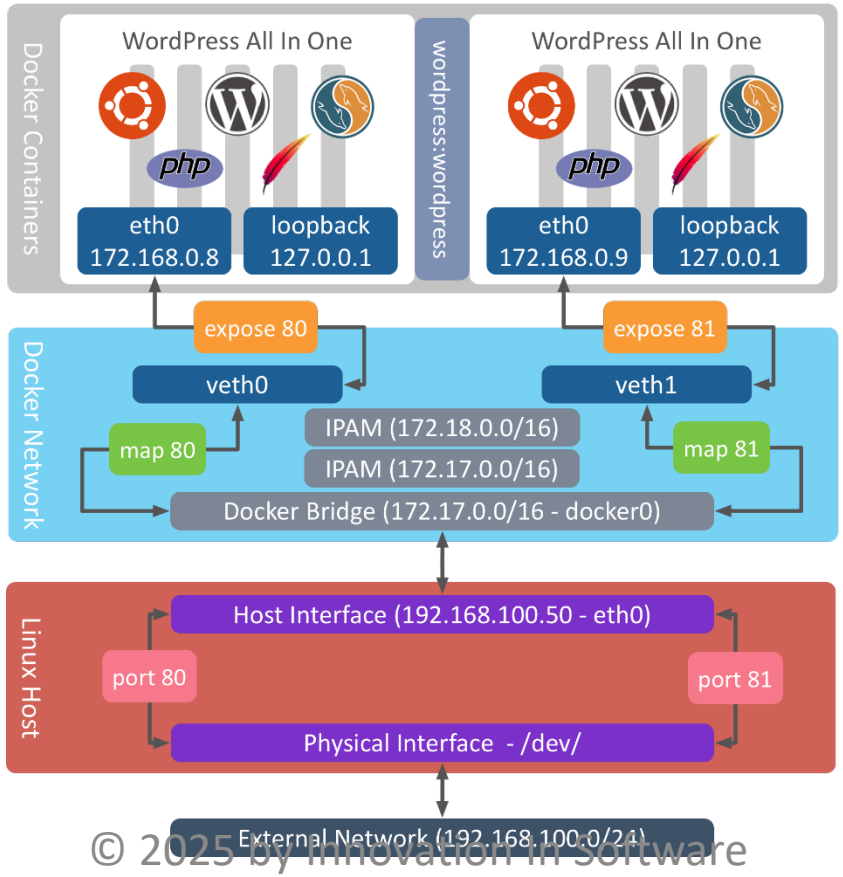
Why use Docker bridge Network segments?

- Use functions of docker bridge to isolate containers from each others by limited **inter-container communication (icc)**
- Provide levels of security in the same docker host

# Add Networks to Docker Bridge

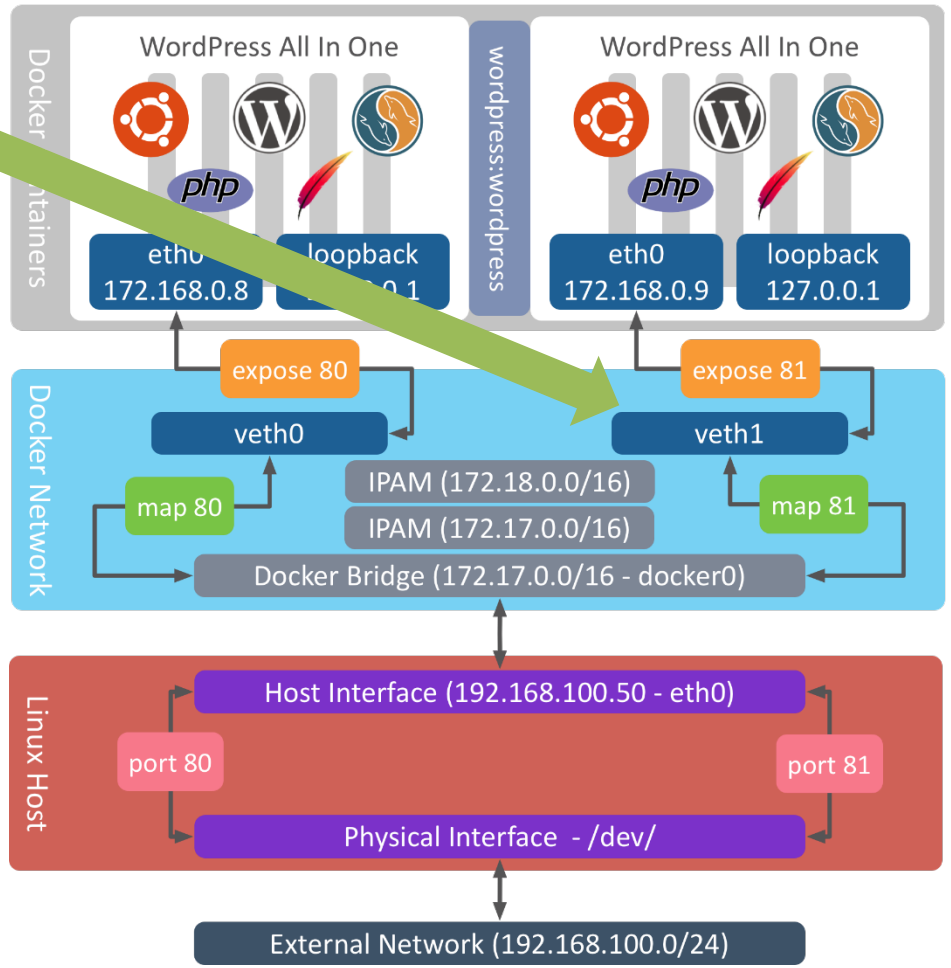
## Create additional networks on bridge docker0

```
$ docker network create database --driver bridge
```



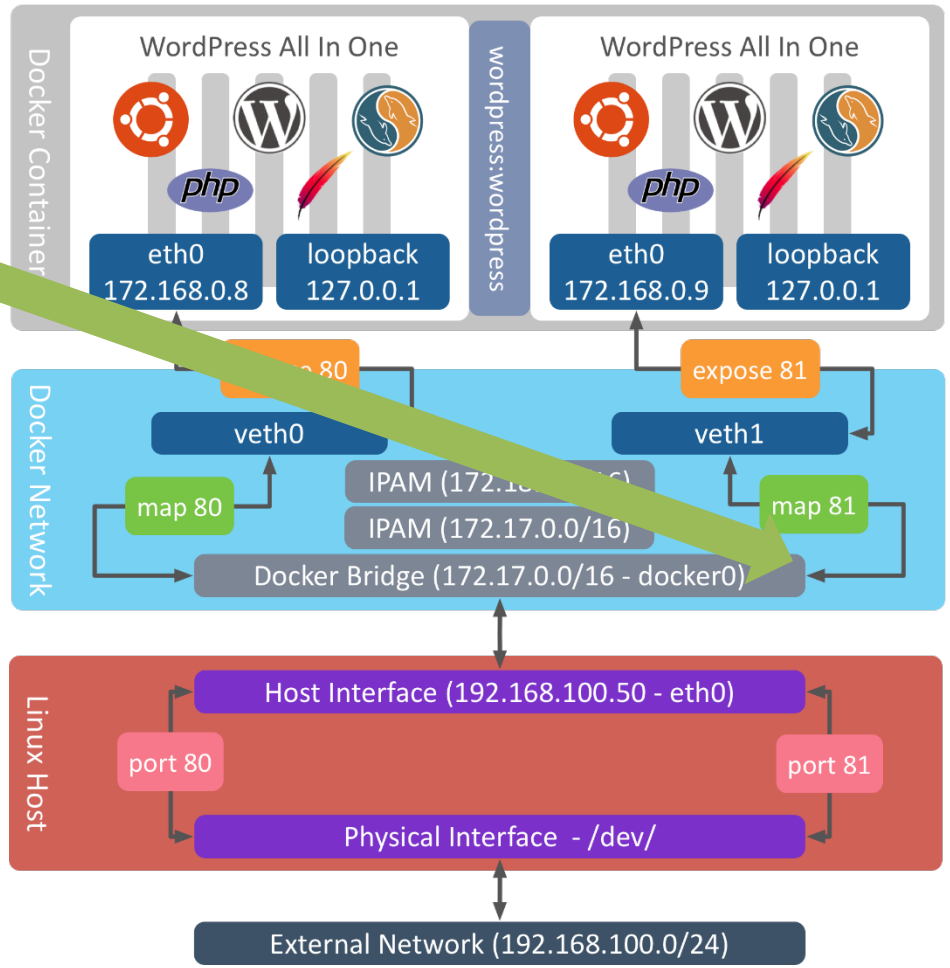
# Add Networks to Docker Bridge

- Creates a veth interface pair
- Connects one end to docker0 bridge
- Connects other end to container eth0
- Assigns an IP address from docker0 IPAM for network



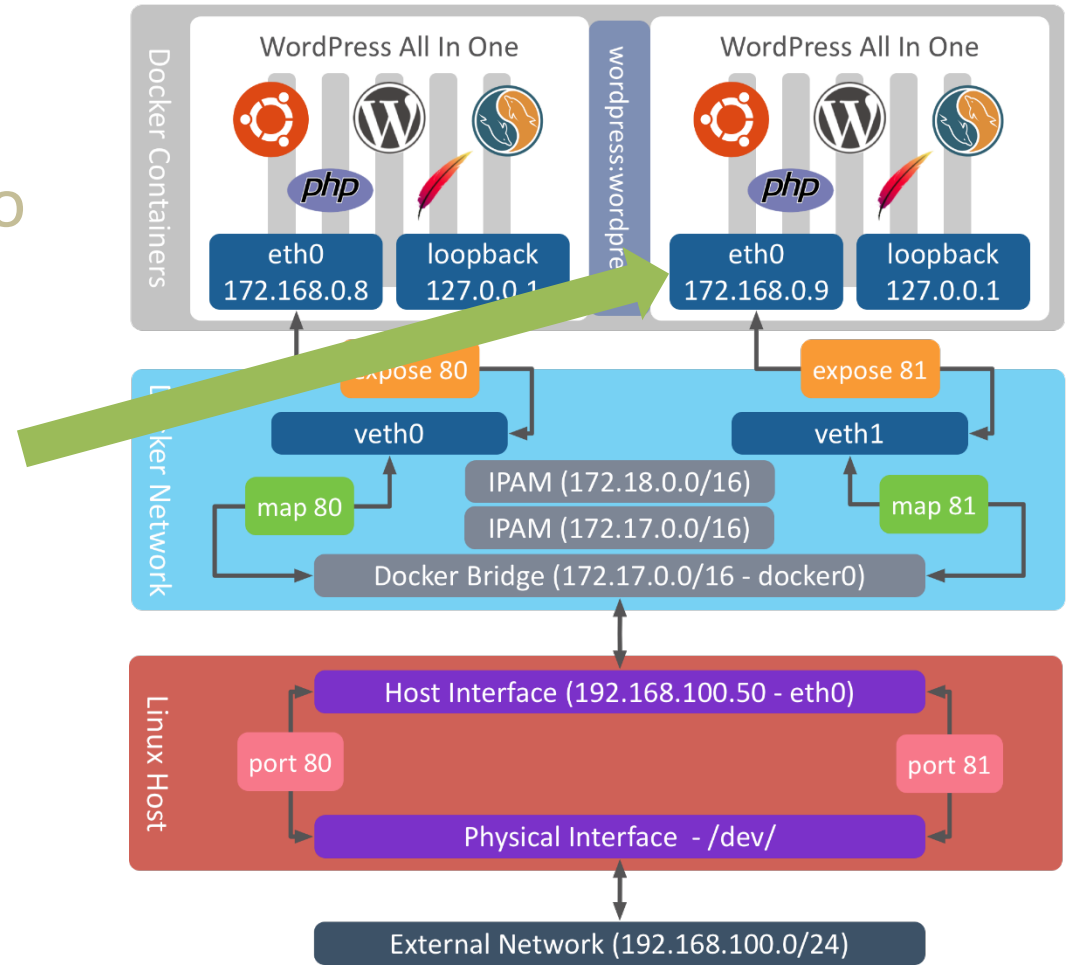
# Add Networks to Docker Bridge

- Creates a veth interface pair
- Connects one end to docker0 bridge
- Connects other end to container eth0
- Assigns an IP address from docker0 IPAM for network



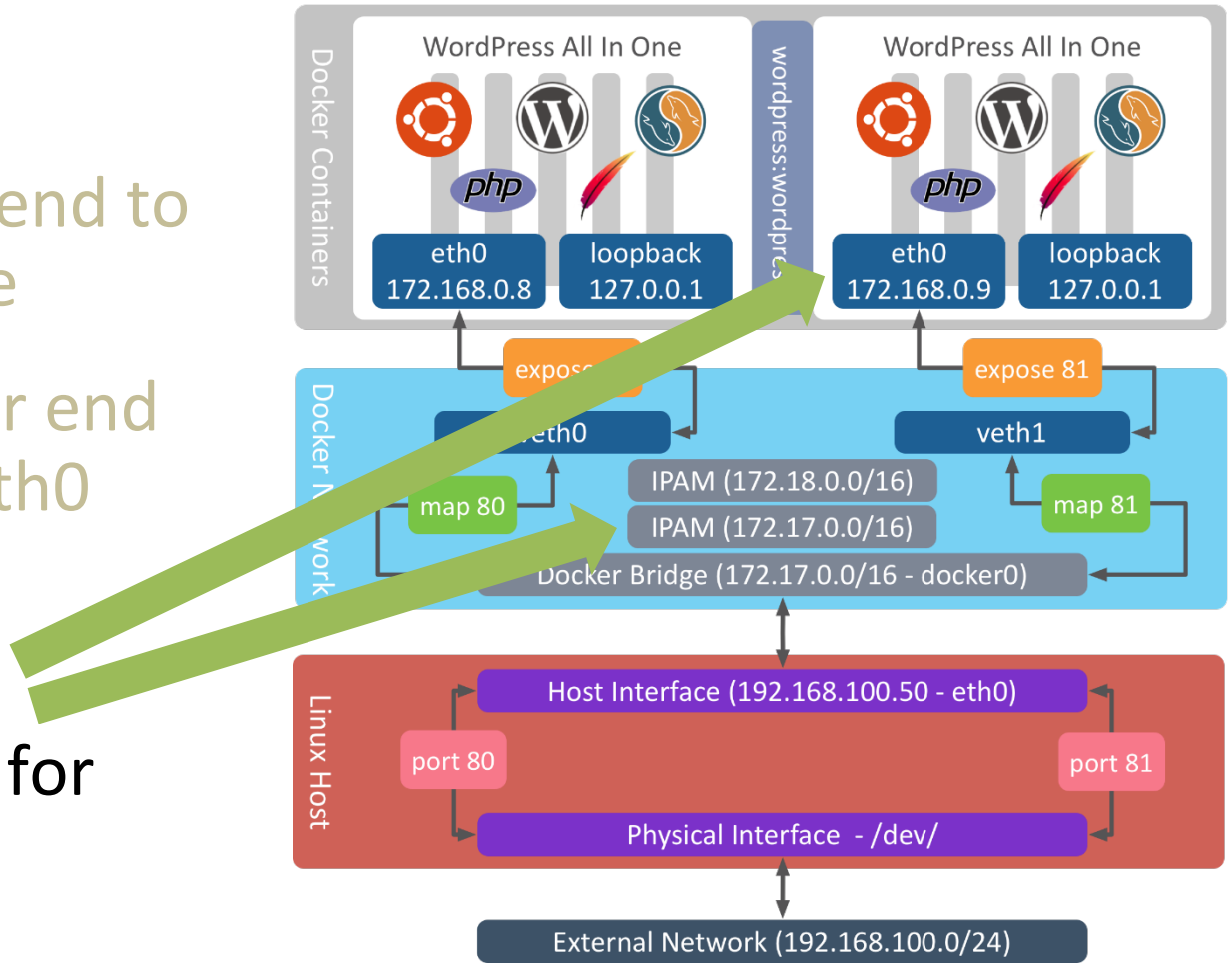
# Add Networks to Docker Bridge

- Creates a veth interface pair
- Connects one end to docker0 bridge
- Connects other end to container eth0
- Assigns an IP address from docker0 IPAM for network



# Add Networks to Docker Bridge

- Creates a veth interface pair
- Connects one end to docker0 bridge
- Connects other end to container eth0
- Assigns an IP address from docker0 IPAM for network



# Add Networks to Docker Bridge

**Create an additional network on docker0:**

- defaults to docker bridge (docker0)

```
$ docker network create database--driver
bridge
```

Network is in `docker network ls` output:

```
$ docker network ls
NETWORK ID          NAME                DRIVER
82cb7fbd657d        bridge              bridge
2e46aa7d66a0        dbase               bridge
957de60d5958        host                host
6b64d7ed1c93        none                null
2c257ae62bcd        wordpress           bridge
```

# Add Networks to Docker Bridge

**Docker network inspect <network>:**

- Provides details on the network

```
$ docker network inspect database
```

- IPAM range, this is the range that the containers are assigned IP Addresses

```
"IPAM": {
            "Driver": "default",
            "Options": {},
            "Config": [
                {
                    "Subnet": "172.19.0.0/16",
                    "Gateway": "172.19.0.1/16"
}
```

# Example Using default network

**Deploy a demo container (default network):**

- network option undefined using “bridge,” the default docker0 network

```
$ docker run -d busybox top
```

- docker inspect formatted for bridge and IPAddress of container.

```
$ docker inspect -f \
{{.NetworkSettings.Networks.bridge.IPAddress}} \
$(docker ps -lq)

172.17.0.5
```

# Example using defined network

**Deploy a demo container (defined network):**

- network option undefined using “bridge,” the default docker0 network

```
$ docker run --network database -d busybox top
```

- docker inspect formatted for bridge and IPAddress of container.

```
$ docker inspect -f \
{{.NetworkSettings.Networks.database.IPAddress}
} \
$(docker ps -lq)

172.19.0.5
```

# Docker Network Metadata



# Network Metadata

**What is it?** – Network metadata describes what the Network is through the use of key value pairs

- **docker network inspect <id>**
  - Queries the Docker Engine, a json formatted output is returned

```
$ docker network inspect <network>
```

```
        "Name"=
         "Id"=
         "Scope"=
         "Driver"=
         "EnableIPv6"=
         "IPAM"=
              "Driver"=
              "Options"=
              "Subnet"=
         "Containers"=
```

# Lab: Docker Networking