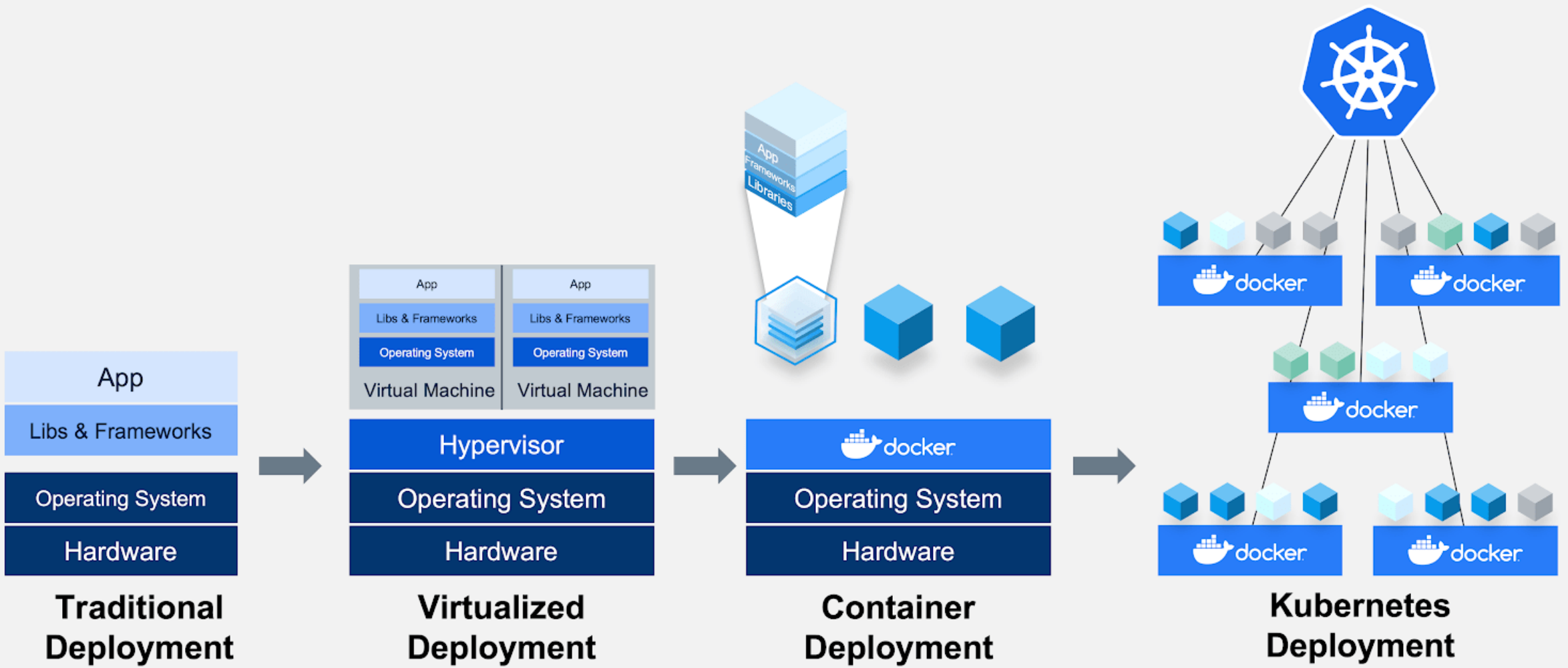
Introduction to

Docker & Docker Swarm

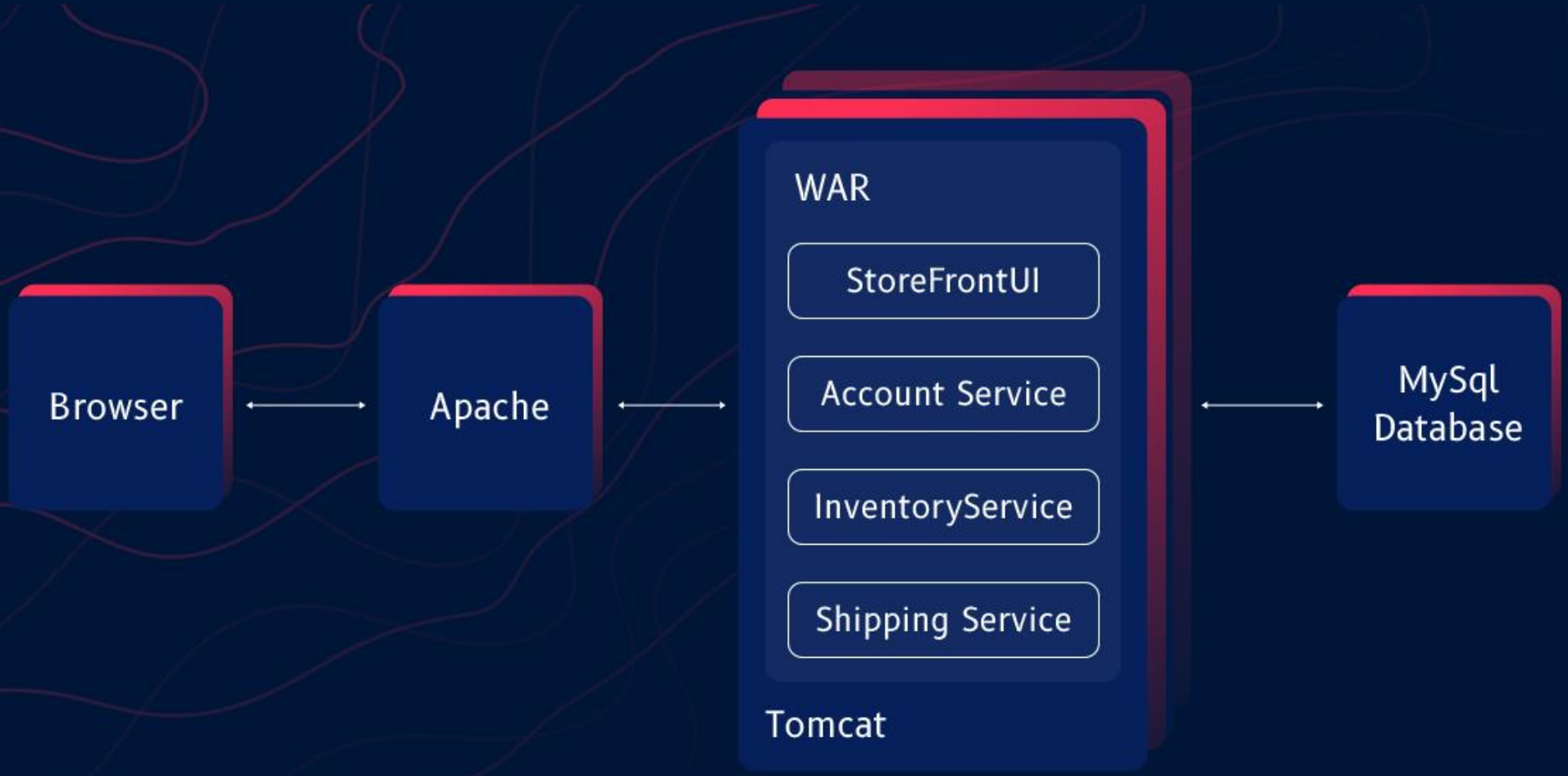
Vikram
IoT Application Dev

github.com/kunchalavikram1427

The past and the present of Apps Deployment



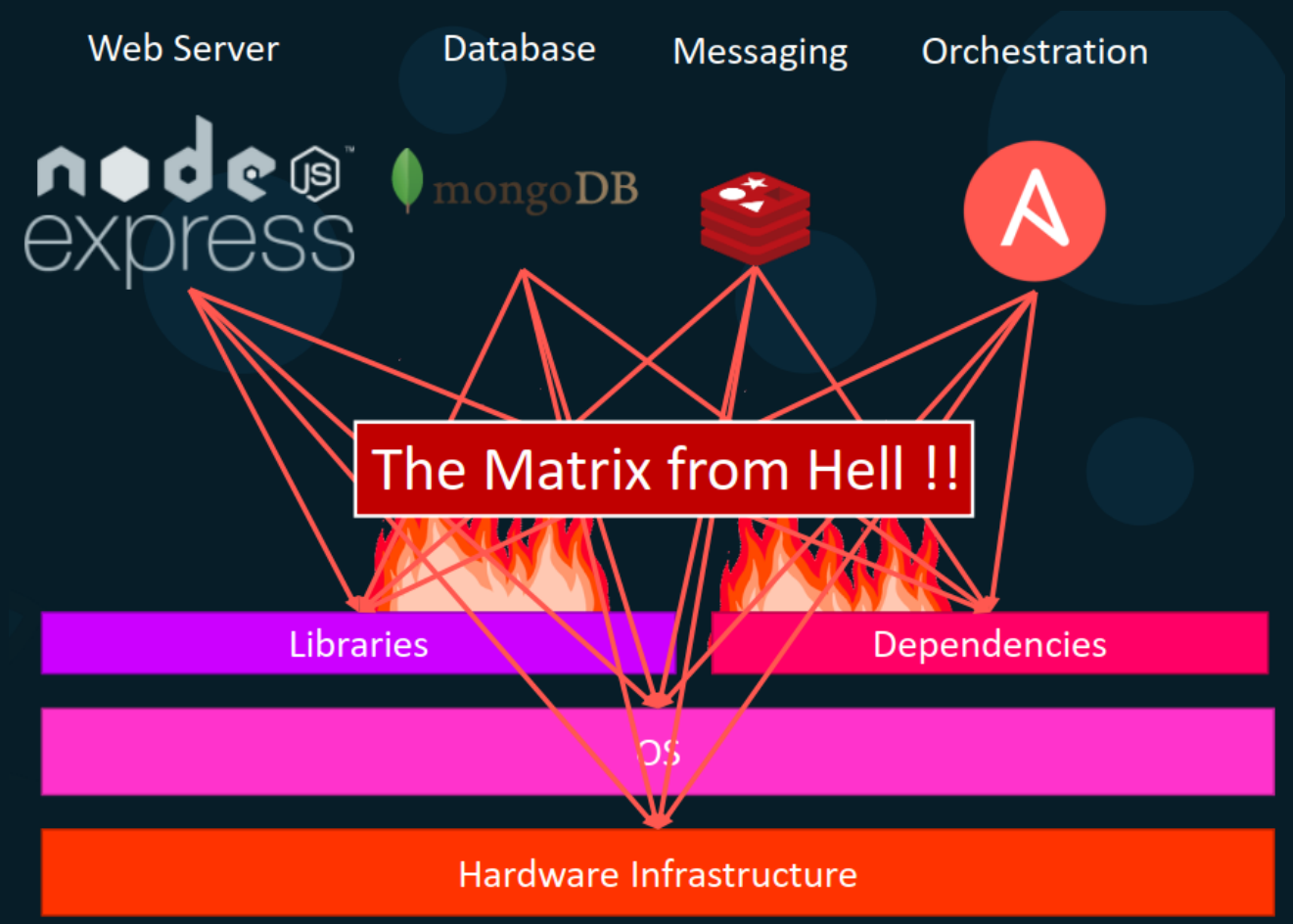
Monolithic Architecture



Simple To : **Develop, Test, Deploy and Scale**

Monolithic Architecture

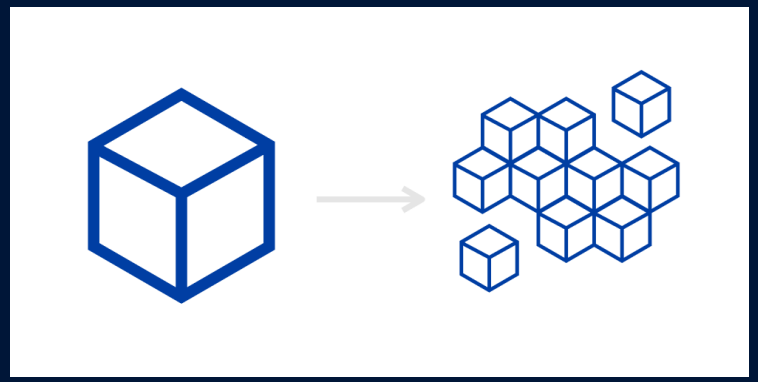
- Different dependency requirements for each service
- Long setup times
- Different Dev/Test/Prod environments



Monolithic Applications

Pros

- Simple to develop
- Simple to deploy – one binary
- Easy Debugging & Error tracing
- Simple to test
- Less Costly

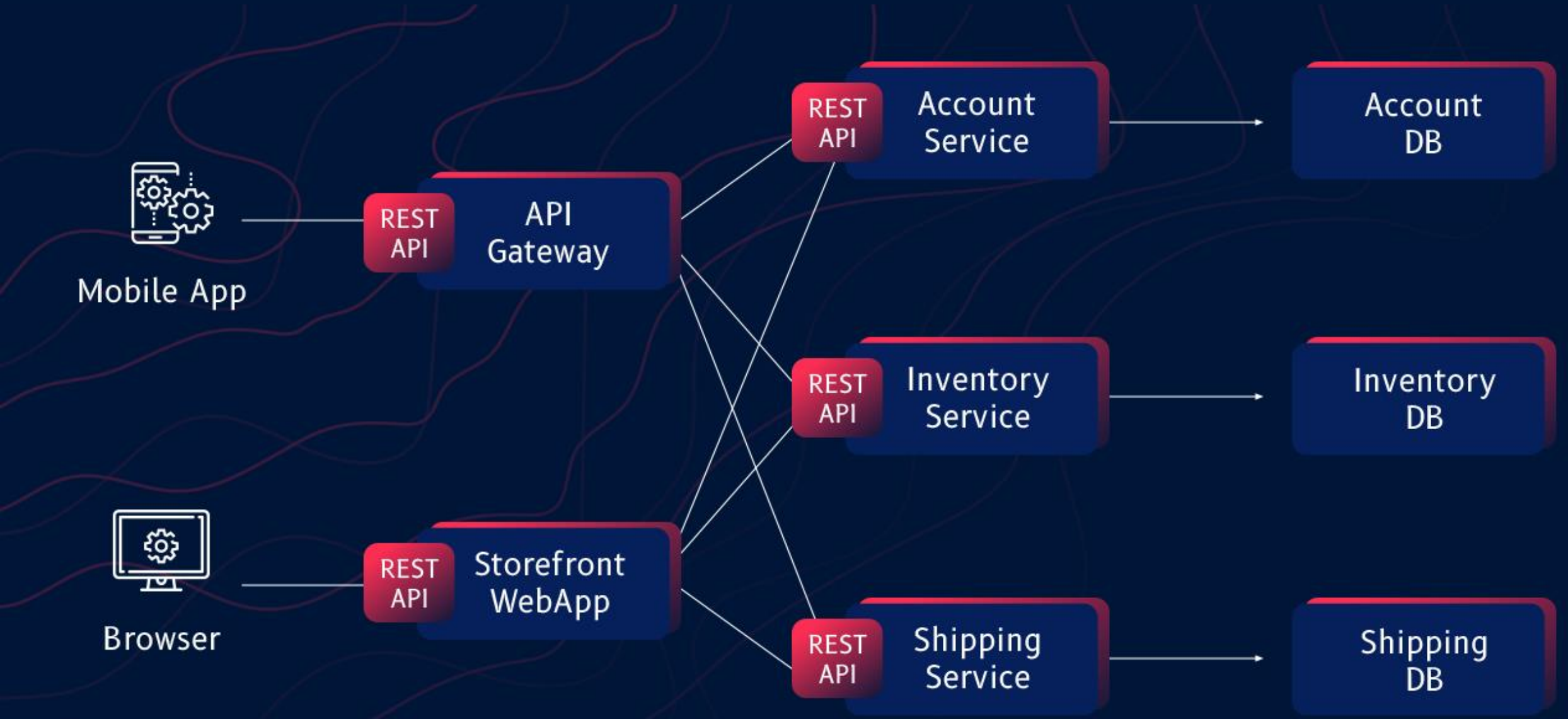


Cons

- Difficult to understand and modify
- Tightly coupled
- Higher start-up and load times
- Redeploy the entire application on each update, and also continuous deployment is difficult
- Less reliable: A single bug can bring down the entire application.
- Scaling the application is difficult
- Difficult to adopt new and advanced technology: Since changes in frameworks or languages will affect an entire application
- Changes are one section of the code can cause an unanticipated impact on the rest of the code

https://www.systango.com/blog/what-should-a-startup-choose-monolith-or-microservices-architecture/

Microservices Architecture



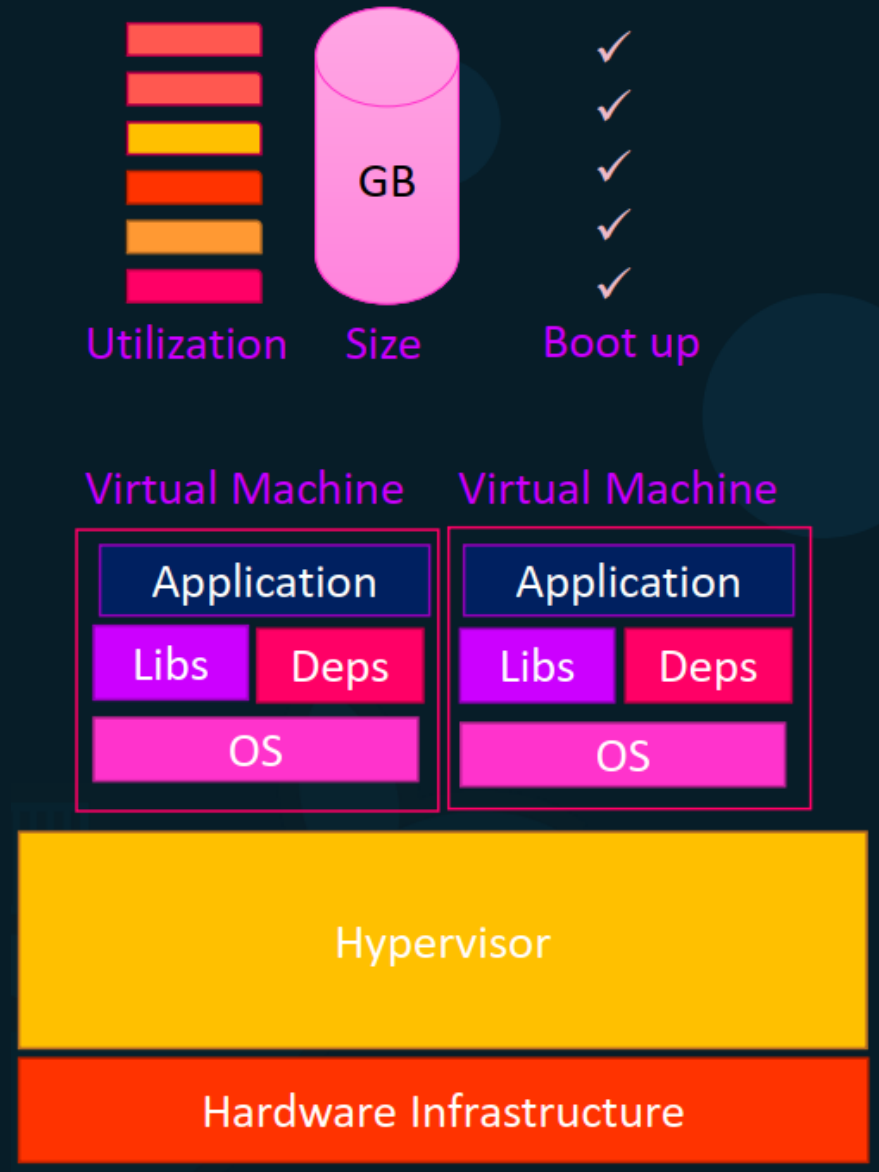
Microservices on VMs

Hypervisor

- A hypervisor is software that creates and runs virtual machines (VMs) also known as guests
- It isolates the hypervisor operating system and resources from the virtual machines and enables the creation and management of those VMs
- The hypervisor treats host resources—like CPU, memory, and storage—as a pool that can be easily reallocated between existing guests or to new virtual machines
- Generally, there are two types of hypervisors

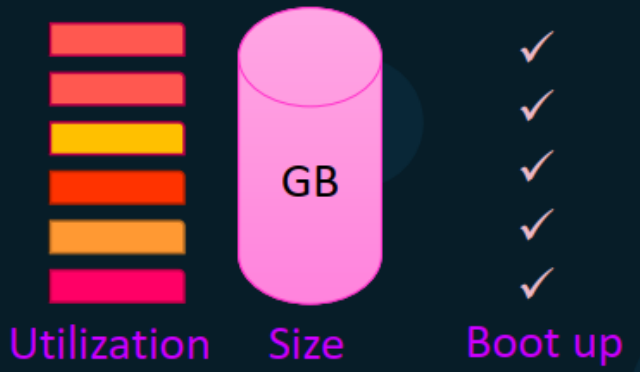
Type 1 hypervisors, called “bare metal,” run directly on the host’s hardware. Ex: Microsoft Hyper-V or VMware ESXi hypervisor

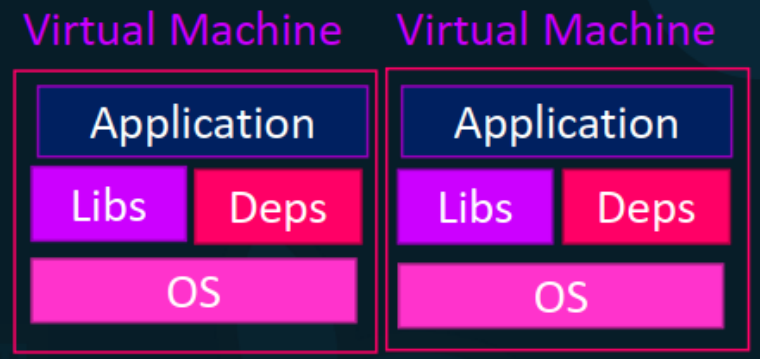
Type 2 hypervisors, called “hosted,” run as a software layer on an operating system. Ex: VirtualBox, VMware Player

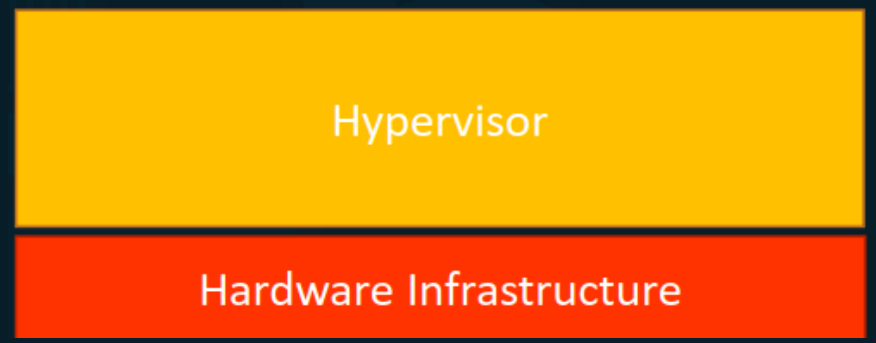


Microservices on VMs

- Run each service with its own dependencies in separate VMs
- Each VM has its own underlying OS and hosts a Microservice
- Strong isolation and resource control between other VMs and host
- Each VM can have its own dependencies and libraries for the services. So different services across VMs can have different versions of same dependency
- Matrix from hell problem is no more







Microservices based Applications

Pros

- Decoupled
- Ensures continuous delivery and deployment of large, complex applications.
- Better testing — since services are smaller and faster to test.
- Better deployments — each service can be deployed independently.
- No long-term commitment to technology – when developing a new service, you can start with a new technology stack.

Cons

- Slow bootup times of VMs
- Increased memory consumption
- Large OS footprint
- Initial Costs are very High and this type of architecture demands for proficiency in the skills of the developers.
- Testing is difficult and time-consuming because there is an additional complexity involved because of the distributed system.
- Deployment Complexity – there is added operational complexity of deploying and managing a system that contains various service types.


docker

Docker

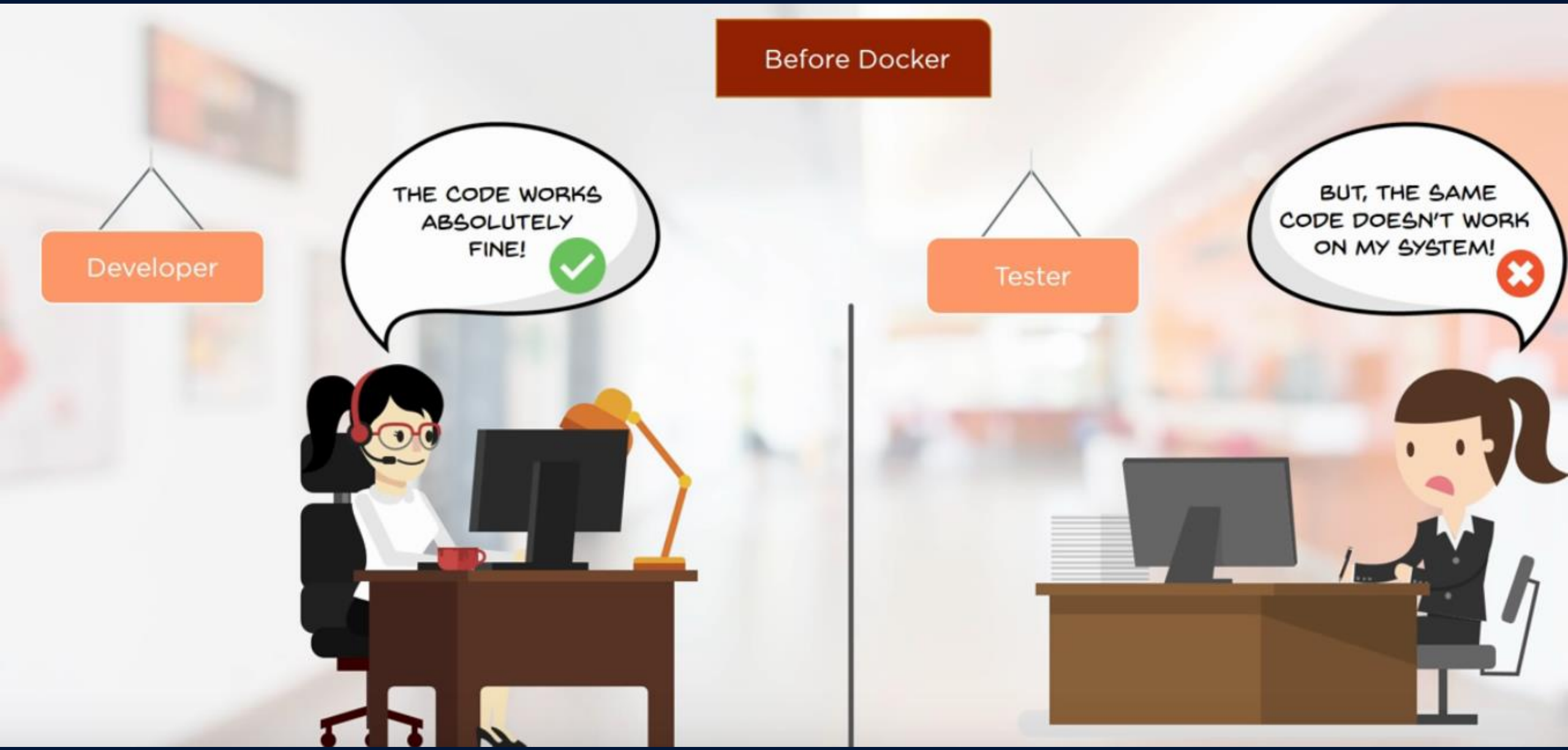
Dev: It works fine in my system!
Tester: It doesn't work in my system

Before Docker

A developer sends code to a tester but it doesn't run on the tester's system due to various dependency issues, however it works fine on the developer's end.

After Docker

As the tester and developer now have the same system running on Docker container, they both are able to run the application in the Docker environment without having to face differences in dependencies issue as before.


Before Docker
Developer
THE CODE WORKS ABSOLUTELY FINE!
Tester
BUT, THE SAME CODE DOESN'T WORK ON MY SYSTEM!

After Docker

Developer

THE CODE WORKS ABSOLUTELY FINE!

Tester

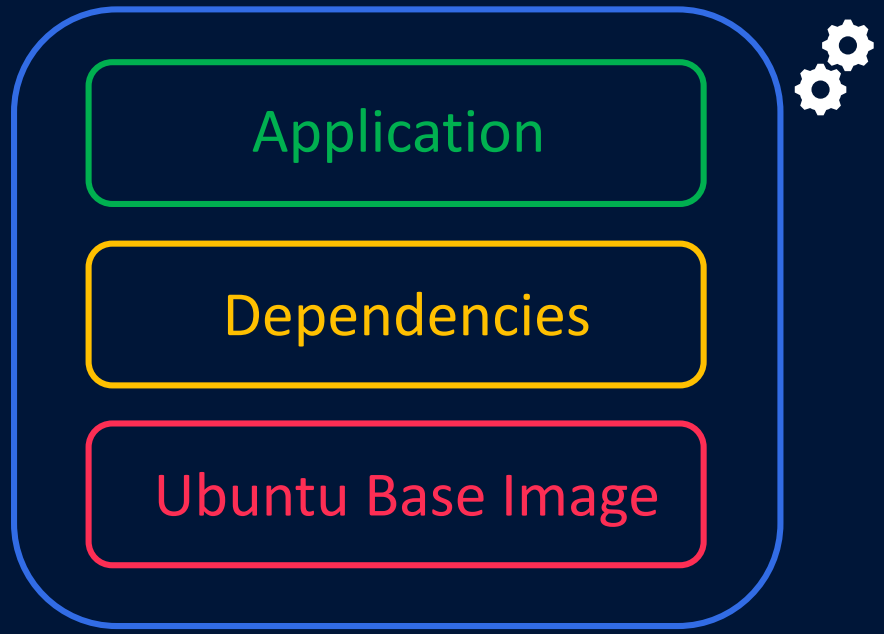
NOW, THE CODE WORKS FOR ME TOO!!

Docker

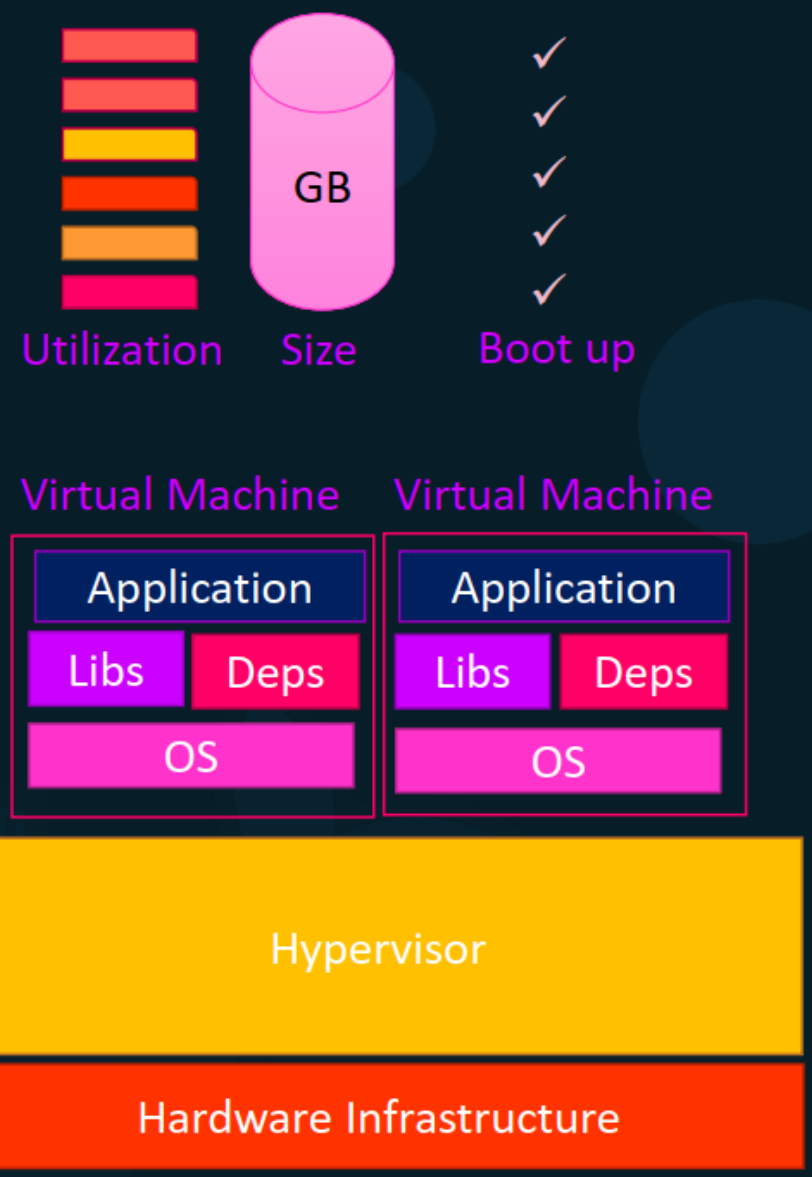
Docker is a software development tool and a virtualization technology that makes it easy to develop, deploy, and manage applications by using containers.

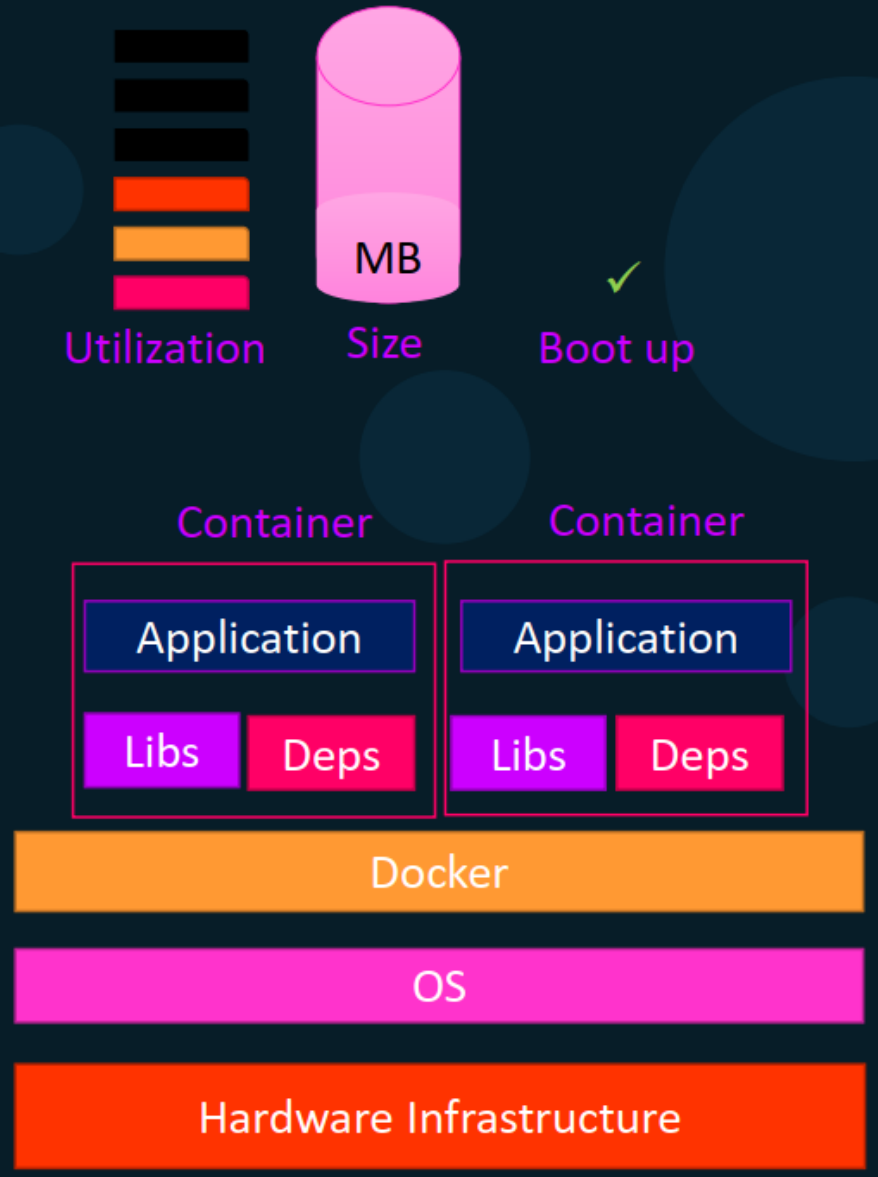
Container refers to a lightweight, stand-alone, executable package of a piece of software that contains all the libraries, configuration files, dependencies, and other necessary parts to operate the application.

Ex: Ubuntu + Python + Dependencies



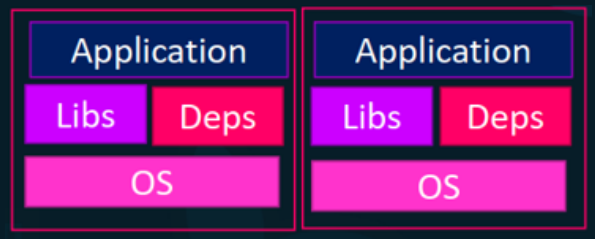
VMs vs Docker Containers





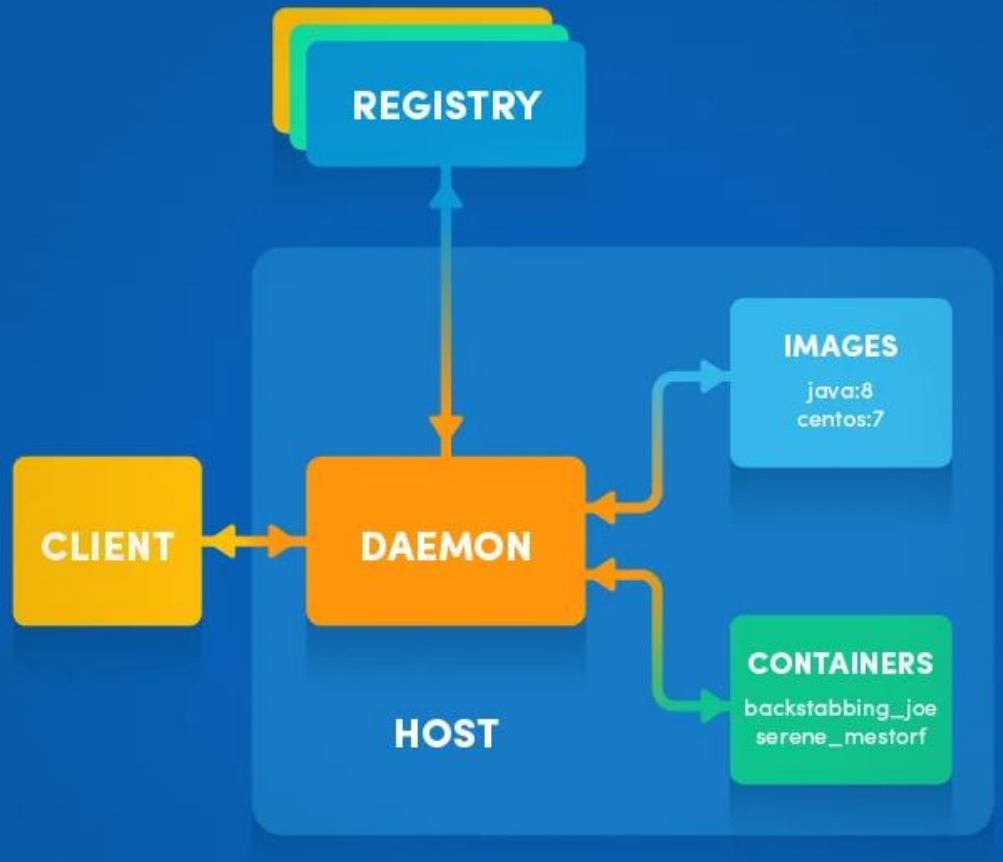
VMs vs Docker Containers

Virtual Machine	Docker Container
Hardware-level process isolation	OS level process isolation
Each VM has a separate OS	Each container can share OS
Boots in minutes	Boots in seconds
VMs are of few GBs	Containers are lightweight (KBs/MBs)
Ready-made VMs are difficult to find	Pre-built docker containers are easily available
VMs can move to new host easily	Containers are destroyed and re-created rather than moving
Creating VM takes a relatively longer time	Containers can be created in seconds
More resource usage	Less resource usage

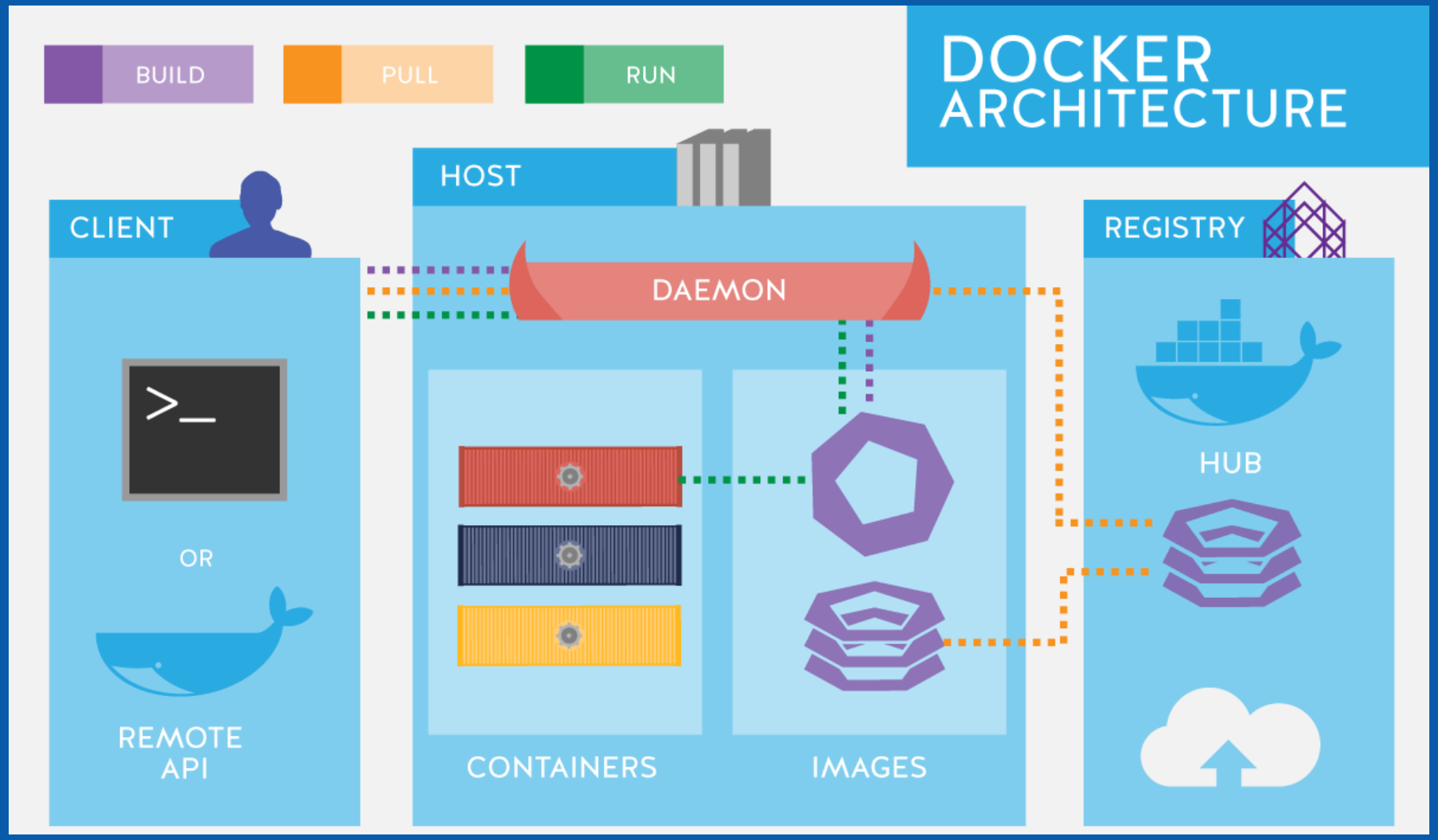


Docker Architecture

- Docker uses a client-server architecture.
- Docker client talks to the Docker daemon, which does the heavy lifting of building, running, and distributing your Docker containers.
- Docker client and daemon can run on the same system, or you can connect a Docker client to a remote Docker daemon.
- For a virtual communication between CLI client and Docker daemon, a REST API is used

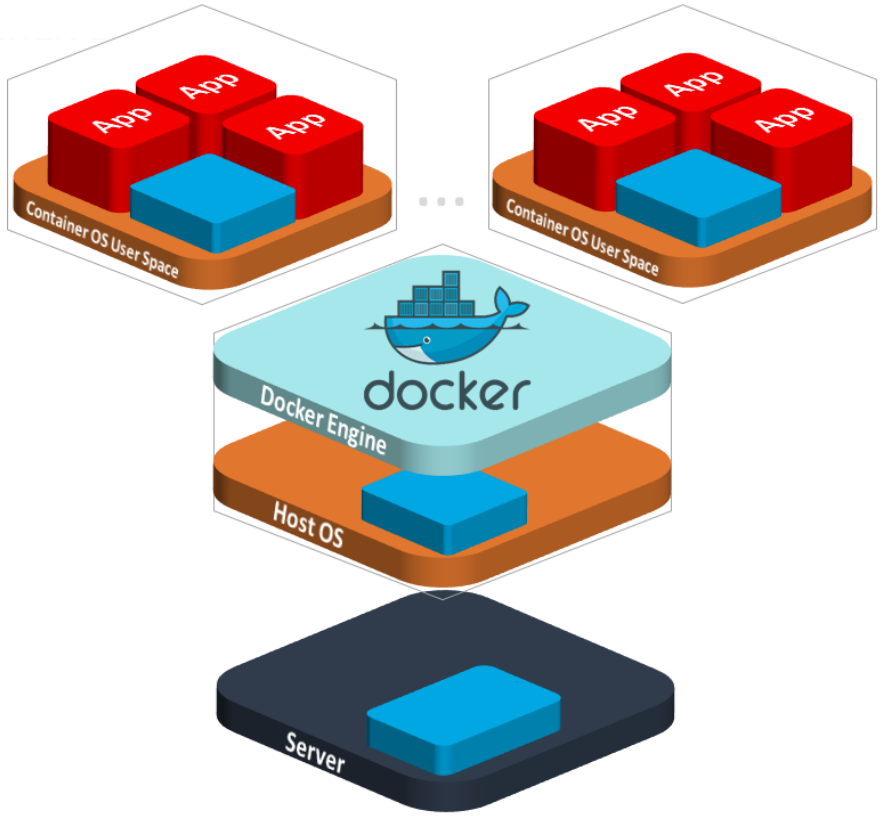


Docker Architecture



Docker Installation

- **Docker for Windows**: Win10 Pro/Ent only
  Uses Hyper-V with tiny Linux VM for Linux Containers
- **Docker Toolbox:** Win7/8/8.1 or Win10 Home
  Runs a tiny Linux VM in VirtualBox
- **Docker for MAC**
- **Docker for Linux**



Online Emulator: https://labs.play-with-docker.com/

https://docs.docker.com/get-docker/

Docker basic commands

docker version

```
root@k-master:/home/osboxes# docker version
Client:
 Version:           19.03.6
 API version:       1.40
 Go version:        go1.12.17
 Git commit:        369ce74a3c
 Built:             Fri Feb 28 23:45:43 2020
 OS/Arch:           linux/amd64
 Experimental:      false

Server:
 Engine:
  Version:          19.03.6
  API version:      1.40 (minimum version 1.12)
  Go version:       go1.12.17
  Git commit:       369ce74a3c
  Built:            Wed Feb 19 01:06:16 2020
  OS/Arch:          linux/amd64
  Experimental:     false
 containerd:
  Version:          1.3.3-0ubuntu1~18.04.2
  GitCommit:
 runc:
  Version:          spec: 1.0.1-dev
  GitCommit:
 docker-init:
  Version:          0.18.0
  GitCommit:
```

Docker basic commands

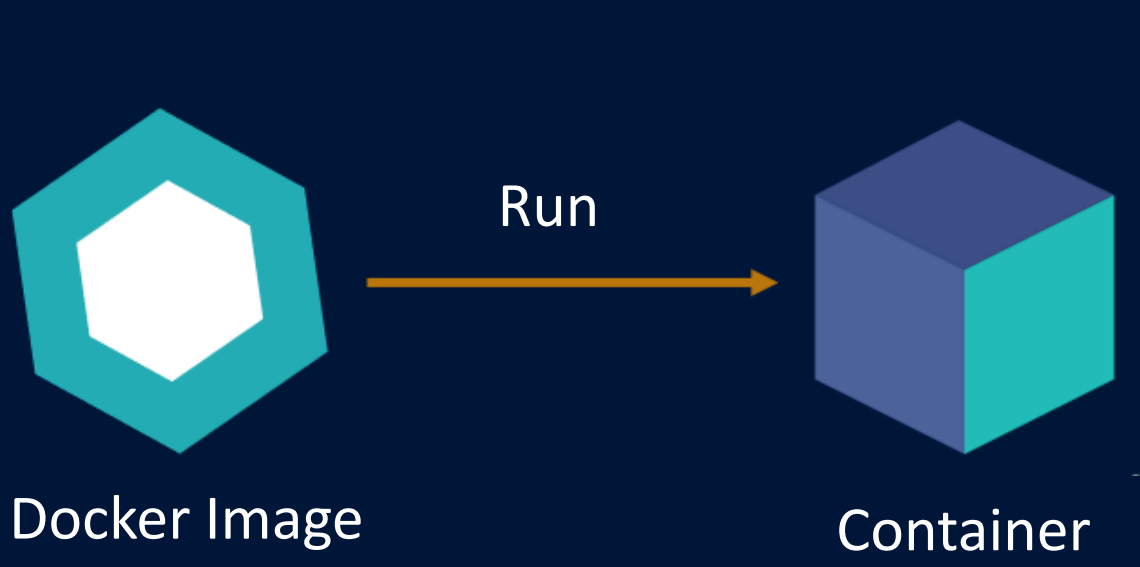
docker info

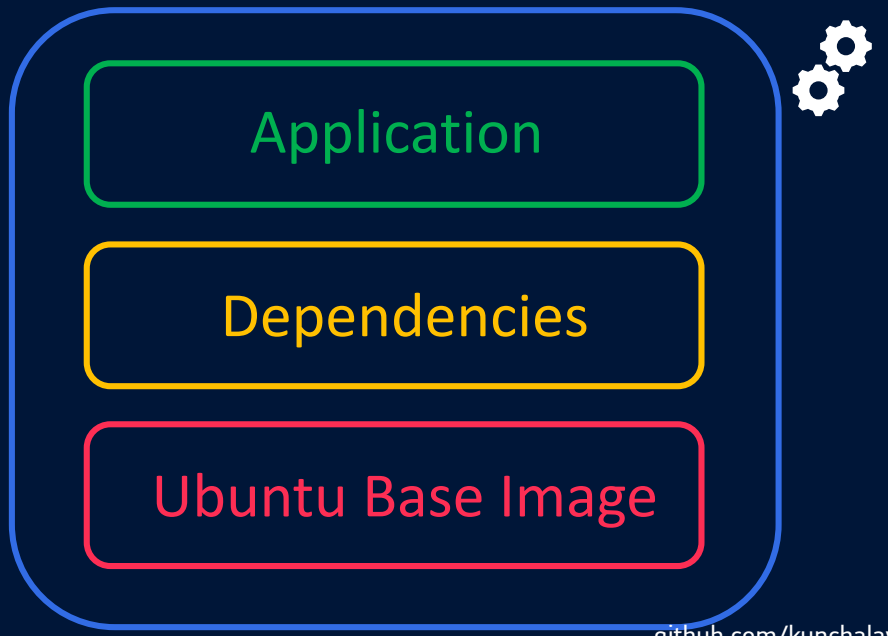
```
root@k-master:/home/osboxes# docker info
Client:
 Debug Mode: false

Server:
 Containers: 17
  Running: 16
  Paused: 0
  Stopped: 1
 Images: 11
 Server Version: 19.03.6
 Storage Driver: overlay2
  Backing Filesystem: extfs
  Supports d_type: true
  Native Overlay Diff: true
 Logging Driver: json-file
 Cgroup Driver: cgroupfs
 Plugins:
  Volume: local
  Network: bridge host ipvlan macvlan null overlay
  Log: awslogs fluentd gcplogs gelf journald json-file local logentries splunk syslog
 Swarm: inactive
 Runtimes: runc
 Default Runtime: runc
 Init Binary: docker-init
 containerd version:
 runc version:
 init version:
 Security Options:
  apparmor
  seccomp
   Profile: default
 Kernel Version: 5.0.0-23-generic
 Operating System: Ubuntu 18.04.3 LTS
```

Docker Images & Containers

- Docker Image: Docker image can be compared to a template that is used to create Docker containers. These are read-only templates that contains application binaries and dependencies. Docker images are stored in the Docker Registry.

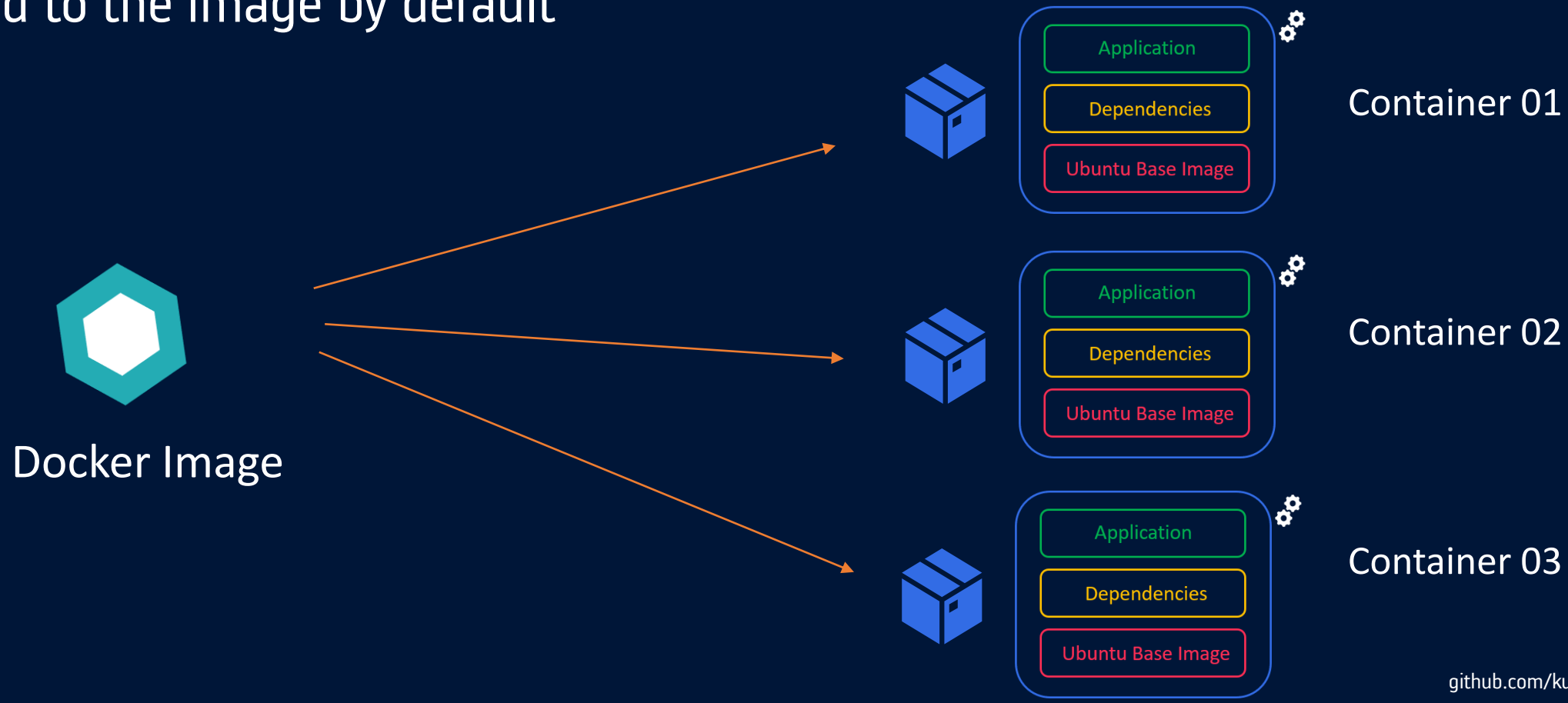
- Docker Container: Docker container is a running instance of a Docker image as they hold the entire package needed to run the application.





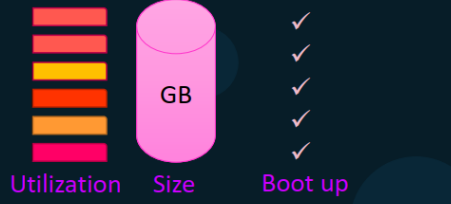
Docker Images & Containers

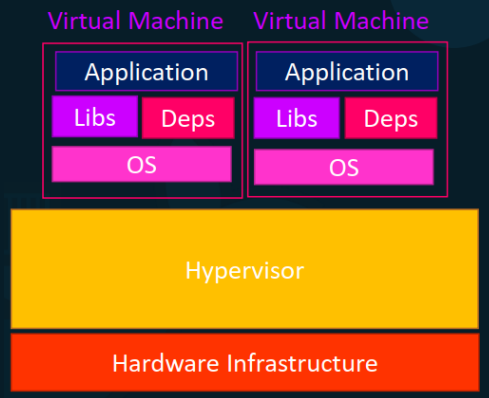
- We can run any number of containers based out of an image and Docker makes sure that each container created has a unique name in the namespace.
- Docker image is a read-only template. Changes made in containers won't be saved to the image by default

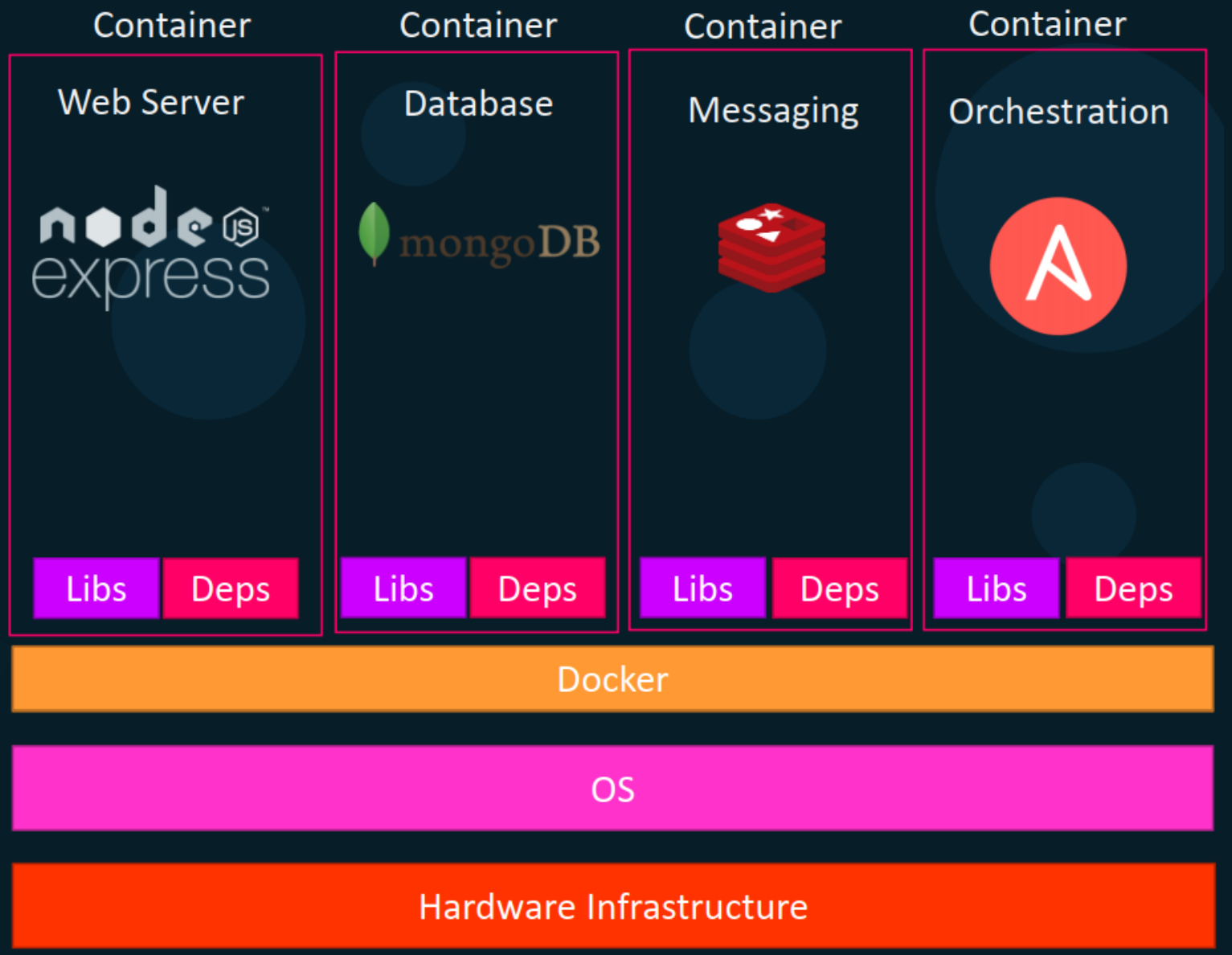


Docker Images & Containers

Containers run each service with its own dependencies in separate containers

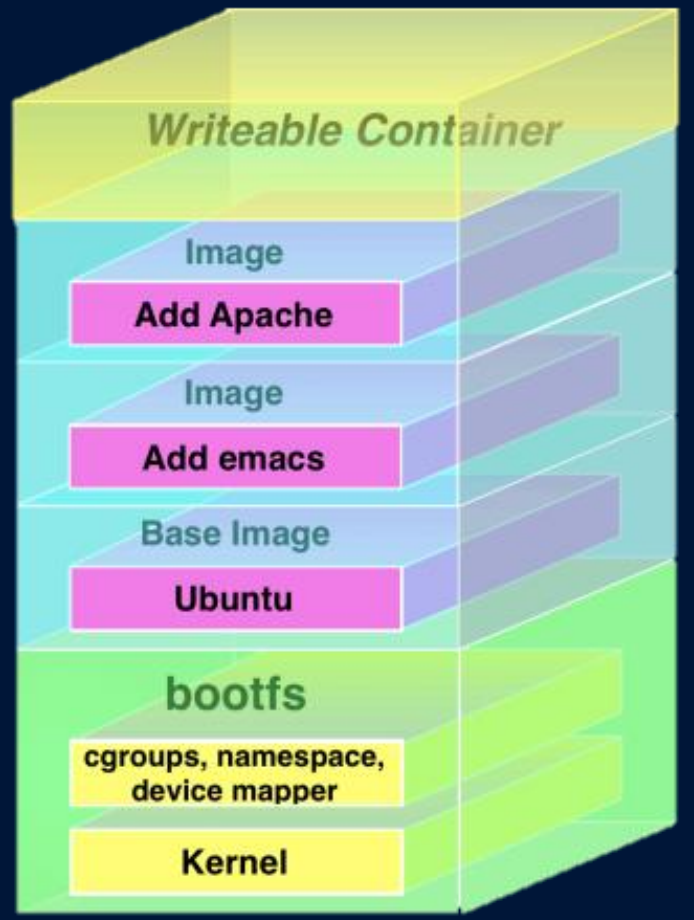






Union File System

- A docker image is a read-only template for creating containers.
- Changes made to the file system inside the running container won't be directly saved on to the image.
- Instead , if a container needs to change a file from the read-only image that provides its filesystem, it copies the file up to its own private read-write layer before making the change
- This is called copy-on-write (COW) mechanism.
- These new or modified files and directories are 'committed' as a new layer.
- docker history command shows all these layers.



Docker Registry

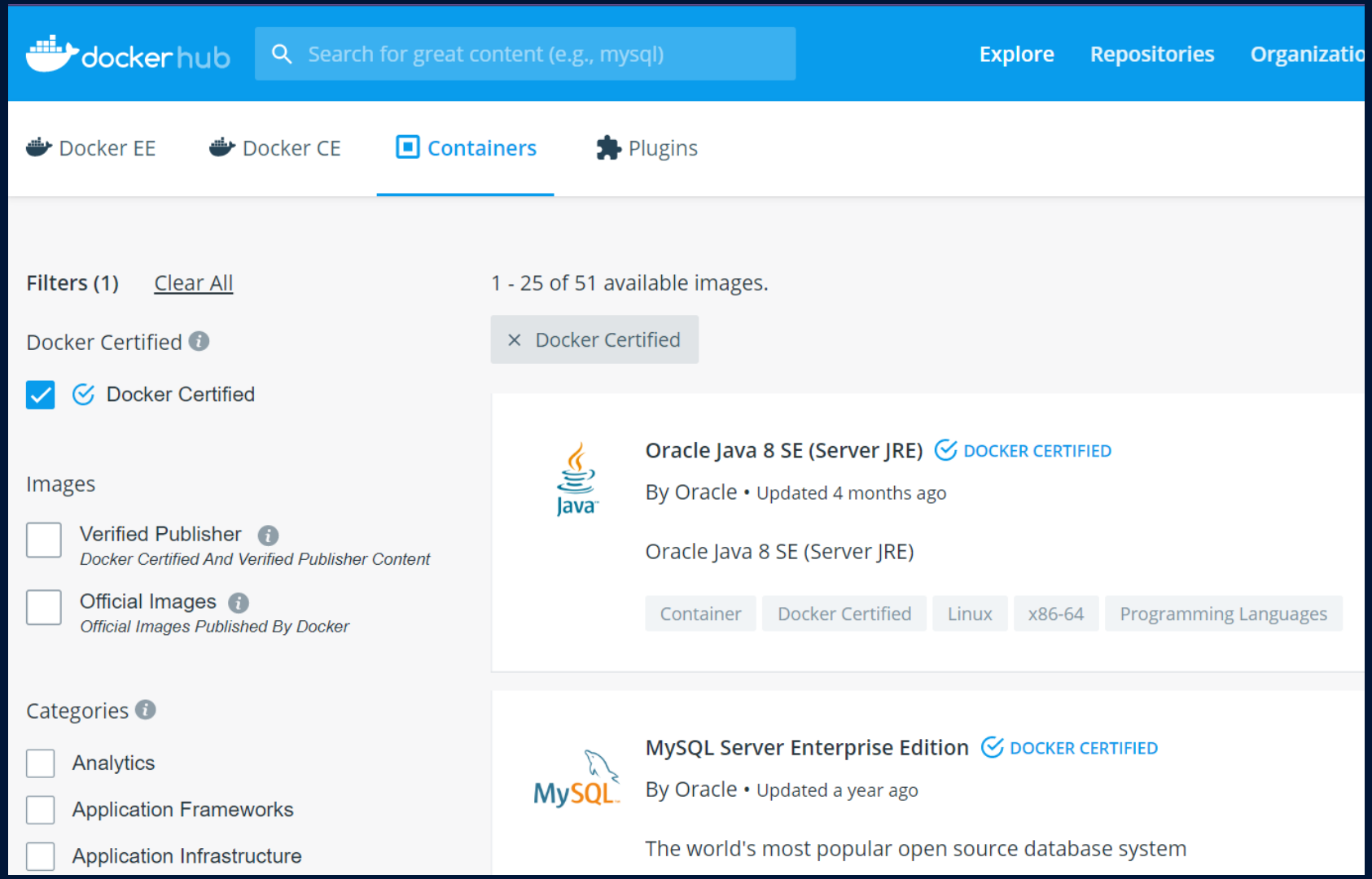
- Docker registry is a storage and distribution system for Docker images.
- It is organized into Docker repositories , where a repository holds all the versions of a specific image.
- By default, the Docker engine interacts with DockerHub , Docker's public registry instance.
- However, it is possible to run on-premise private repositories
  Ex: Harbor

Some Cloud Provider repos

- Amazon Elastic Container Registry
- Google Container Registry
- Azure Container Registry

Docker Registry

- https://hub.docker.com/



Docker Registry

ubuntu ☆

Docker Official Images

Ubuntu is a Debian-based Linux operating system based on free software.

1B+

Container | Linux | 386 | IBM Z | PowerPC 64 LE | ARM 64 | x86-64 | ARM | Base Images | Operating Systems | Official Image

Image name: username/image-name

Linux - ARM 64 (latest)

Copy and paste to pull this image

```
docker pull ubuntu
```

View Available Tags

Description | Reviews | Tags

Supported tags and respective `Dockerfile` links

- `18.04` , `bionic-20200403` , `bionic`
- `19.10` , `eoan-20200410` , `eoan`
- `20.04` , `focal-20200423` , `focal` , `latest` , `rolling`
- `20.10` , `groovy-20200505` , `groovy` , `devel`
- `14.04` , `trusty-20191217` , `trusty`
- `16.04` , `xenial-20200326` , `xenial`

docker pull ubuntu:19.10

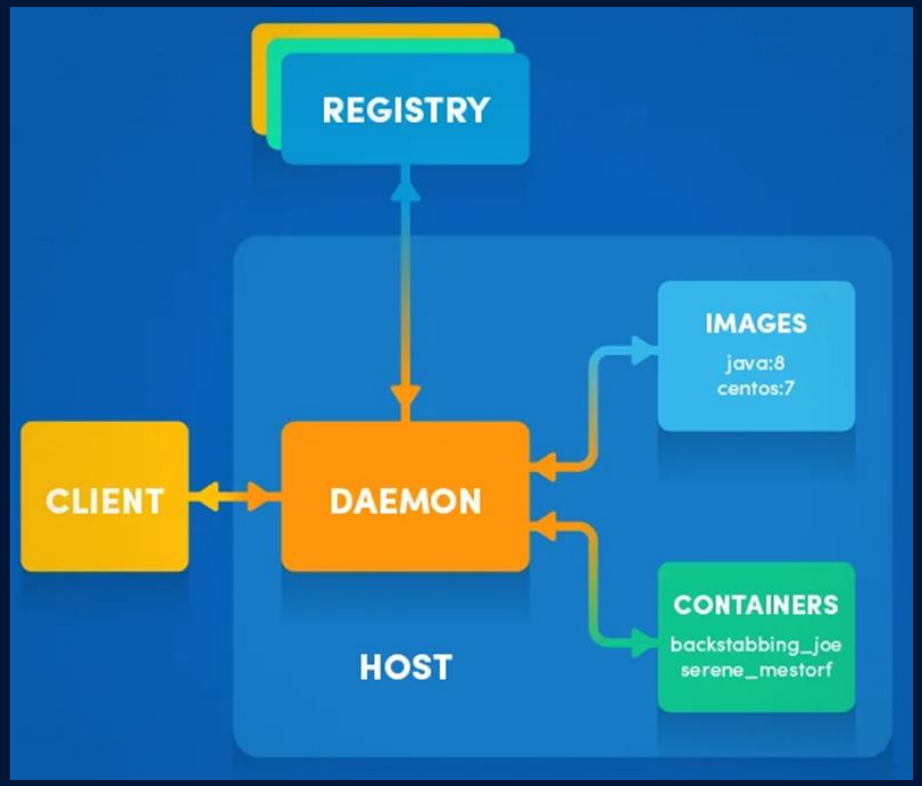
Docker pull

- By default, docker pull will pull an image from docker hub. If you need to pull from a private repo, use repo URL

  docker pull myregistry.local:5000/testing/test-image

  docker pull ubuntu

```
root@k-master:/home/osboxes# docker pull ubuntu
Using default tag: latest
latest: Pulling from library/ubuntu
d51af753c3d3: Pull complete
fc878cd0a91c: Pull complete
6154df8ff988: Pull complete
fee5db0ff82f: Pull complete
Digest:
sha256:747d2dbbaaee995098c9792d99bd333c6783ce56150d1b11e3
33bbceed5c54d7
Status: Downloaded newer image for ubuntu:latest
```



Docker pull

- docker images

```
root@k-master:/home/osboxes# docker images
REPOSITORY                           TAG                 IMAGE ID            CREATED             SIZE
mongo                                latest              3f3daf863757        2 weeks ago         388MB
ubuntu                               latest              1d622ef86b13        3 weeks ago         73.9M
B
nginx                                latest              602e111c06b6        3 weeks ago         127MB
k8s.gcr.io/kube-proxy                v1.18.2             0d40868643c6        4 weeks ago         117MB
k8s.gcr.io/kube-controller-manager   v1.18.2             ace0a8c17ba9        4 weeks ago         162MB
k8s.gcr.io/kube-apiserver            v1.18.2             6ed75ad404bd        4 weeks ago         173MB
k8s.gcr.io/kube-scheduler            v1.18.2             a3099161e137        4 weeks ago         95.3M
B
quay.io/coreos/flannel               v0.12.0-amd64       4e9f801d2217        2 months ago        52.8M
B
k8s.gcr.io/pause                     3.2                 80d28bedfe5d        3 months ago        683kB
k8s.gcr.io/coredns                   1.6.7               67da37a9a360        3 months ago        43.8M
B
k8s.gcr.io/etcd                      3.4.3-0             303ce5db0e90        6 months ago        288MB
```

Docker home directory

/var/lib/docker

```
root@k-master:/var/lib/docker# ls -l
total 104
drwx------  2 root root  4096 May  4 08:32 builder
drwx--x--x  4 root root  4096 May  4 08:32 buildkit
drwx------ 19 root root 20480 May 14 22:11 containers
drwx------  3 root root  4096 May  4 08:32 image
drwxr-x---  3 root root  4096 May  4 08:32 network
drwx------ 78 root root 40960 May 14 22:22 overlay2
drwx------  4 root root  4096 May  4 08:32 plugins
drwx------  2 root root  4096 May 13 09:31 runtimes
drwx------  2 root root  4096 May  4 08:32 swarm
drwx------  2 root root  4096 May 14 22:22 tmp
drwx------  2 root root  4096 May  4 08:32 trust
drwx------  2 root root  4096 May  4 08:32 volumes
root@k-master:/var/lib/docker#
```

docker stores all the images, containers, volumes, networks information in it’s default home directory

Docker basic commands

Format: docker <command> <sub-command>

docker image ls

docker help

```
root@k-node02:/home/osboxes# docker help

Usage:  docker [OPTIONS] COMMAND

A self-sufficient runtime for containers

Options:
      --config string      Location of client config files (default "/root/.docker")
  -c, --context string     Name of the context to use to connect to the daemon (overrides DOCKER_HOST env var
                           context use")
  -D, --debug              Enable debug mode
  -H, --host list          Daemon socket(s) to connect to
  -l, --log-level string   Set the logging level ("debug"|"info"|"warn"|"error"|"fatal") (default "info")
      --tls                Use TLS; implied by --tlsverify
      --tlscacert string   Trust certs signed only by this CA (default "/root/.docker/ca.pem")
      --tlscert string     Path to TLS certificate file (default "/root/.docker/cert.pem")
      --tlskey string      Path to TLS key file (default "/root/.docker/key.pem")
      --tlsverify          Use TLS and verify the remote
  -v, --version            Print version information and quit

Management Commands:
  builder     Manage builds
  config      Manage Docker configs
  container   Manage containers
  context     Manage contexts
  engine      Manage the docker engine
  image       Manage images
  network     Manage networks
  node        Manage Swarm nodes
  plugin      Manage plugins
  secret      Manage Docker secrets
  service     Manage services
  stack       Manage Docker stacks
  swarm       Manage Swarm
```

Docker basic commands

docker <command> <sub-command> docker image ls

docker image --help

```
root@k-node02:/home/osboxes# docker image --help

Usage:  docker image COMMAND

Manage images

Commands:
  build       Build an image from a Dockerfile
  history     Show the history of an image
  import      Import the contents from a tarball to create a filesystem image
  inspect     Display detailed information on one or more images
  load        Load an image from a tar archive or STDIN
  ls          List images
  prune       Remove unused images
  pull        Pull an image or a repository from a registry
  push        Push an image or a repository to a registry
  rm          Remove one or more images
  save        Save one or more images to a tar archive (streamed to STDOUT by default)
  tag         Create a tag TARGET_IMAGE that refers to SOURCE_IMAGE
```

Docker Container basics

Running a container

docker container run nginx

What happens in 'docker container run'

1. Docker looks for that image locally in image cache
2. If it doesn't find anything, it looks in remote image repository
3. Downloads the latest version (nginx:latest by default)
4. Creates new container based on that image and prepares to start
5. Container gets attached to a network and gets a virtual IP inside the private network, typically default bridge network
6. Opens up ports to serve the requests
7. Starts process mentioned in the CMD of image's Dockerfile

NGINX is open source software for web serving, reverse proxying, caching, load balancing, media streaming, and more

Docker Container basics

docker run nginx

```
root@docker-master:/home/osboxes# docker run nginx
Unable to find image 'nginx:latest' locally
latest: Pulling from library/nginx
54fec2fa59d0: Pull complete
4ede6f09aefe: Pull complete
f9dc69acb465: Pull complete
Digest: sha256:404ed8de56dd47adadadf9e2641b1ba6ad5c
Status: Downloaded newer image for nginx:latest
```

- When we run nginx, a container is created and it runs in the foreground and terminal is attached to its process
- When we exit out of the terminal, the container will be killed
- To avoid this we need to run the container in detached(background) mode using --detach

  docker run --detach nginx

Docker Container basics

docker ps list all running containers

```
root@docker-master:/home/osboxes# docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS               NAMES
fe55958cc7b3        nginx               "nginx -g 'daemon of…"   42 seconds ago      Up 39 seconds       80/tcp              jovial_borg
root@docker-master:/home/osboxes#
```

docker ps –a list all running and exited containers

docker container inspect <container-id> - gives the config and meta data used to start this container; returns JSON array

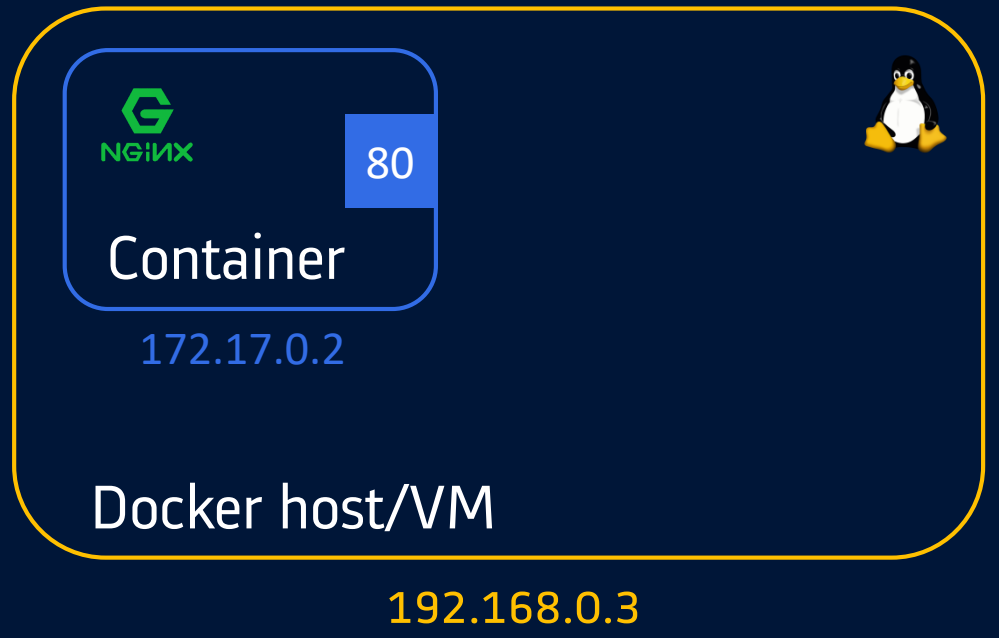
```
root@docker-master:/home/osboxes# docker inspect fe55958cc7b3
[
    {
        "Id": "fe55958cc7b3e6aa20c907e2047ea7ce503bd6a467649da0a303
        "Created": "2020-05-15T04:31:42.68932019Z",
        "Path": "nginx",
        "Args": [
            "-g",
            "daemon off;"
        ],
        "State": {
            "Status": "running",
            "Running": true,
            "Paused": false,
            "Restarting": false,
            "OOMKilled": false,
            "Dead": false,
            "Pid": 1910,
            "ExitCode": 0,
            "Error": "",
```

```
"Gateway": "172.17.0.1",
"GlobalIPv6Address": "",
"GlobalIPv6PrefixLen": 0,
"IPAddress": "172.17.0.2",
"IPPrefixLen": 16,
"IPv6Gateway": "",
"MacAddress": "02:42:ac:11:00:02",
"Networks": {
    "bridge": {
        "IPAMConfig": null,
        "Links": null,
        "Aliases": null,
        "NetworkID": "1f61a79805d0f5697c7b12861b5f0e27a811c656fb85be
        "EndpointID": "0d3d69306e169e8feee1b9031b8b49b5e8f07a7b710c4
        "Gateway": "172.17.0.1",
        "IPAddress": "172.17.0.2",
        "IPPrefixLen": 16,
        "IPv6Gateway": "",
        "GlobalIPv6Address": "",
        "GlobalIPv6PrefixLen": 0,
        "MacAddress": "02:42:ac:11:00:02",
        "DriverOpts": null
```

Docker Container basics

- curl <ip-of-container>

```
root@docker-master:/home/osboxes# curl 172.17.0.2:80
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully
working. Further configuration is required.</p>
```



You can also find IP by running below command

root@docker-master:/home/osboxes# docker container inspect --format '{{.NetworkSettings.IPAddress}}' fe559

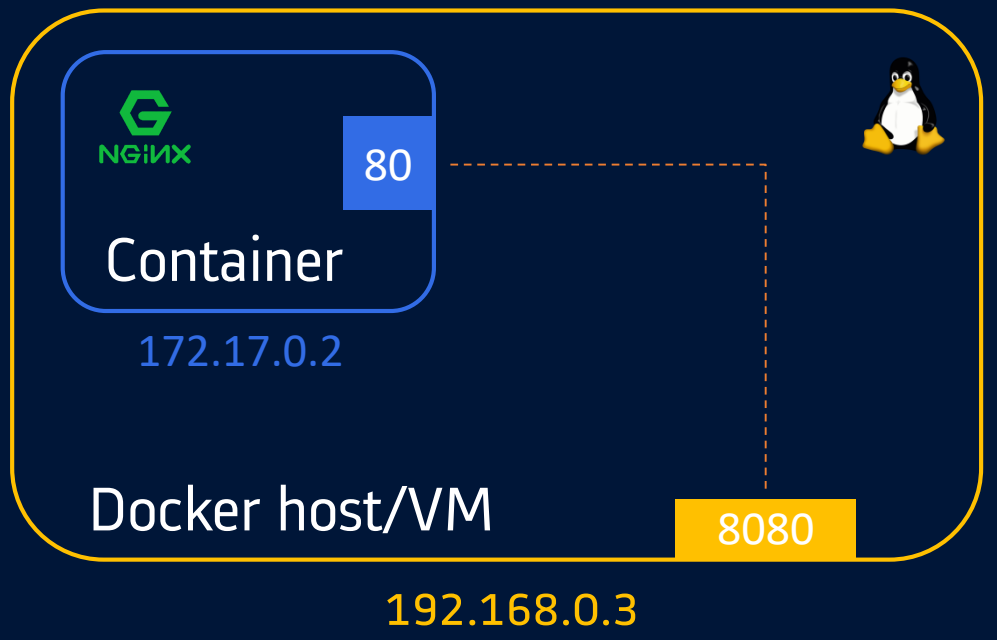
172.17.0.2

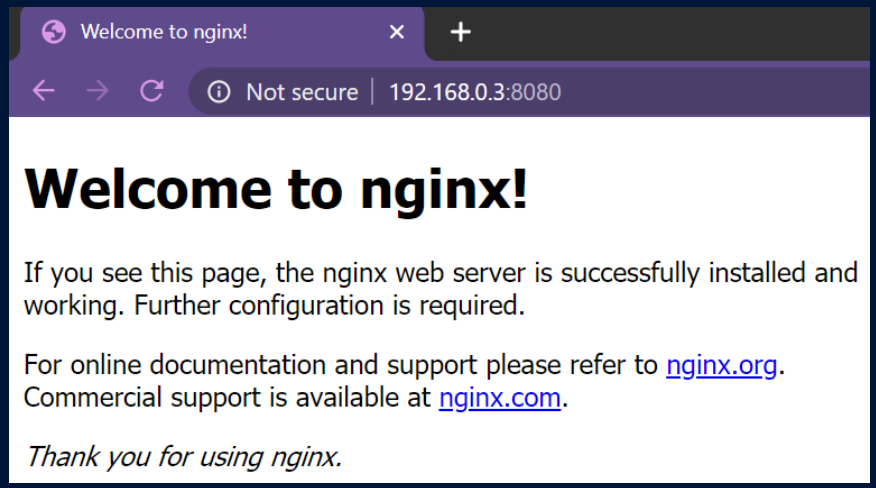
Docker Container basics: Port Mapping

- curl <ip-of-container>

```
root@docker-master:/home/osboxes# curl 172.17.0.2:80
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully
working. Further configuration is required.</p>
```

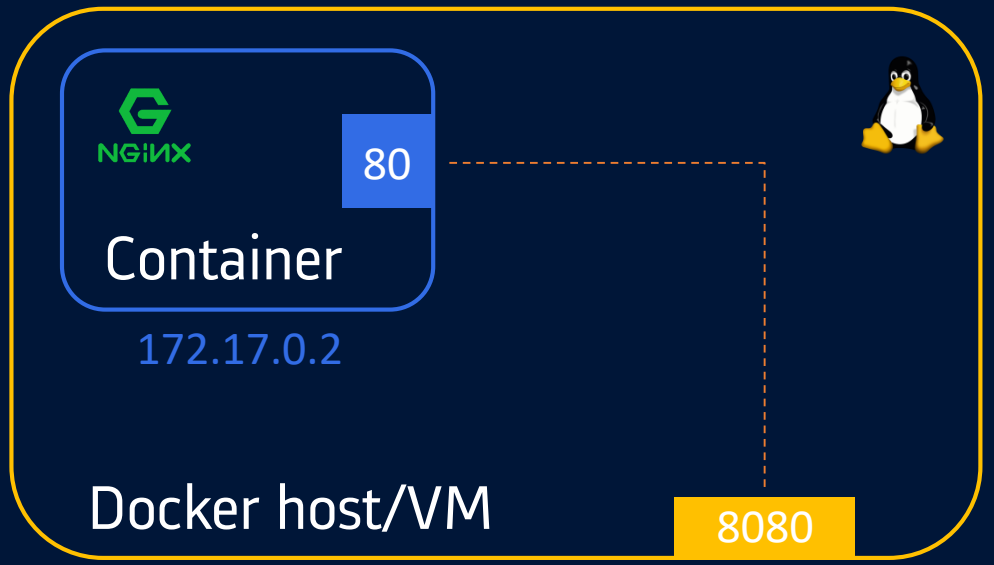
docker run –detach –p 8080:80 nginx





Docker Container

- Naming a container: docker run --detach –p 8080:80 --name webhost nginx
- Stop a container: docker container stop <unique container id>
- Start a stopped container: docker start < unique container id>
- Kill a container: docker container kill <unique container id>
- Logs: docker container logs <container name>
  docker container logs -f <container name>
- Remove a container: docker container rm <container name>
- Lists specific processes in a specific container
  docker top <container>
- Get CPU, Mem usage of the container
  docker container stats <container>



Docker Container: Getting shell access

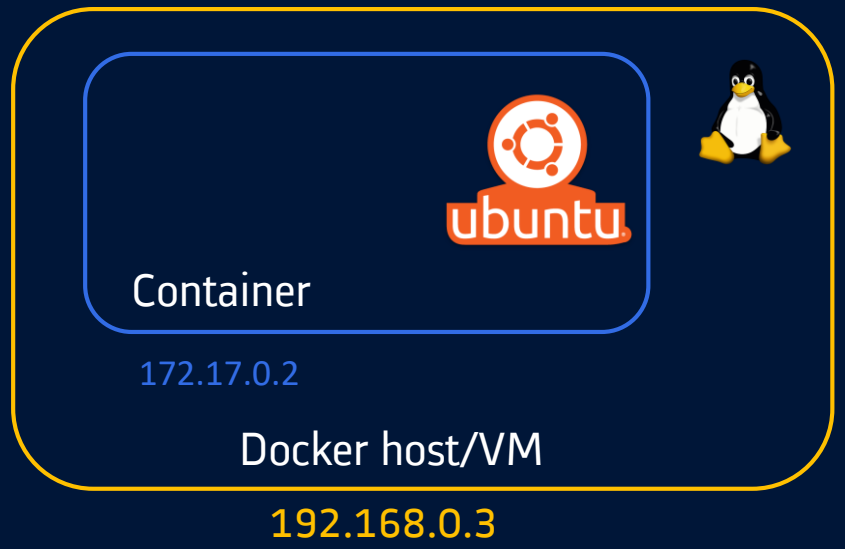
- Start new container interactively: Getting container's shell access

  docker container run -it <container-name> bash

  docker container run -it --name ubuntu bash

  -i : interactive or STD_IN

  -t : terminal or STD_OUT

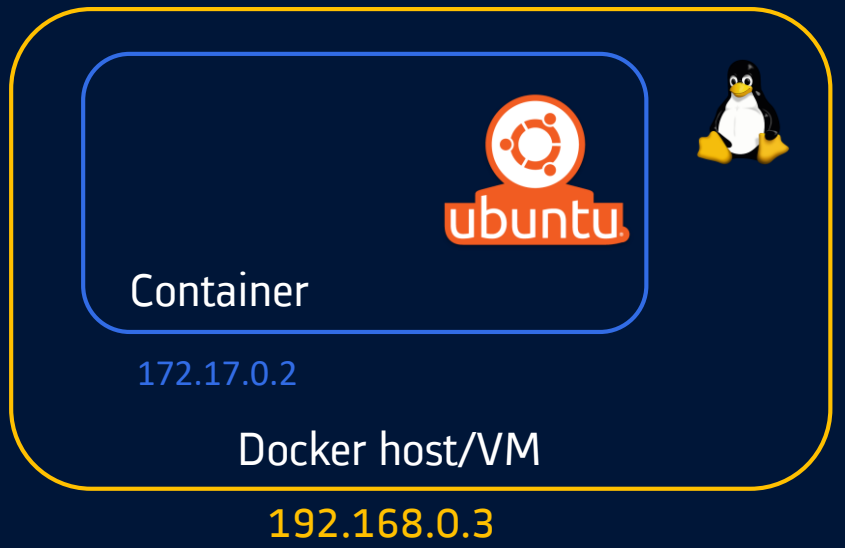



```
root@docker-master:/home/osboxes# docker run -it ubuntu bash
root@da8d3a230533:/# uname -a
Linux da8d3a230533 5.0.0-23-generic #24~18.04.1-Ubuntu SMP Mon Jul 29 16:12:28 UTC 2019 x86_64 x86_64 x86_64 GNU/Linux
root@da8d3a230533:/# hostname
da8d3a230533
root@da8d3a230533:/# exit
exit
root@docker-master:/home/osboxes# docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS                     PORTS
da8d3a230533        ubuntu              "bash"              14 seconds ago      Exited (0) 4 seconds ago
root@docker-master:/home/osboxes#
```

Docker Container: Getting shell access

Importance of -i -t flags

Any image that has shell as its starting process expects a terminal(-t) and standard input(-i) to be attached when starting the container.
If the container doesn't find the terminal, it simply exits.




```
root@proxyserver:/home/osboxes# docker run ubuntu
Unable to find image 'ubuntu:latest' locally
latest: Pulling from library/ubuntu
d51af753c3d3: Pull complete
fc878cd0a91c: Pull complete
6154df8ff988: Pull complete
fee5db0ff82f: Pull complete
Digest: sha256:747d2dbbaaee995098c9792d99bd333c6783ce56150d1b11e333bbceed5c54d7
Status: Downloaded newer image for ubuntu:latest
root@proxyserver:/home/osboxes# docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS                      PORTS
77157250b32f        ubuntu              "/bin/bash"         11 seconds ago      Exited (0) 10 seconds ago
root@proxyserver:/home/osboxes#
```

```
root@proxyserver:/home/osboxes# docker run -it ubuntu
root@837c4268bc30:/#
```

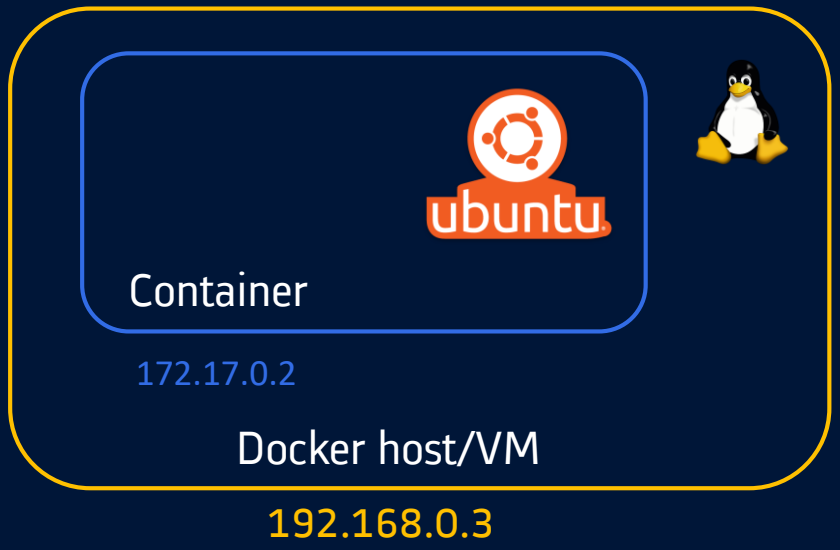
Docker Container: Getting shell access

- Containers are not full fledged operating systems
- They are meant to run a specific process/application
- Containers run as long as process inside them is alive. When process completes, the container simply exits
- Ubuntu runs bash as the starting process. When bash process is terminated, the container gets terminated.

Use image inspect to find default CMD of the image

docker image inspect ubuntu

```
root@docker-master:/home/osboxes# docker image inspect ubuntu | grep CMD
                "CMD [\"/bin/bash\"]"
root@docker-master:/home/osboxes#
```



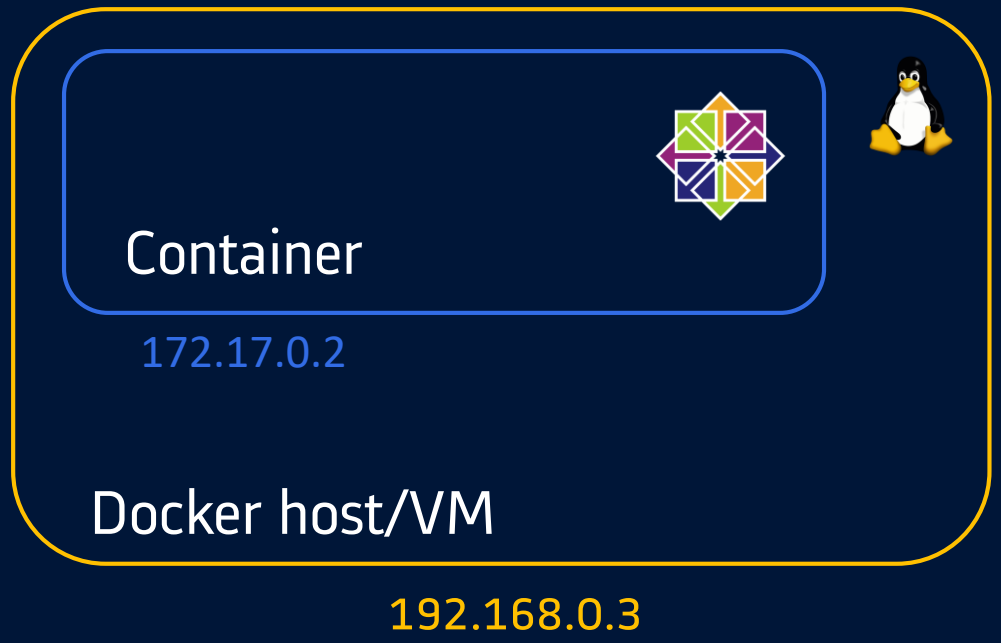
Docker Container

Interacting with the running container

- Use exec to interact with a running container
  docker run -it -d --name centos centos
  docker exec -it centos bash

Commit changes in container to a image

- By default changes made to container are lost once container is deleted, to preserve the changes, commit to an image
  docker commit <container-id> <new-image-name>



Docker Networking

- Docker gives 3 default networks: bridge, none and host
- When you start Docker, a default bridge network (also called bridge) is created automatically, and newly-started containers connect to it unless otherwise specified.

docker network ls

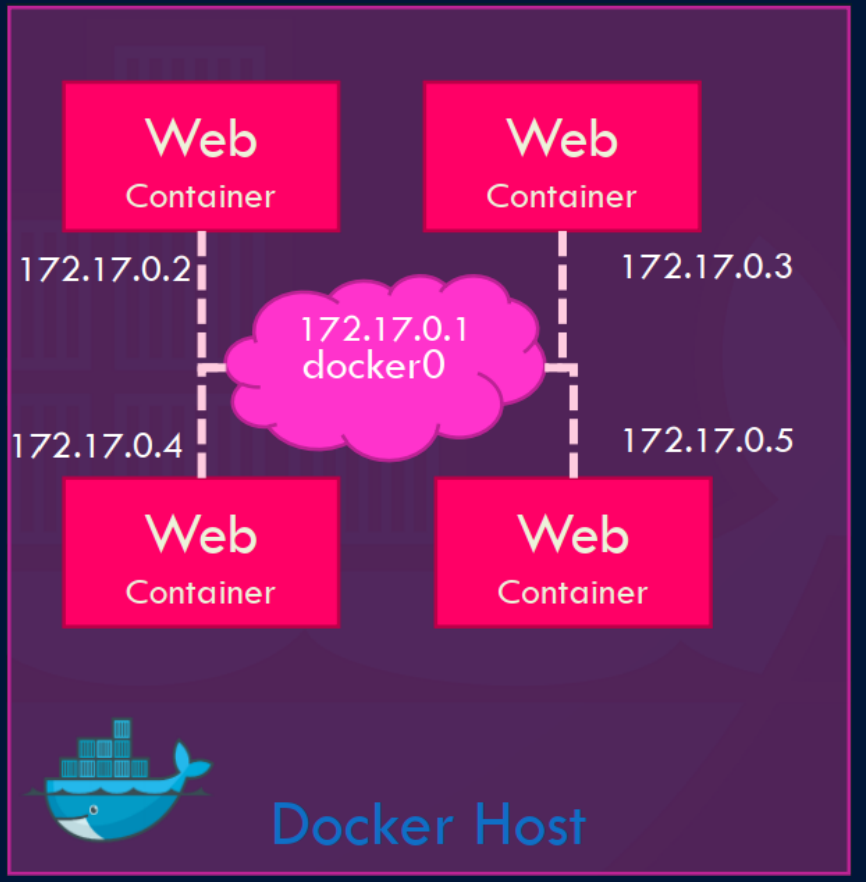
```
root@docker-master:/home/osboxes# docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
1f61a79805d0        bridge              bridge              local
5a0881810dc4        host                host                local
ba6be27e8a6d        none                null                local
```

docker network inspect bridge

```
root@docker-master:/home/osboxes# docker network inspect bridge
[
    {
        "Name": "bridge",
        "Id": "1f61a79805d0f5697c7b12861b5f0e27a811c656fb85be3cbb90f89f62090f89",
        "Created": "2020-05-15T00:31:00.229136167-04:00",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": null,
            "Config": [
                {
                    "Subnet": "172.17.0.0/16",
                    "Gateway": "172.17.0.1"
                }
            ]
        },
```

Docker Networking: Bridge

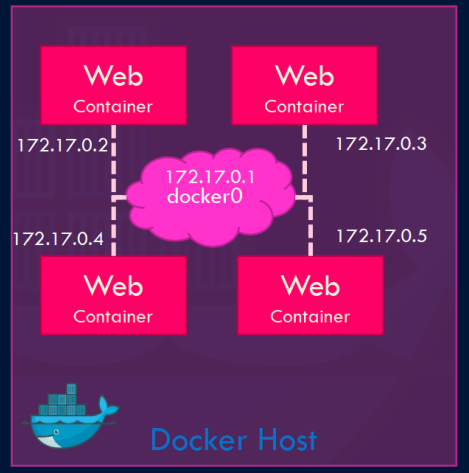
- In Bridge network, all containers get private internal IPs and they are isolated from host.
- Port forwarding forwards outside traffic to the containers.
- Containers on the default bridge network can only access each other by IP addresses, unless you use the --link option, which is considered legacy
- You can also create user-defined custom bridge network
- User-defined bridge networks are superior to the default bridge network
- On a user-defined bridge network, containers can resolve each other by name or alias(DNS)



Create a bridge network: docker network create --driver bridge my-net
Attach a container to it: docker run -d --name web --net my-net nginx

Docker Networking: Bridge

Understanding DNS resolution in bridge network



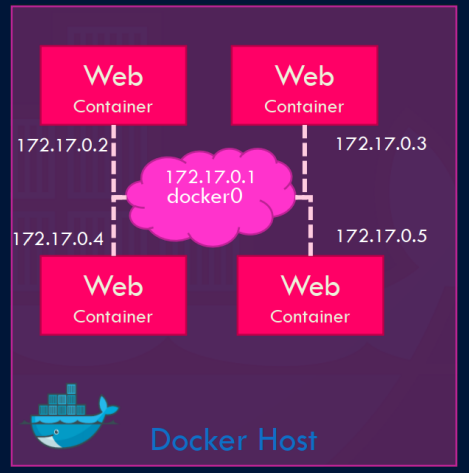
- When containers are run in default bridge network they cannot find each other using their container names.
- Simply put, DNS resolution through container names will not work under default bridge network

In the below example, 2 containers are created under the default bridge network
Note that ping to second container from the first container using the second container's name didn't resolve(DNS server is not available under default bridge network)

```
root@proxyserver:/home/osboxes# docker run -d -it --name c1 kunchalavikram/ubuntu_with_ping
af3b1f94947a31b1a6e5d01b5df078b440935324ec867f1f468db1a79ff718cf
root@proxyserver:/home/osboxes# docker run -d -it --name c2 kunchalavikram/ubuntu_with_ping
feb7ada04e5de274a5594c44ca430d5eab69e86a7cf38adf33328504b3876281
root@proxyserver:/home/osboxes# docker ps
CONTAINER ID        IMAGE                              COMMAND             CREATED             STATUS              PORTS               NAMES
feb7ada04e5d        kunchalavikram/ubuntu_with_ping    "/bin/bash"         6 seconds ago       Up 4 seconds                            c2
af3b1f94947a        kunchalavikram/ubuntu_with_ping    "/bin/bash"         12 seconds ago      Up 10 seconds                           c1
root@proxyserver:/home/osboxes# docker exec -it c1 bash
root@af3b1f94947a:/# ping c2
ping: c2: Name or service not known
root@af3b1f94947a:/#
```

Docker Networking: Bridge

Understanding DNS resolution in bridge network



- Now a new bridge network is created and containers are attached to that network using --net flag
- In this case, containers find each other using their container names(DNS resolution through container names)

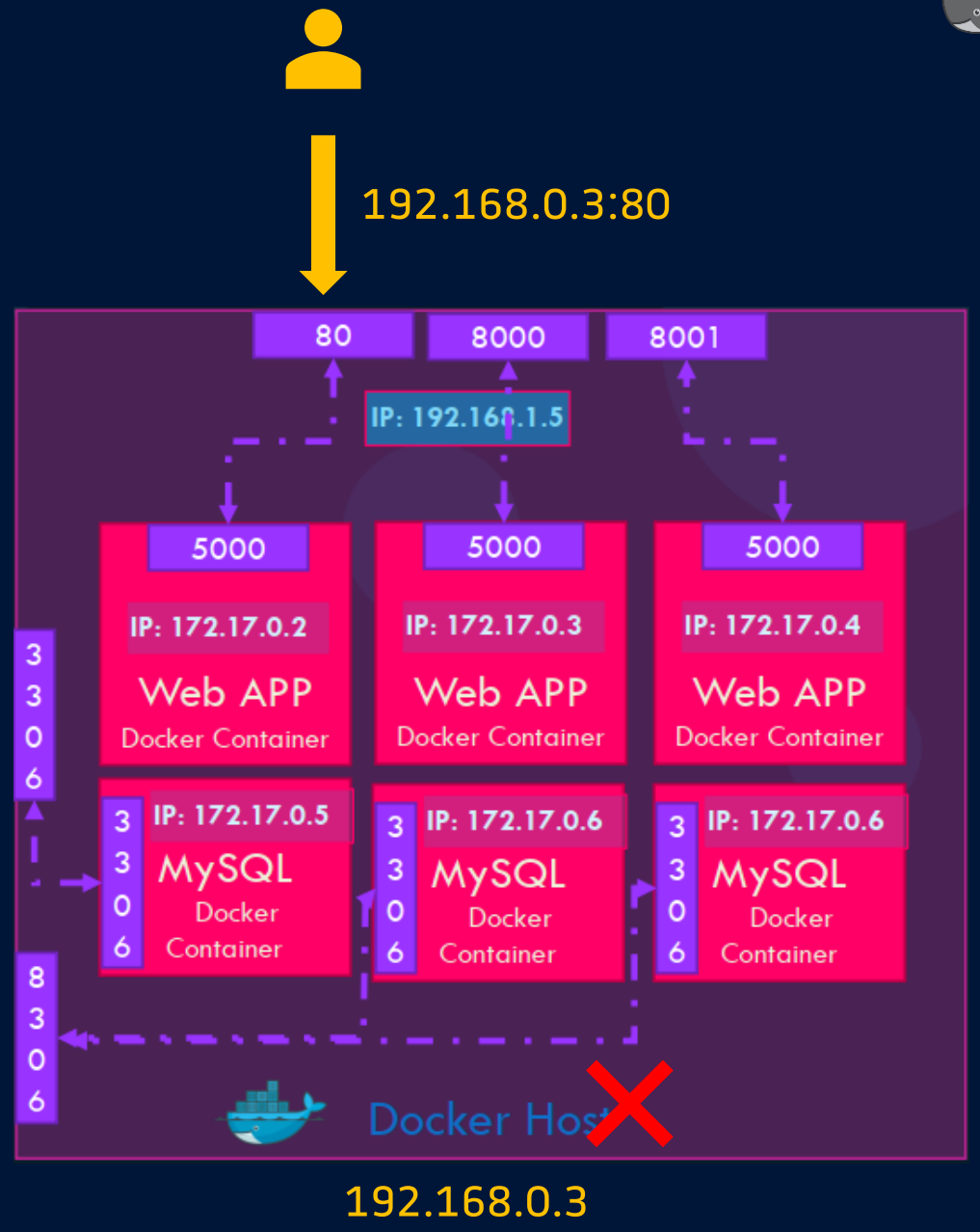
```
root@proxyserver:/home/osboxes# docker network create -d bridge my_net
cfd33cda13575304a589acbb9683e805cd49a7fa300bfd97d9641f42d59e73c3
root@proxyserver:/home/osboxes# docker run -d -it --net my_net --name c1 kunchalavikram/ubuntu_with_ping
51162f205336f3a385a0471beb093a01af014dd331d6f46b9d9ffc61cf231fa9
root@proxyserver:/home/osboxes# docker run -d -it --net my_net --name c2 kunchalavikram/ubuntu_with_ping
6b14069ee4c2af2ec1c649d54ec34a4f992cfa6424e9b483da4f7460e2ae67d9
root@proxyserver:/home/osboxes# docker ps
CONTAINER ID        IMAGE                              COMMAND             CREATED             STATUS
6b14069ee4c2        kunchalavikram/ubuntu_with_ping    "/bin/bash"         6 seconds ago       Up 5 seconds
51162f205336        kunchalavikram/ubuntu_with_ping    "/bin/bash"         12 seconds ago      Up 11 seconds
root@proxyserver:/home/osboxes# docker exec -it c1 bash
root@51162f205336:/# ping c2
PING c2 (172.18.0.3) 56(84) bytes of data.
64 bytes from c2.my_net (172.18.0.3): icmp_seq=1 ttl=64 time=0.052 ms
64 bytes from c2.my_net (172.18.0.3): icmp_seq=2 ttl=64 time=0.049 ms
64 bytes from c2.my_net (172.18.0.3): icmp_seq=3 ttl=64 time=0.049 ms
64 bytes from c2.my_net (172.18.0.3): icmp_seq=4 ttl=64 time=0.048 ms
^C
--- c2 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3068ms
rtt min/avg/max/mdev = 0.048/0.049/0.052/0.001 ms
root@51162f205336:/#
```

Docker Networking: Port Mapping

Port forwarding enables access to applications running inside containers from outside world

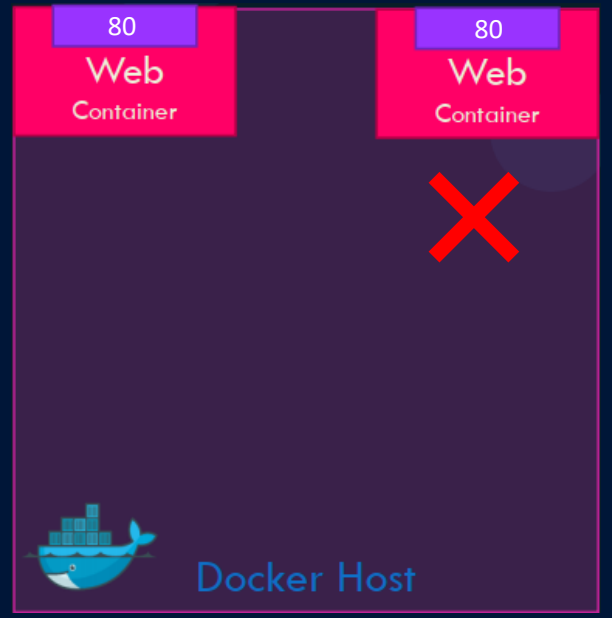
```
docker run –p 80:5000 nginx
docker run –p 8000:5000 nginx
docker run –p 8001:5000 nginx
docker run –p 3306:3306 mysql
docker run –p 8306:3306 mysql
docker run -p 8306:3306 mysql
```

Port is already allocated



Docker Networking: Host

- In host network, all containers directly get connected to host.
- Multiple containers cannot run on same hosts because of port conflicts on host side

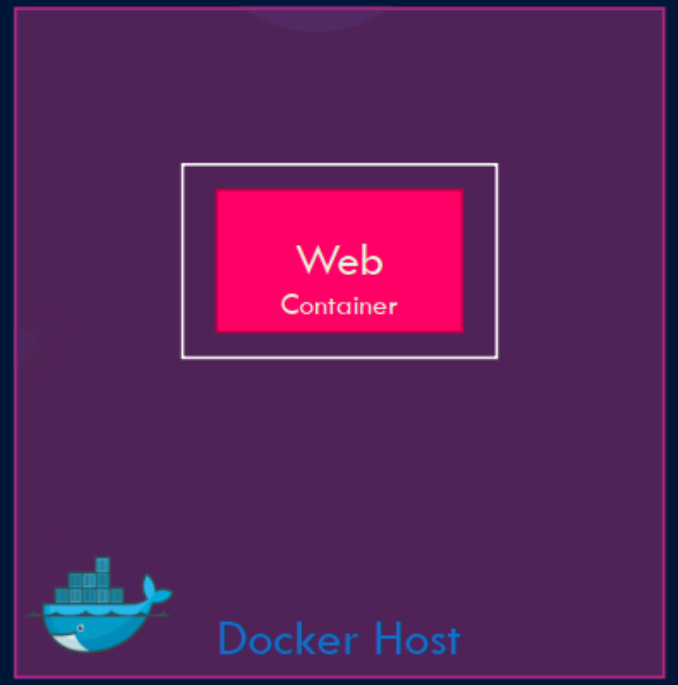


192.168.0.101

docker run -d --name web -net host nginx

Docker Networking: None

- This offers a container-specific network stack that lacks a network interface.
- Containers run in pure isolation
- This container only has a local loopback interface (i.e., no external network interface)



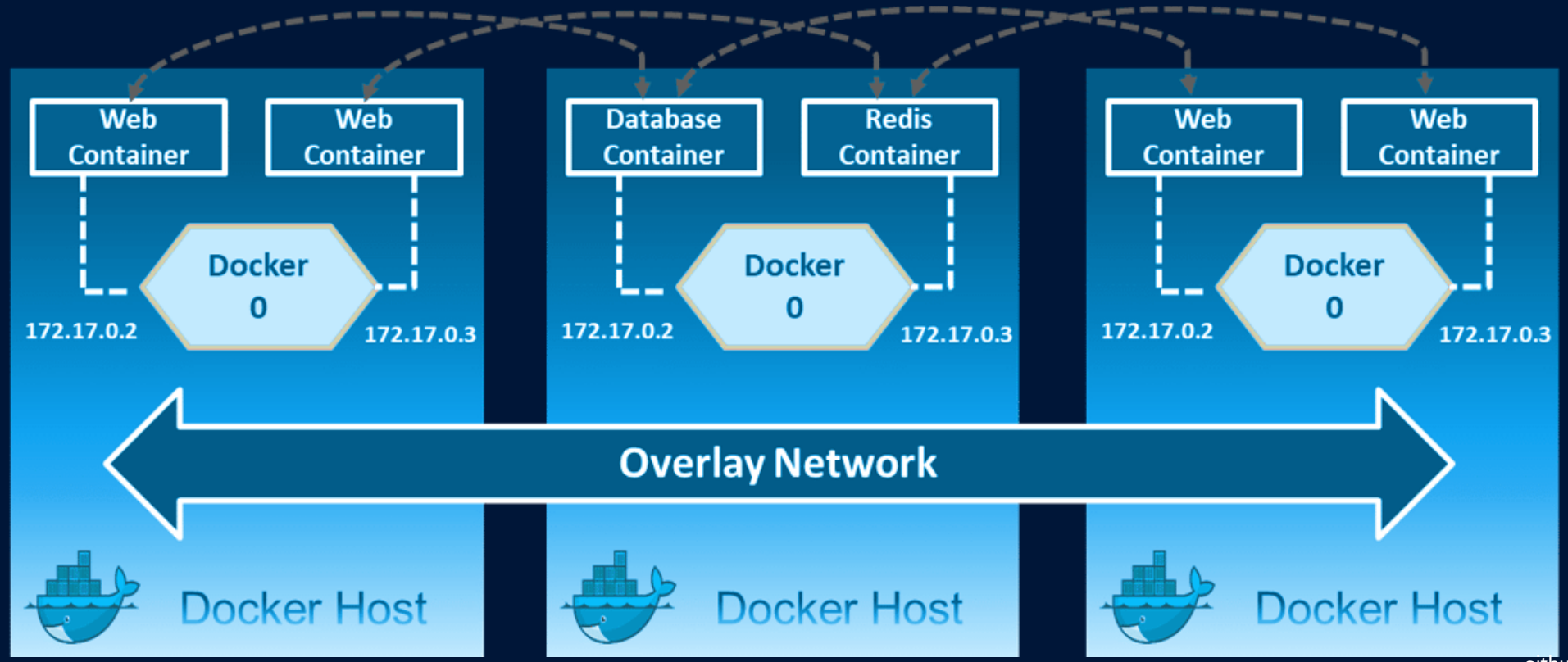
docker run -d --name web --net none nginx

```
root@docker-master:/home/osboxes# docker run -d --name web --net none nginx
f5bf2e6ea26c9e6d826b15951ba10d72dac1ab67ac88201a694861fa21ecd65e
root@docker-master:/home/osboxes# docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS
f5bf2e6ea26c        nginx               "nginx -g 'daemon of…"   6 seconds ago       Up 5 seconds
root@docker-master:/home/osboxes#
```

```
root@docker-master:/home/osboxes# docker container inspect web | grep IPAddress
            "SecondaryIPAddresses": null,
            "IPAddress": "",
                    "IPAddress": "",
root@docker-master:/home/osboxes#
```

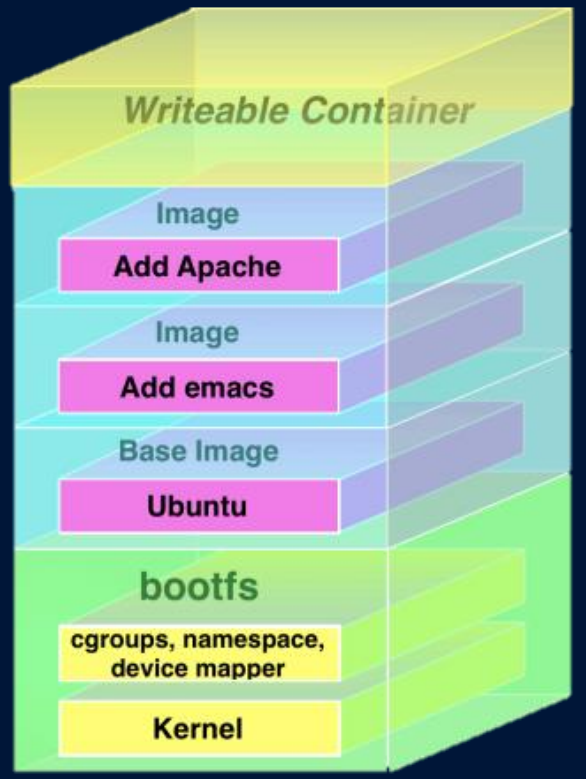
Docker Networking: Overlay

- Bridge networks apply to containers running on the same Docker daemon host. For communication among containers running on different Docker daemon hosts, we should use an overlay network which spans across the entire cluster



Docker Volumes

- Images are a series of read-only layers
- A container is merely an instantiation of those read-only layers with a single read-write layer on top.
- Any file changes that are made within a container are reflected as a copy of modified data from the read-only layer.
- The version in the read-write layer hides the underlying file but does not remove it.
- When deleting a container, the read-write layer containing the changes are destroyed and gone forever!
- In order to persist these changes we use docker volumes



Advantages:

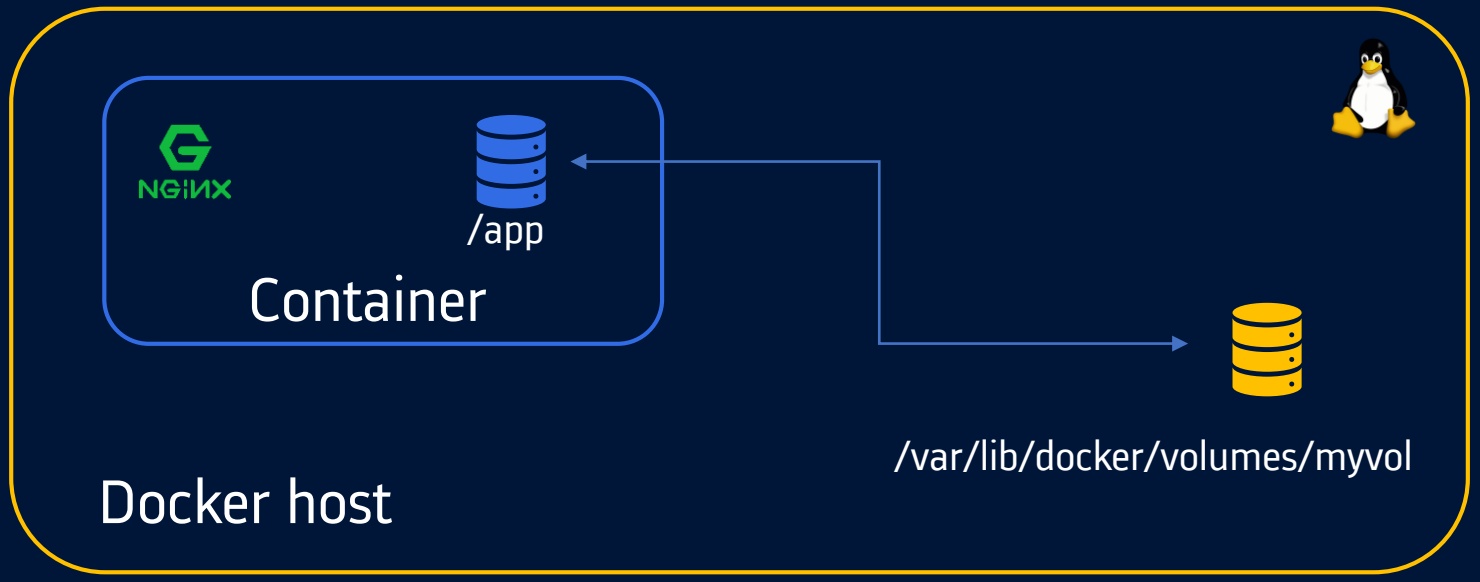
1. To keep data around when a container is removed
2. To share data between the host filesystem and the Docker container

Docker Volumes

Two types of volume mounts: Named and Bind

Named Volume: Mounting a volume created using 'docker volume create' command and mounting it from default volume location /var/lib/docker/volumes

```
docker volume create my-vol
docker run -d  --name nginx  -v myvol:/app  nginx
docker run -d  --name nginx  --mount source=myvol2,target=/app  nginx
```



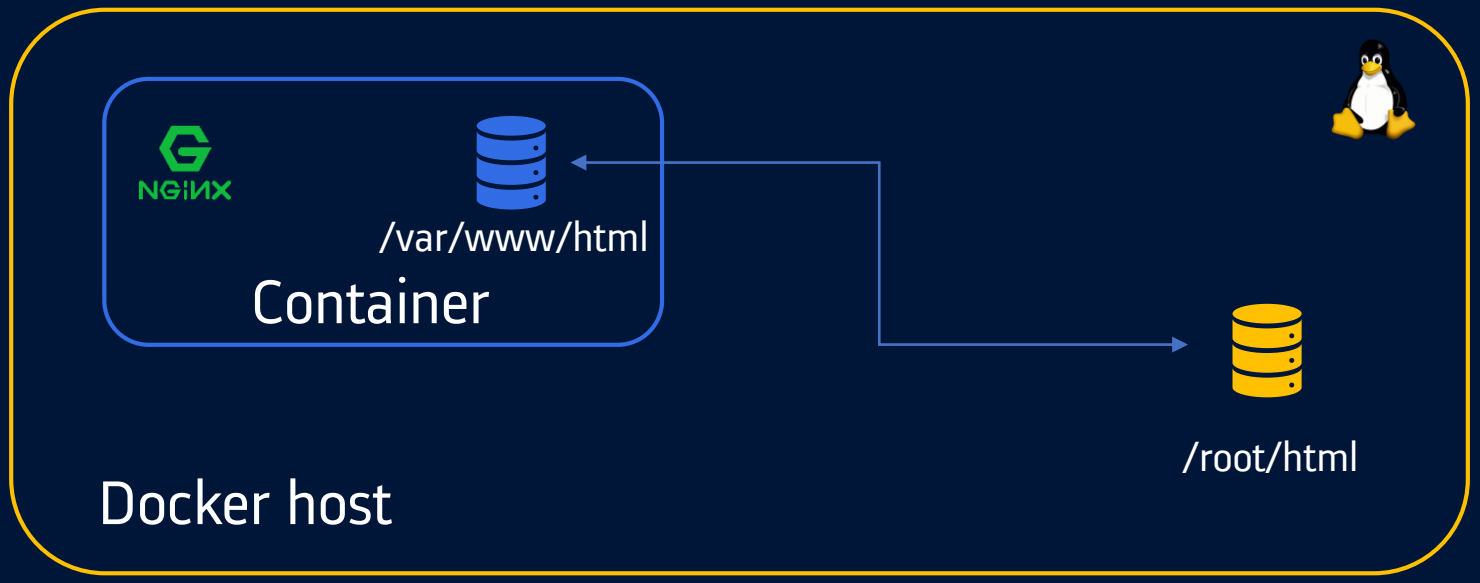
Docker Volumes

Two types of volume mounts: **Named** and **Bind**

Bind Volume: External mounting(external hard disks etc.)

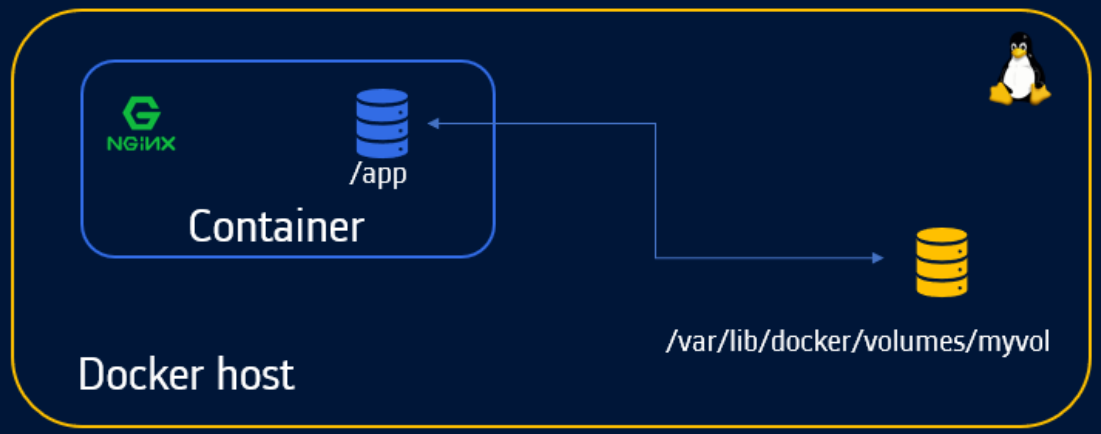
Bind mounts may be stored anywhere on the host system. They usually start with '/'

docker run –name web -v /root/html:/var/www/html/ nginx



Docker volumes commands

docker volume create <volume_name>
docker volume ls
docker volume inspect <volume_name>
docker volume rm <volume_name>
docker volume prune



Named Volume

Bind Volume

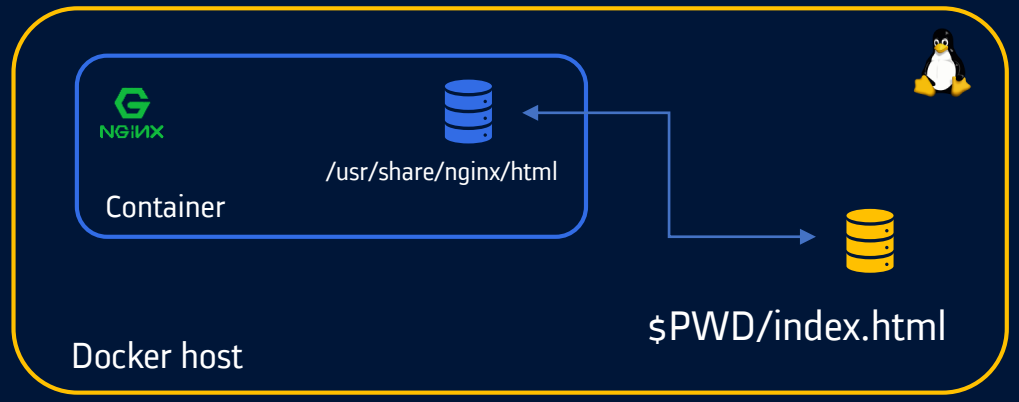
Docker Volumes

Demo: Hosting a static website using nginx

```
docker run -d --name web –p 80:80 nginx
```

```
docker exec -it web bash
root@768faf801706:/#  ls /usr/share/nginx/html
50x.html  index.html
```


```
docker run -d --name web -p 80:80 -v $PWD:/usr/share/nginx/html nginx
```



Building Images

- Hosting a static website was easier as base image nginx has all dependencies to host.
- What if we are required to host an application which requires lot of dependencies which base image do not provide? We build the image!!!

Let's say a flask application has to be built

```
docker run –it  --name my-app ubuntu bash
root@beb164a51e6a:/#  apt update
root@beb164a51e6a:/#  apt install python
root@beb164a51e6a:/#  pip install flask
root@beb164a51e6a:/#  exit
```

```
docker commit my-app my-flask-app
```

container-name image-name

Dockerfile

- Dockerfile is essentially the build instructions to build your image
- It is a text document that contains all the commands a user could call on the command line to assemble the image
- Using docker build users can create an automated build that executes several command-line instructions in succession

Name of the file is Dockerfile without any extensions

Dockerfile contents

- FROM defines the base image used to start the build process
- MAINTAINER defines a full name and email address of the image creator
- COPY copies one or more files from the Docker host into the Docker image
- EXPOSE exposes a specific port to enable networking between the container and the outside world
- RUN runs commands while image is being built from the dockerfile and saves result as a new layer
- VOLUME is used to enable access from the container to a directory on the host machine
- WORKDIR used to set default working directory for the container
- CMD command that runs when the container starts
- ENTRYPOINT command that runs when the container starts

http://tutorials.jenkov.com/docker/dockerfile.html

Dockerfile

```
FROM python:alpine3.7
COPY . /app
WORKDIR /app
RUN pip install flask
EXPOSE 5000
CMD python ./app.py
```

(or)

```
FROM ubuntu
RUN apt-get update
RUN apt-get install -y python python-pip wget
RUN pip install Flask
COPY app.py /home/app.py
WORKDIR /home
CMD python app.py
```

Build the image:

```
docker build  -t flaskapp .
```

(make sure you are in the directory of docker file and Dockerfile has no name extensions)

-t – Tag the image with a name

. – Dot indicates look for Dockerfile from PWD/Present Working Directory

Source:
https://github.com/kunchalavikram1427/Docker_public/tree/master/Examples/Dockerfile-flask

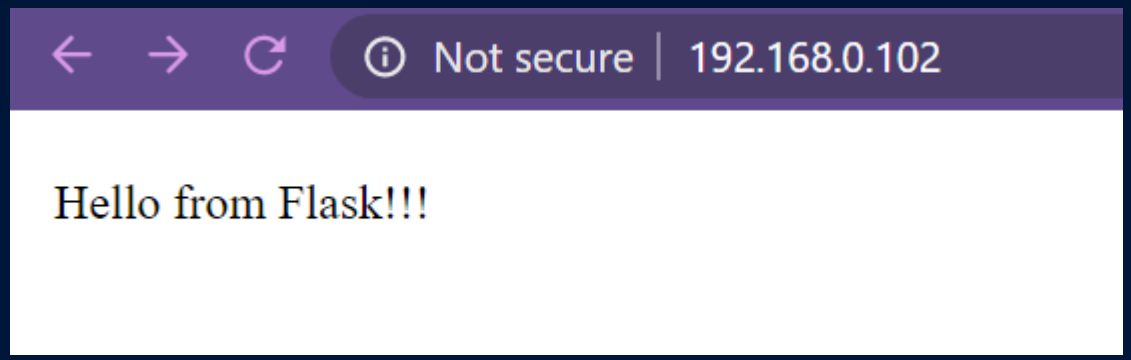
Dockerfile

docker images

```
root@docker-master:/home/osboxes/docker/Dockerfile-flask# docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
flaskapp            latest              afa14e17a2ed        4 seconds ago       91.6MB
nginx               latest              9beeba249f3e        4 days ago          127MB
python              alpine3.7           00be2573e9f7        15 months ago       81.3MB
```

```
FROM python:alpine3.7
COPY . /app
WORKDIR /app
RUN pip install flask
EXPOSE 5000
CMD python ./app.py
```

docker run -d --name flask -p 80:5000 flaskapp



Flask listens on port 5000 by default. You can also configure it to a different port

Dockerfile

Demo: Building a single landing page dynamic website using flask

flask_app

- Dockerfile
- appv3.py
- static/image.jpg
- templates/index.html

index.html file should in templates directory and image should be in static directory

Dockerfile

```
FROM python:alpine3.7
COPY . /app
WORKDIR /app
RUN pip install flask
EXPOSE 5000
CMD python ./appv3.py
```

appv3.py

```
import socket
from flask import Flask,request, render_template

def getIP():
    hostname = socket.gethostname()
    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    s.connect(("8.8.8.8", 80))
    ip = s.getsockname()[0]
    print(ip)
    s.close()
    return str(hostname),str(ip)
app = Flask(__name__)
@app.route("/")
def hello():
    hostname,ip = getIP()
    return render_template('index.html',hostname=hostname,ip=ip)
if __name__ == "__main__":
    app.run(host="0.0.0.0", port=int("5000"), debug=True)
```

We will pass these variables to index.html file

Source: https://github.com/kunchalavikram1427/Docker_public/tree/master/Examples/Dockerfile-flask-example

Dockerfile

Demo: Building a single landing page dynamic website using flask

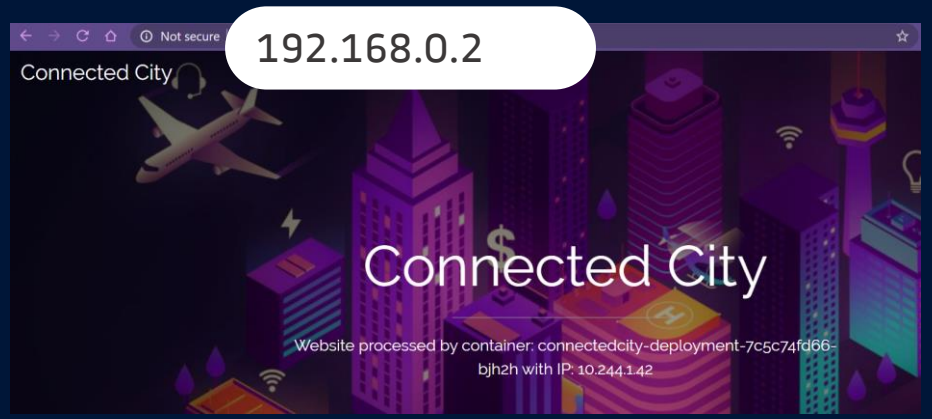
Build Image and Push to docker hub

```
docker build –t <username>/flask-app .
docker push <username>/flask-app
```

<username> - Your docker hub account name

Run the container

```
docker run -d --name flaskapp -p 80:5000 <user-name>/flask-app
```



index.html

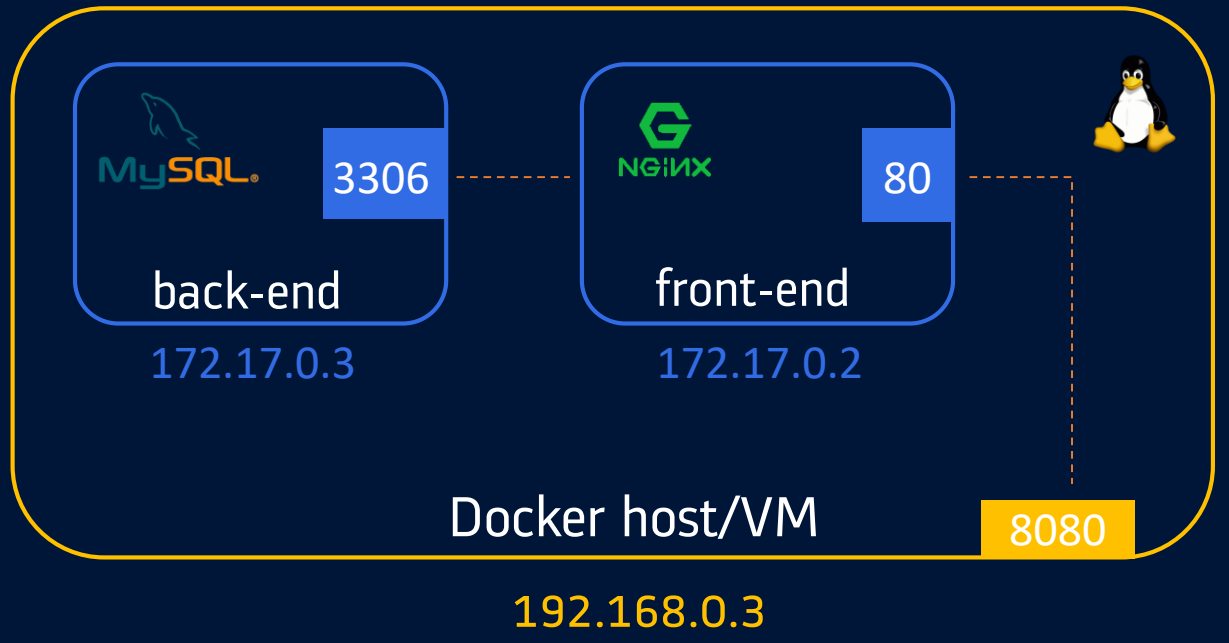
```
<!DOCTYPE html>
<html>
<title>Smart City</title>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1">
<link rel="stylesheet" href="https://www.w3schools.com/w3css/4/w3.css">
<link rel="stylesheet" href="https://fonts.googleapis.com/css?family=Raleway">
<style>
body,h1 {font-family: "Raleway", sans-serif}
body, html {height: 100%}
.bgimg {
  background-image: url('/static/smartcity.jpg');
  min-height: 100%;
  background-position: center;
  background-size: cover;
}
</style>
<body>

<div class="bgimg w3-display-container w3-animate-opacity w3-text-white">
  <div class="w3-display-topleft w3-padding-large w3-xlarge">
    Connected City
  </div>
  <div class="w3-display-middle">
    <h1 class="w3-jumbo w3-animate-top" style="text-align:center"> Connected City </h1>
    <hr class="w3-border-grey" style="margin:auto;width:40%">
    <p class="w3-large w3-center"> Website processed by container: {{ hostname }} with IP: {{ ip }} </p>
  </div>
  <div class="w3-display-bottomleft w3-padding-large">
    Powered by <a href="https://github.com/kunchalavikram1427" target="_blank">Github</a>
  </div>
</div>

</body>
</html>
```

Docker Compose

- Docker Compose is used to run multiple containers as a single service.
- For example, an application requires both NGNIX and MySQL containers, you could create one file which would start both the containers as a service(docker compose) or start each one separately(docker run)
- All services are to be defined in YAML format

compose file: docker-compose.yml

Bring up the app: docker-compose up –d
Bring down the app: docker-compose down

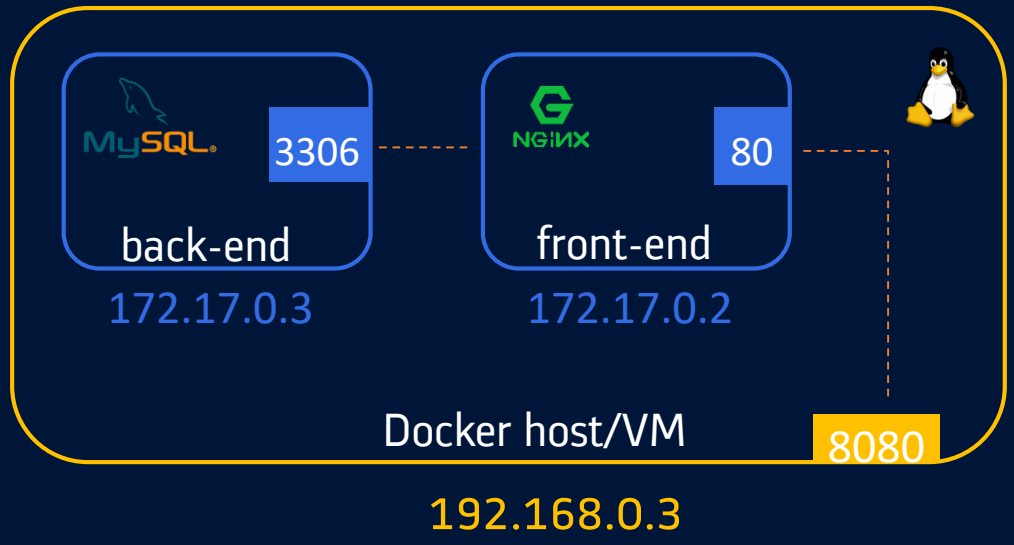


https://takacsmark.com/docker-compose-tutorial-beginners-by-example-basics/

Docker Compose

Compose file syntax

```
version: '3'  # if no version is specified then v1 is assumed.
services:  # containers. same as docker run
   service_name: # container name. this is also DNS name inside network
      image: # name of the image
      command: # Optional, replace the default CMD specified by the image
      environment: # same as -e in docker run
      ports: # same as –p in docker run
      volumes: # same as -v in docker run
   service_name2:

volumes: # Optional, same as docker volume create

networks: # Optional, same as docker network create
```



Docker Compose versions

Version 1

- Compose files that do not declare a version are considered “version 1”
- Do not support named volumes, user-defined networks or build arguments
- Every container is placed on the default bridge network and is reachable from every other container at its IP address. You need to use links to enable discovery between containers
- No DNS resolution using container names

Version 2

- Links are deprecated. DNS resolution through container names
- All services must be declared under the ‘services’ key
- Named volumes can be declared under the volumes key, and networks can be declared under the networks key
- New bridge network to connect all containers

Version 3

- Support for docker swarm

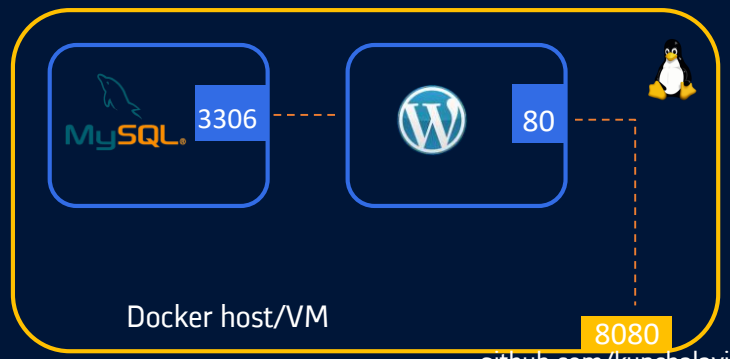
Docker Compose

```yaml
version: '3.3'
services:
  wordpress:
    image: wordpress
    depends_on:
      - mysql
    ports:
      - 8080:80
    environment:
      WORDPRESS_DB_HOST: mysql
      WORDPRESS_DB_NAME: wordpress
      WORDPRESS_DB_USER: wordpress
      WORDPRESS_DB_PASSWORD: wordpress
    volumes:
      - ./wordpress-data:/var/www/html
    networks:
      - my_net
  mysql:
    image: mariadb
    environment:
      MYSQL_ROOT_PASSWORD: wordpress
      MYSQL_DATABASE: wordpress
      MYSQL_USER: wordpress
      MYSQL_PASSWORD: wordpress
    volumes:
      - mysql-data:/var/lib/mysql
    networks:
      - my_net
volumes:
  mysql-data:
networks:
  my_net:
```

```yaml
wordpress:
  image: wordpress
  depends_on:
    - mysql
  ports:
    - 8080:80
  environment:
    WORDPRESS_DB_HOST: mysql
    WORDPRESS_DB_NAME: wordpress
    WORDPRESS_DB_USER: wordpress
    WORDPRESS_DB_PASSWORD: wordpress
  volumes:
    - ./wordpress-data:/var/www/html
  networks:
    - my_net
```

```yaml
mysql:
  image: mariadb
  environment:
    MYSQL_ROOT_PASSWORD: wordpress
    MYSQL_DATABASE: wordpress
    MYSQL_USER: wordpress
    MYSQL_PASSWORD: wordpress
  volumes:
    - mysql-data:/var/lib/mysql
  networks:
    - my_net
```

```yaml
volumes:
  mysql-data:
networks:
  my_net:
```

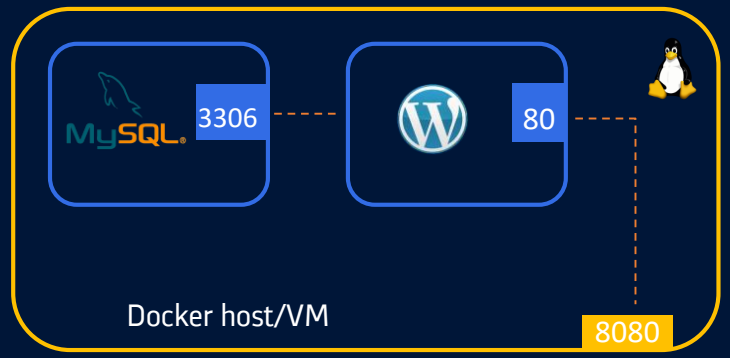


Docker Compose

```yaml
version: '3.3'
services:
  wordpress:
    image: wordpress
    depends_on:
      - mysql
    ports:
      - 8080:80
    environment:
      WORDPRESS_DB_HOST: mysql
      WORDPRESS_DB_NAME: wordpress
      WORDPRESS_DB_USER: wordpress
      WORDPRESS_DB_PASSWORD: wordpress
    volumes:
      - ./wordpress-data:/var/www/html
    networks:
      - my_net
  mysql:
    image: mariadb
    environment:
      MYSQL_ROOT_PASSWORD: wordpress
      MYSQL_DATABASE: wordpress
      MYSQL_USER: wordpress
      MYSQL_PASSWORD: wordpress
    volumes:
      - mysql-data:/var/lib/mysql
    networks:
      - my_net
volumes:
  mysql-data:
networks:
  my_net:
```

Services file: docker-compose.yml

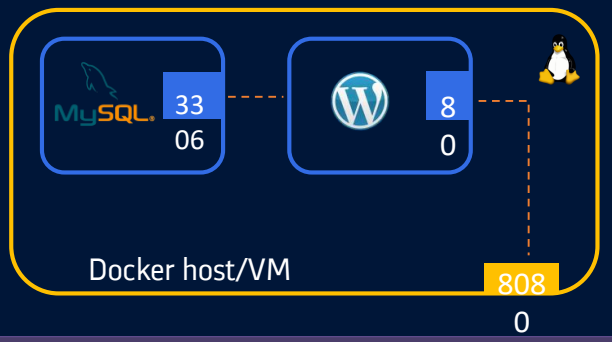
Bring up: docker-compose up –d
Bring down: docker-compose down
Process state: docker-compose ps

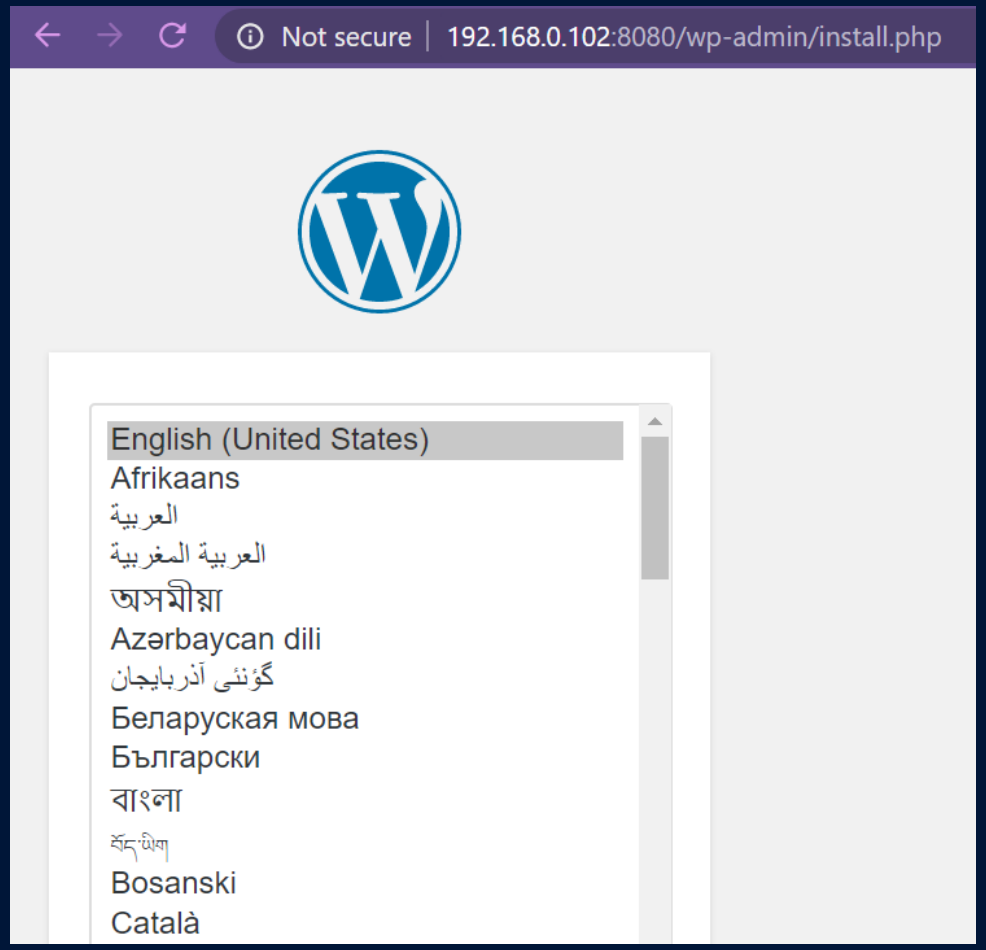


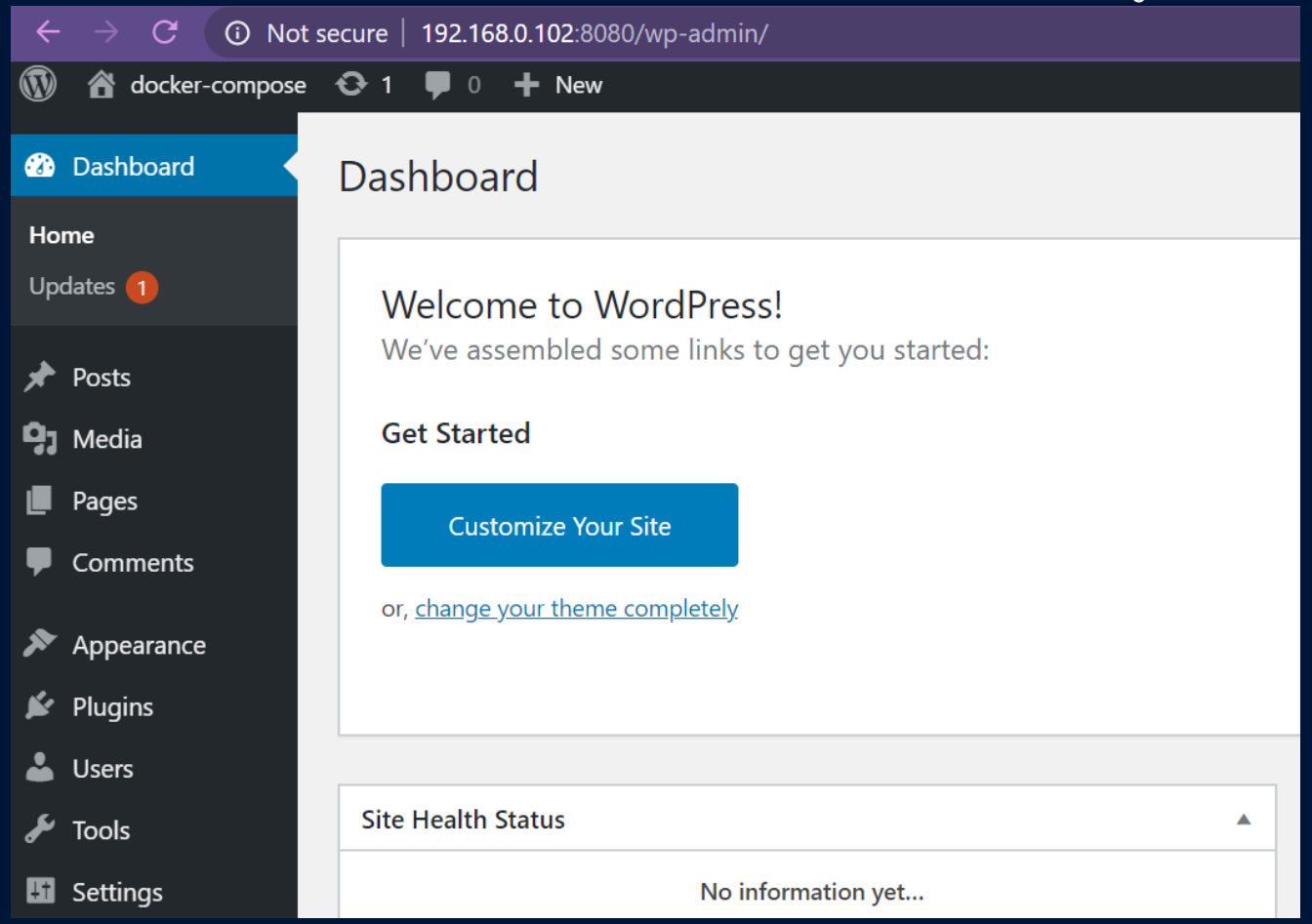

```
root@docker-master:/home/osboxes/docker# docker-compose up -d
Creating network "docker_my_net" with the default driver
Creating volume "docker_mysql-data" with default driver
Pulling mysql (mariadb:)...
latest: Pulling from library/mariadb
23884877105a: Pull complete
bc38caa0f5b9: Pull complete
2910811b6c42: Pull complete
36505266dcc6: Pull complete
e69dcc78e96e: Pull complete
222f44c5392d: Pull complete
efc64ea97b9c: Pull complete
9912a149de6b: Extracting [==================================================>]     115B/115B
```

```
root@docker-master:/home/osboxes/docker# docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS
9784a2cc6e02        wordpress           "docker-entrypoint.s…"   5 minutes ago       Up 5 minutes        0.0.0.0
2657b6db3f94        mariadb             "docker-entrypoint.s…"   5 minutes ago       Up 5 minutes        3306/tc
root@docker-master:/home/osboxes/docker#
```

Docker Compose







Docker Compose

Image build

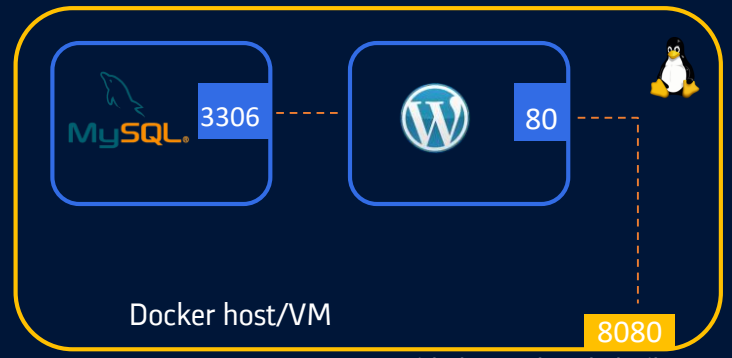
```yaml
version: '3.3'
services:
  wordpress:
    image: wordpress
    depends_on:
      - mysql
    ports:
      - 8080:80
    environment:
      WORDPRESS_DB_HOST: mysql
      WORDPRESS_DB_NAME: wordpress
      WORDPRESS_DB_USER: wordpress
      WORDPRESS_DB_PASSWORD: wordpress
    volumes:
      - ./wordpress-data:/var/www/html
    networks:
      - my_net
  mysql:
    image: mariadb
    environment:
      MYSQL_ROOT_PASSWORD: wordpress
      MYSQL_DATABASE: wordpress
      MYSQL_USER: wordpress
      MYSQL_PASSWORD: wordpress
    volumes:
      - mysql-data:/var/lib/mysql
    networks:
      - my_net
volumes:
  mysql-data:
networks:
  my_net:
```

If the image has to be built before deployment, include dockerfile in compose file

```yaml
version: "3"
services:
    wordpress:
        build:
            context: .
            dockerfile: Dockerfile-wordpress
            image: wordpress
            container_name: wordpress
```

To build only images: docker-compose build

build + deploy: docker-compose up -d

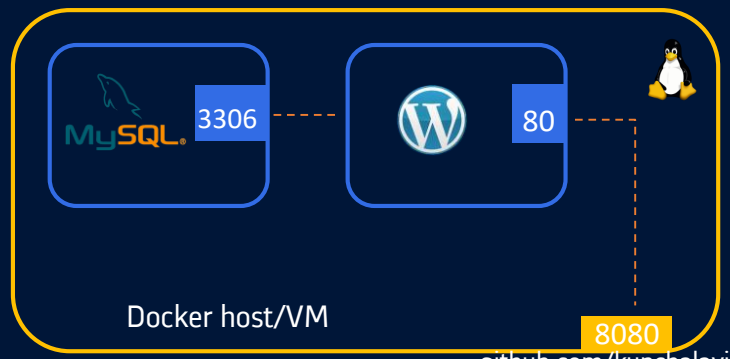


Docker Compose

```
version: '3.3'
services:
  wordpress:
    image: wordpress
    depends_on:
      - mysql
    ports:
      - 8080:80
    environment:
      WORDPRESS_DB_HOST: mysql
      WORDPRESS_DB_NAME: wordpress
      WORDPRESS_DB_USER: wordpress
      WORDPRESS_DB_PASSWORD: wordpress
    volumes:
      - ./wordpress-data:/var/www/html
    networks:
      - my_net
  mysql:
    image: mariadb
    environment:
      MYSQL_ROOT_PASSWORD: wordpress
      MYSQL_DATABASE: wordpress
      MYSQL_USER: wordpress
      MYSQL_PASSWORD: wordpress
    volumes:
      - mysql-data:/var/lib/mysql
    networks:
      - my_net
volumes:
  mysql-data:
networks:
  my_net:
```

Deployment through imperative commands

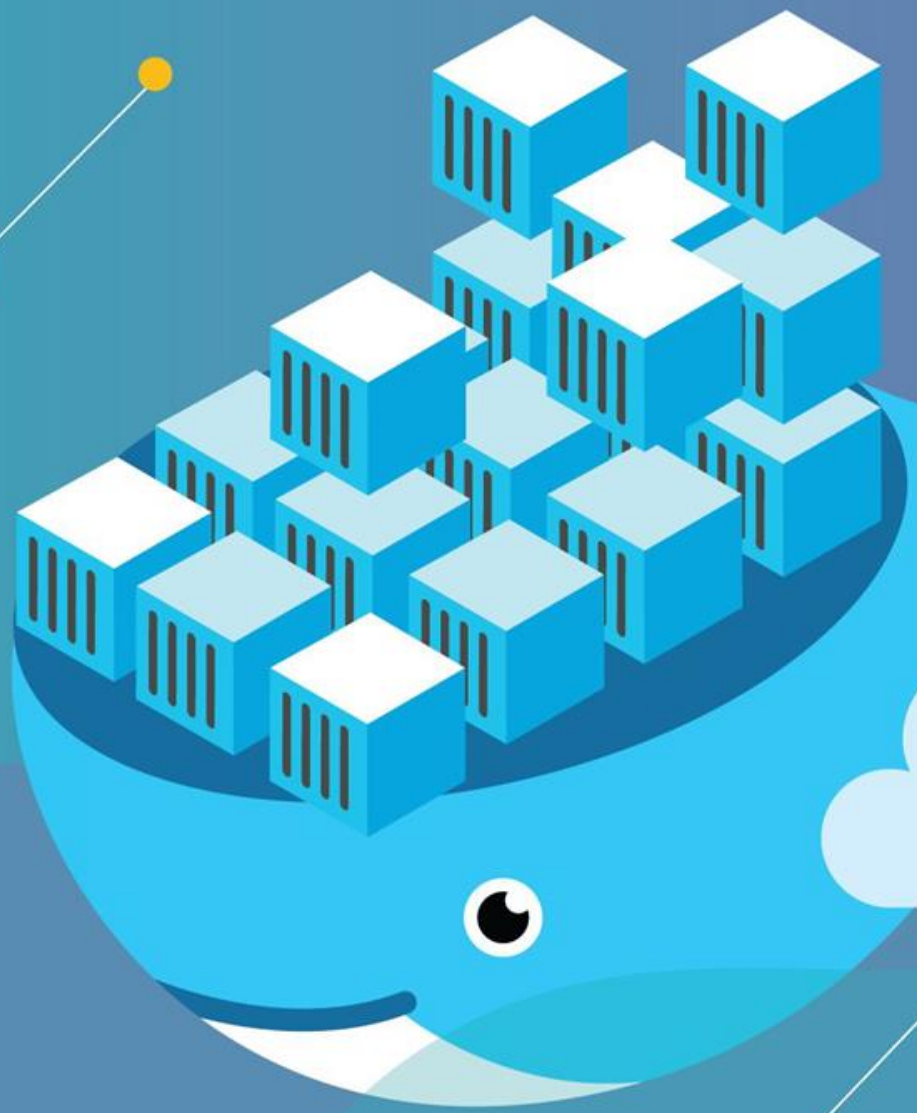
```
docker network create –driver bridge my_net
docker volume create mysql-data

docker run --name wordpress –p 8080:80 –v ./wordpress-data:/var/www/html \
–net my_net –e  WORDPRESS_DB_HOST = mysql \
-e WORDPRESS_DB_NAME: wordpress \
-e WORDPRESS_DB_USER: wordpress \
-e  WORDPRESS_DB_PASSWORD: wordpress \
wordpress

docker run --name wordpress –p 3306 –v  mysql-data:/var/lib/mysql –net my_net
-e  MYSQL_ROOT_PASSWORD: wordpress \
-e  MYSQL_DATABASE: wordpress \
-e  MYSQL_USER: wordpress \
-e  MYSQL_PASSWORD: wordpress \
mariadb
```



NEXT

Docker Swarm

Docker **Swarm**



Vikram
IoT Application Dev

github.com/kunchalavikram1427

Container Orchestration

- Containers are ephemeral by nature.
- They stop when process inside them finishes, or ends because of an error.
- Also a single container per service may not be sufficient enough to handle the growing traffic to the application.
- In all the above scenarios we need a tool that can bring up the stopped containers or spin up new one's to handle the growing traffic and to ensure Load balancing & High availability of the application all the time.
- This is where container orchestration comes into play!!!

What is container orchestration?

Container orchestration is all about managing the lifecycles of containers, especially in large, dynamic environments.

- ✓ Provisioning and deployment of containers
- ✓ Scaling up or removing containers to spread application load evenly across host infrastructure
- ✓ Movement of containers from one host to another if there is a shortage of resources in a host, or if a host dies
- ✓ Load balancing of service discovery between containers

Docker Swarm

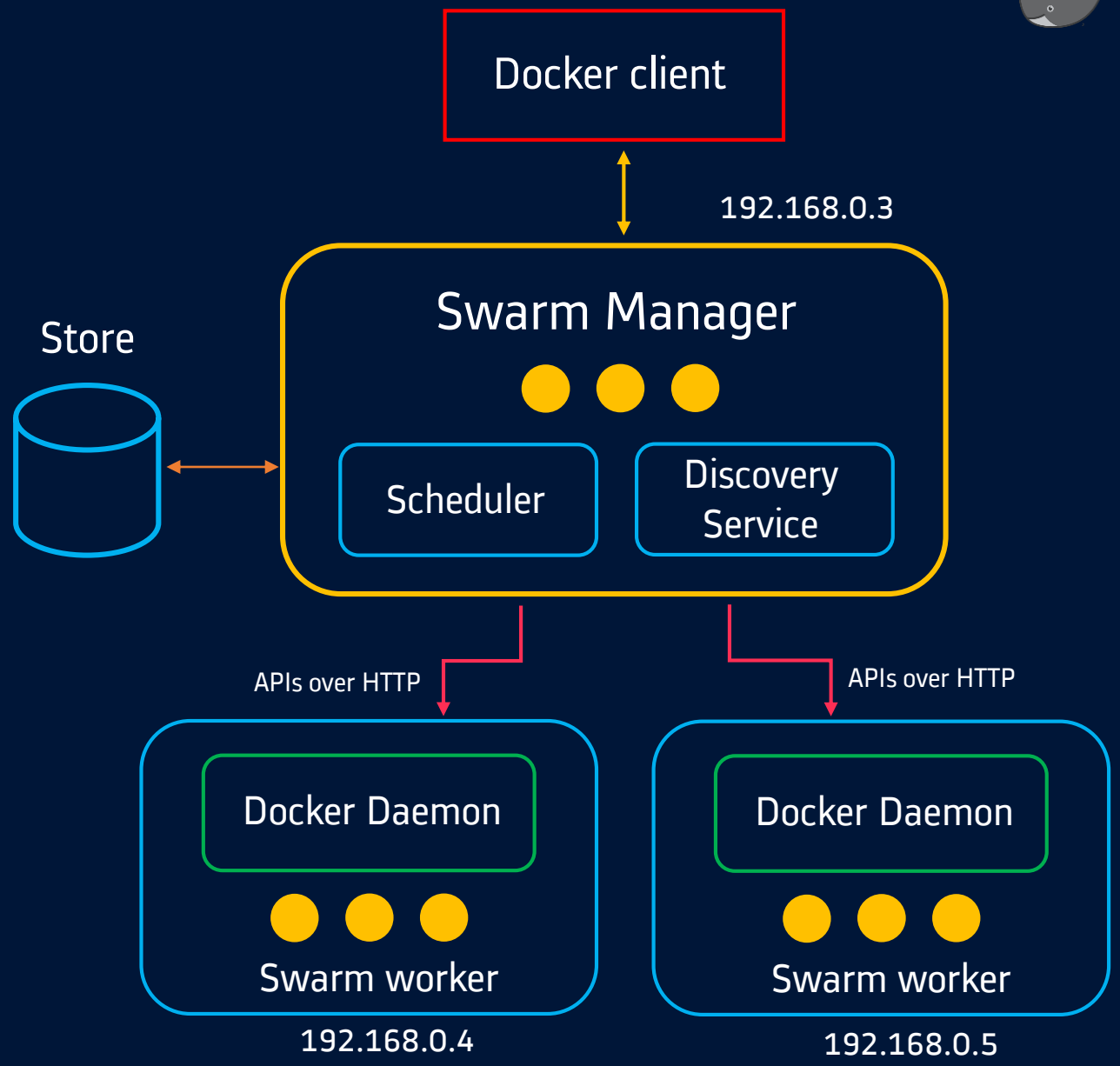
- Swarm is Docker's built in container orchestrator solution, its main purpose is to manage containers in a cluster, i.e. a set of connected machines that work together.
- When a new machine joins the cluster, it becomes a node in that swarm.
- Using Docker Swarm we can achieve high availability, load balancing and Decentralized access of our applications.
- Swarm comes built into the Docker Engine, you don't need to install anything to get started.

What can one do by using Docker Swarm?

- ✓ Deploying new containers to replace failed ones
- ✓ Scale the number of containers for load balancing
- ✓ Rolling out application updates among the containers in rolling update fashion
- ✓ Rollback of applications to older versions
- ✓ Bring down a node for maintenance without any application downtime

Docker Swarm Architecture

- Manager: manages the cluster
- Worker: runs tasks assigned by scheduler
- Scheduler: schedules containers onto nodes depending upon rules and filters
- Discovery service: helps Swarm Manager discover new nodes and fetch the available nodes
- Store: stores state of cluster; like cluster and swarm services info. Essentially a key-value db

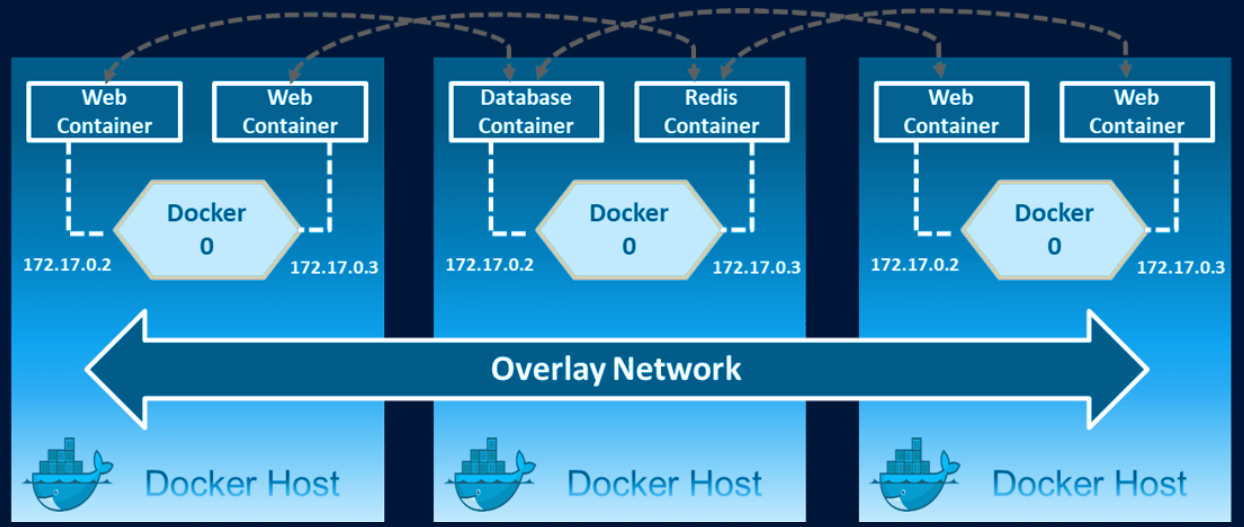


Docker Swarm: Overlay driver

- Bridge networks apply to containers running on the same Docker daemon host. For communication among containers running on different Docker daemon hosts like in swarm, we should use an overlay network
- If you create swarm services and do not specify a network, they are connected to default ingress network, which is also of overlay type
- Overlay network spans across the entire swarm cluster, allowing communication between containers across multiple nodes

docker network ls

```
root@docker-master:/home/osboxes/docker# docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
4a505a08943f        bridge              bridge              local
91bc90acf6b6        docker_gwbridge     bridge              local
305bfc47d41a        host                host                local
zyaa6fek5d7j        ingress             overlay             swarm
56ddd6ef1d30        my_net              bridge              local
5347bd97c94f        none                null                local
root@docker-master:/home/osboxes/docker#
```



https://docs.docker.com/engine/swarm/ingress/

Docker Swarm: Start a cluster

docker swarm init --advertise-addr MANAGER_IP

```
root@docker-master # docker swarm init --advertise-addr 192.168.0.100
Swarm initialized: current node (pet61mspmyfvk6lz0klisd9e9) is now a manager.

To add a worker to this swarm, run the following command:

    docker swarm join --token SWMTKN-1-3kabuqlekspyqdk02vj0strami9mcysj8lw4klp5cwmh3wm4ej-9rlom6m90chce1tgi0ke58t4l 192.168.0.100:2377

To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instructions.
```

master node

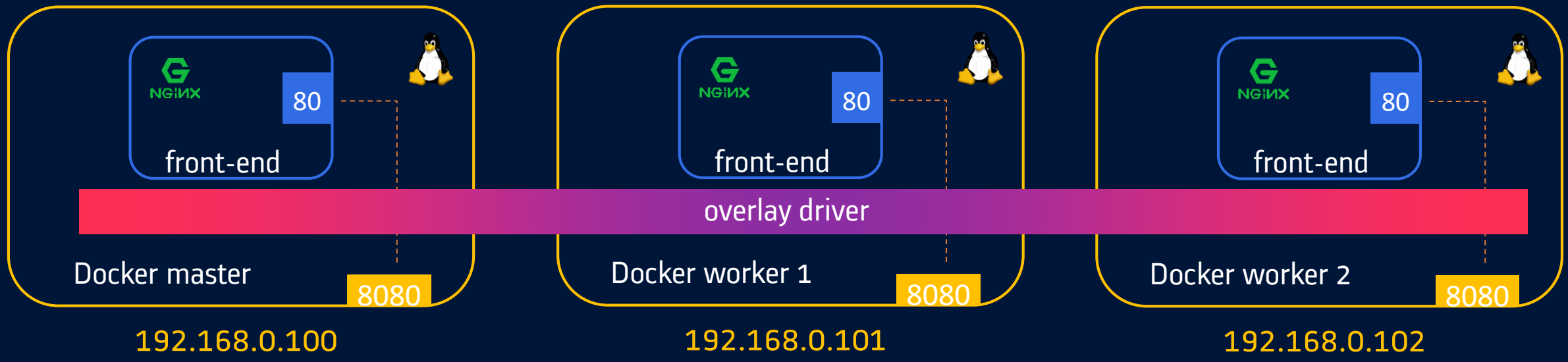
```
root@docker-slave01 #  docker swarm join --token SWMTKN-1-3kabuqlekspyqdk02vj0strami9mcysj8lw4klp5cwmh3wm4ej-9rlom6m90chce1tgi0ke58t4l 192.168.0.100:2377
This node joined a swarm as a worker.
root@docker-slave01:/home/osboxes#
```

worker node

Docker Swarm

docker node ls

```
root@docker-master:/home/osboxes# docker node ls
ID                            HOSTNAME            STATUS              AVAILABILITY        MANAGER STATUS
m75v5bd6jd1vsy9usawecsu7r *   docker-master       Ready               Active              Leader
wywos5cppaol45967s0mtqj1n     docker-slave01      Ready               Active
uimguox32lco6odxhpdq3nxf2     docker-slave02      Ready               Active
root@docker-master:/home/osboxes#
```



root@docker-master # docker network ls | grep overlay
qh17ylskxm31 ingress overlay swarm

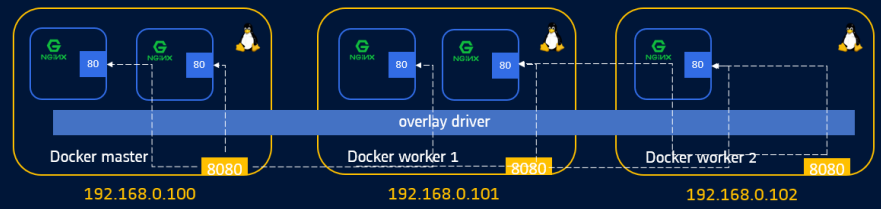
Docker Service

- To deploy an application image when Docker Engine is in swarm mode, you create a service.
- A service is the image for a microservice like an HTTP server, a database, or any other type of executable program that you wish to run in a distributed environment.
- A service needs container image to use, the port where the swarm makes the service available outside the swarm, an overlay network for the service to connect to other services in the swarm and the number of replicas of the image to run in the swarm.

Create a service

docker service create --replicas 5 -p 80:80 --name web nginx

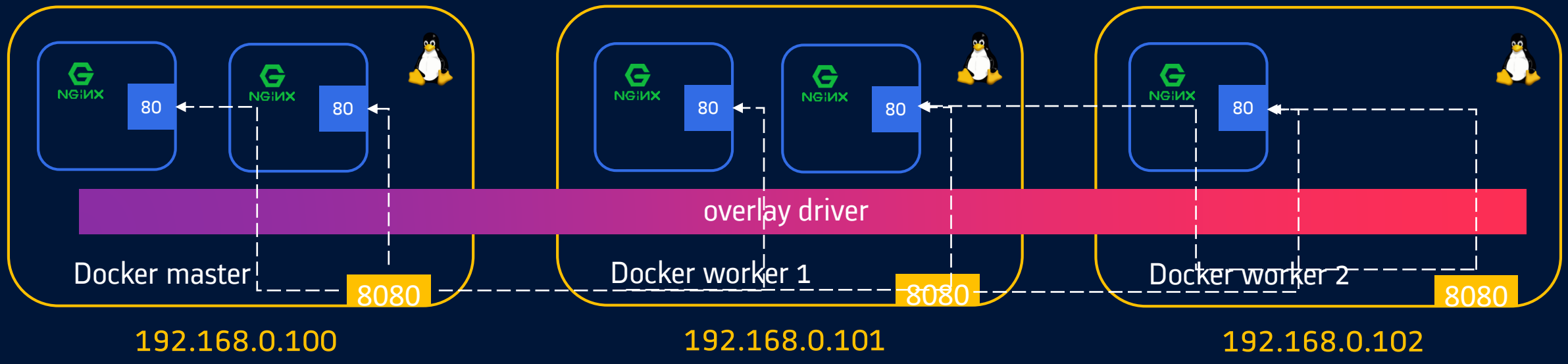
```
root@docker-master:/home/osboxes# docker service create --replicas 5 -p 80:80 --name web nginx
6obmcjveg1zk5eb9nd3ypj90e
overall progress: 5 out of 5 tasks
1/5: running   [==================================================>]
2/5: running   [==================================================>]
3/5: running   [==================================================>]
4/5: running   [==================================================>]
5/5: running   [==================================================>]
verify: Service converged
```



Docker Service

docker service ls

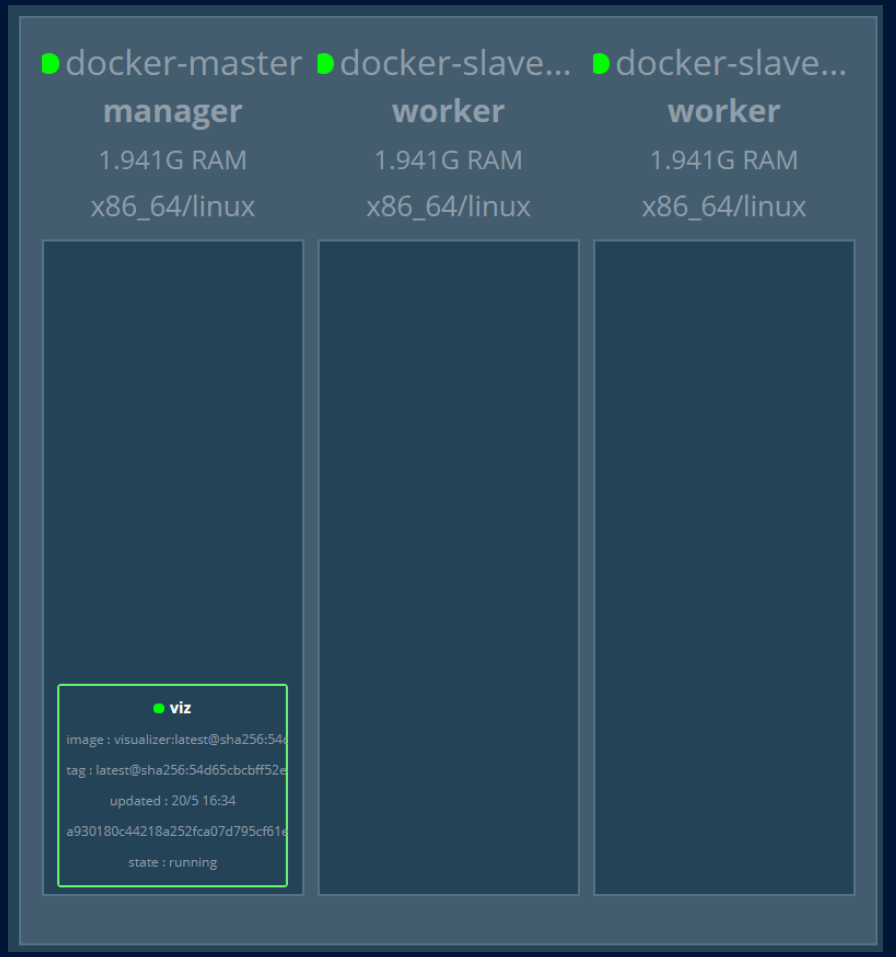
```
root@docker-master:/home/osboxes# docker service ls
ID                  NAME                MODE                REPLICAS            IMAGE               PORTS
6obmcjveg1zk        web                 replicated          5/5                 nginx:latest        *:80->80/tcp
root@docker-master:/home/osboxes#
```



Docker Swarm Visualizer

Deploy a visualizer

```
docker service create \
  --name=viz \
  --publish=8080:8080/tcp \
  --constraint=node.role==manager \
  --mount=type=bind,src=/var/run/docker.sock,dst=/var/run/docker.sock \
  dockersamples/visualizer
```



Dashboard available at http://<node-ip>:8080

*node-ip: any IP of node participating in cluster
*although the pod is bound to master, it can be reached from all nodes because of ingress routing mesh

https://github.com/dockersamples/docker-swarm-visualizer

Docker Service

docker service ps web

Name of the service

```
root@docker-master:/home/osboxes# docker service ps web
ID                  NAME                IMAGE               NODE                DESIRED STATE       CURRENT STATE
      PORTS
qpu0955zl5rb        web.1               nginx:latest        docker-slave02      Running             Running 2 minutes ago

ytrie6t5kb50        web.2               nginx:latest        docker-master       Running             Running 2 minutes ago

rxubw2mh8hxv        web.3               nginx:latest        docker-slave01      Running             Running 2 minutes ago

rv9apxkz07xa        web.4               nginx:latest        docker-slave02      Running             Running 2 minutes ago

s1dgil2sp35d        web.5               nginx:latest        docker-slave01      Running             Running 2 minutes ago
```

docker node ps docker-slave01

Name of the node

```
root@docker-master:/home/osboxes# docker node ps docker-slave01
ID                  NAME                IMAGE               NODE                DESIRED STATE       CURRENT STATE
      PORTS
rxubw2mh8hxv        web.3               nginx:latest        docker-slave01      Running             Running 3 minutes ago

s1dgil2sp35d        web.5               nginx:latest        docker-slave01      Running             Running 3 minutes ago
```

Docker Service

docker service ps web

```
root@docker-master:/home/osboxes# docker service ps web
ID                  NAME                IMAGE               NODE                DESIRED STATE       CURRENT STATE
      PORTS
qpu0955zl5rb        web.1               nginx:latest        docker-slave02      Running             Running 2 minutes ago

ytrie6t5kb50        web.2               nginx:latest        docker-master       Running             Running 2 minutes ago

rxubw2mh8hxv        web.3               nginx:latest        docker-slave01      Running             Running 2 minutes ago

rv9apxkz07xa        web.4               nginx:latest        docker-slave02      Running             Running 2 minutes ago

s1dgil2sp35d        web.5               nginx:latest        docker-slave01      Running             Running 2 minutes ago
```

Master

Slave -01

Slave -02

docker service create --replicas 5 -p 80:80 --name web nginx

Docker Service

Scale a service

docker service scale <service_name>=8

docker service scale web=10

Update a service

docker service update --image <image_name>:<version> <service_name>

docker service update --image nginx:1.18 web

Rolling update

docker service update –image <new_image> --update-parallelism 2 --update-delay 10s <service_name>

docker service update --image nginx:1.18 --update-parallelism 2 --update-delay 10s web

Docker Service

Let’s say, a node is down because of an issue

In this case the affected containers in that node are recreated in available nodes in the cluster

docker service ls

```
root@docker-master:/home/osboxes# docker service ls
ID                  NAME                MODE                REPLICAS            IMAGE
6obmcjveg1zk        web                 replicated          3/5                 nginx:latest
```

```
root@docker-master:/home/osboxes# docker service ls
ID                  NAME                MODE                REPLICAS            IMAGE
6obmcjveg1zk        web                 replicated          5/5                 nginx:latest
```

docker service ps web

```
root@docker-master:/home/osboxes# docker service ps web
ID                  NAME                IMAGE               NODE                DESIRED STATE       CURRENT STATE
4d65ub7ctbyg        web.1               nginx:latest        docker-master       Running             Running 18 seconds ago
qpu0955zl5rb         \_ web.1           nginx:latest        docker-slave02      Shutdown            Running 37 seconds ago
ytrie6t5kb50        web.2               nginx:latest        docker-master       Running             Running 8 minutes ago
rxubw2mh8hxv        web.3               nginx:latest        docker-slave01      Running             Running 8 minutes ago
kaytsldhrdho        web.4               nginx:latest        docker-master       Running             Running 17 seconds ago
rv9apxkz07xa         \_ web.4           nginx:latest        docker-slave02      Shutdown            Running 37 seconds ago
s1dgil2sp35d        web.5               nginx:latest        docker-slave01      Running             Running 8 minutes ago
```

Docker Service

- Let's say, a node has to be brought down for maintenance

  In this case we drain a node of its workloads onto other nodes for graceful termination

  Drain a node: docker node update --availability drain <node-name>
  Undrain a node: docker node update --availability active <node-name>

- Let's say, the node is up and running after maintenance

  By default, swarm doesn't re-deploy the containers the node was previously running. We need to force the cluster to share the load among the available nodes

  Force workload balance: docker service update --force <service_name>

Rollback an update: docker service update --rollback <service_name>

Remove a service: docker service rm <service_name>

Docker Swarm: Overlay driver

Creating overlay network

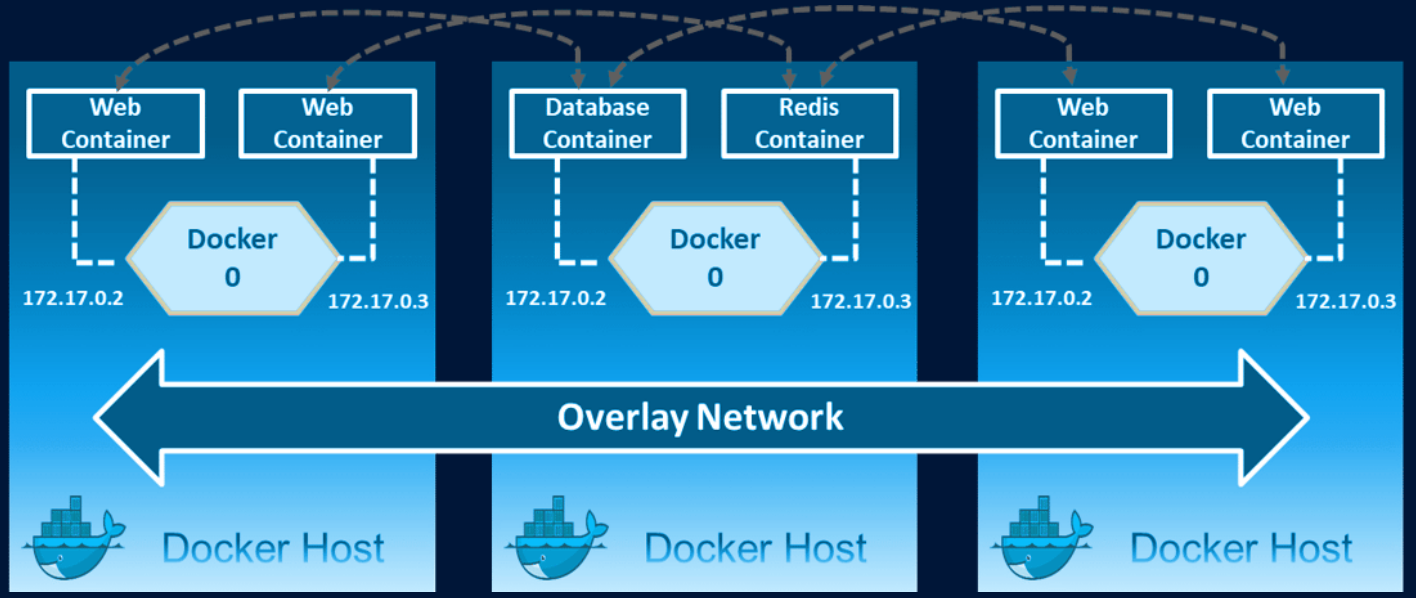
docker network create --driver overlay my_overlay

```
root@docker-master:/home/osboxes/docker# docker network ls | grep overlay
zyaa6fek5d7j        ingress             overlay             swarm
mv4mvtgnteeb        my_overlay          overlay             swarm
root@docker-master:/home/osboxes/docker#
```

docker service create --replicas 5 -p 80:80 --network my_overlay --name web nginx

Inspecting the service:

docker service inspect web



https://docs.docker.com/engine/swarm/ingress/

Docker Service

Demo: Swarm service

Dockerfile

```
FROM python:alpine3.7
COPY . /app
WORKDIR /app
RUN pip install flask
EXPOSE 5000
CMD python ./app.py
```

```
docker build –t <username>/flask-app .
docker push <username>/flask-app
```

app.py

```
import socket
from flask import Flask
def getIP():
        hostname = socket.gethostname()
        s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        s.connect(("8.8.8.8", 80))
        ip = s.getsockname()[0]
        print(ip)
        s.close()
        return hostname,ip

app = Flask(__name__)
@app.route("/")
def hello():
    hostname,ip = getIP()
    out = "Hello from hostname: " + hostname + " with host ip: " + ip
    return out
if __name__ == "__main__":
    app.run(host="0.0.0.0", port=int("5000"), debug=True)
```

In order to use a custom image in swarm cluster, make sure to push this image to docker hub or private repo so that the swarm nodes will pull the image, or, make sure this image is available locally on all nodes by manually building the image from dockerfile. Swarm will not schedule containers on to those nodes which fail to pull the image or do not have this image locally!

Docker Service

Demo: Swarm service

```
docker service create --replicas 5 -p 80:5000 --name web kunchalavikram/sampleflask:v2
```

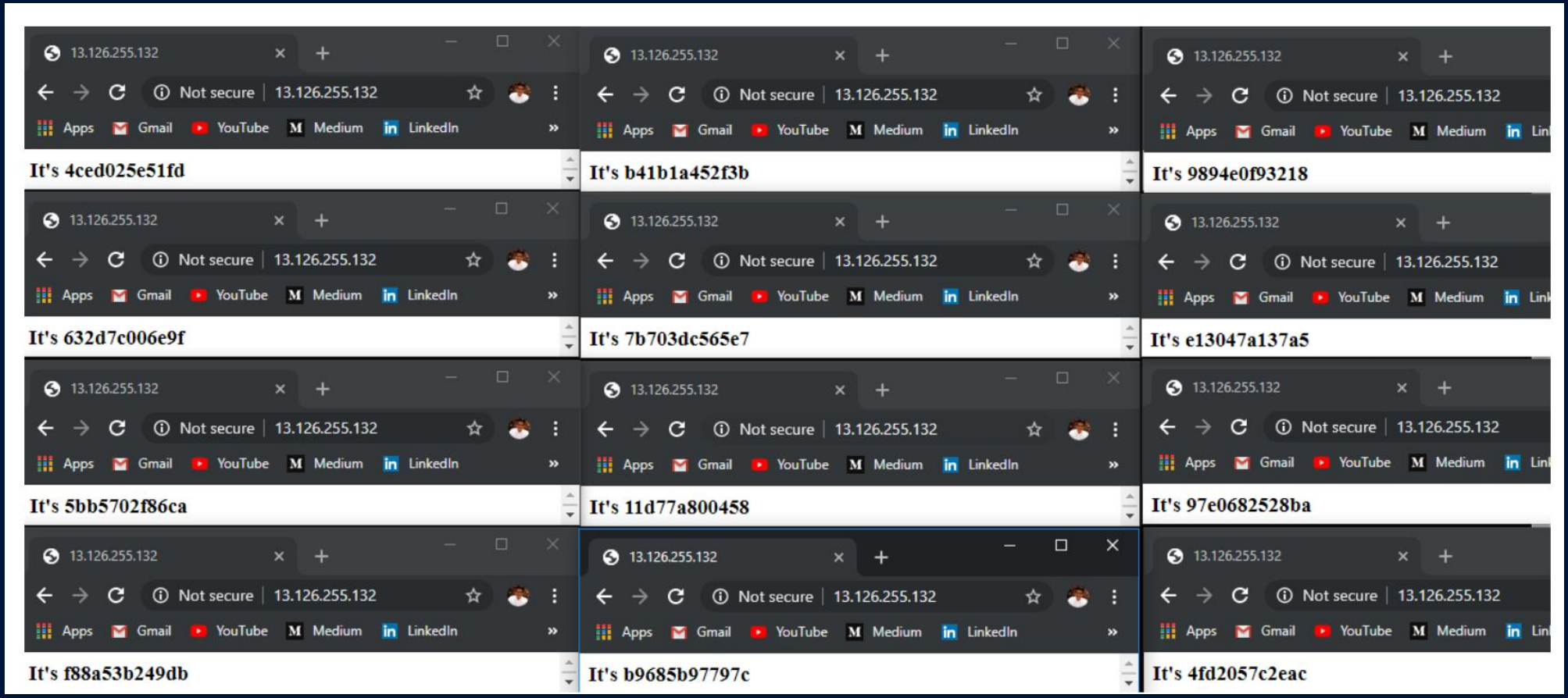
visit
http://<node-ip>:80
and refresh the page to see the load balancing effect

Docker Service

Demo: Swarm service

Find similar swarm load-balancing demo at below link

https://levelup.gitconnected.com/load-balance-and-scale-node-js-containers-with-nginx-and-docker-swarm-9fc97c3cff81



Docker Networking

Ingress Network

Is it possible to run 2 containers with same host port published in non swarm mode?

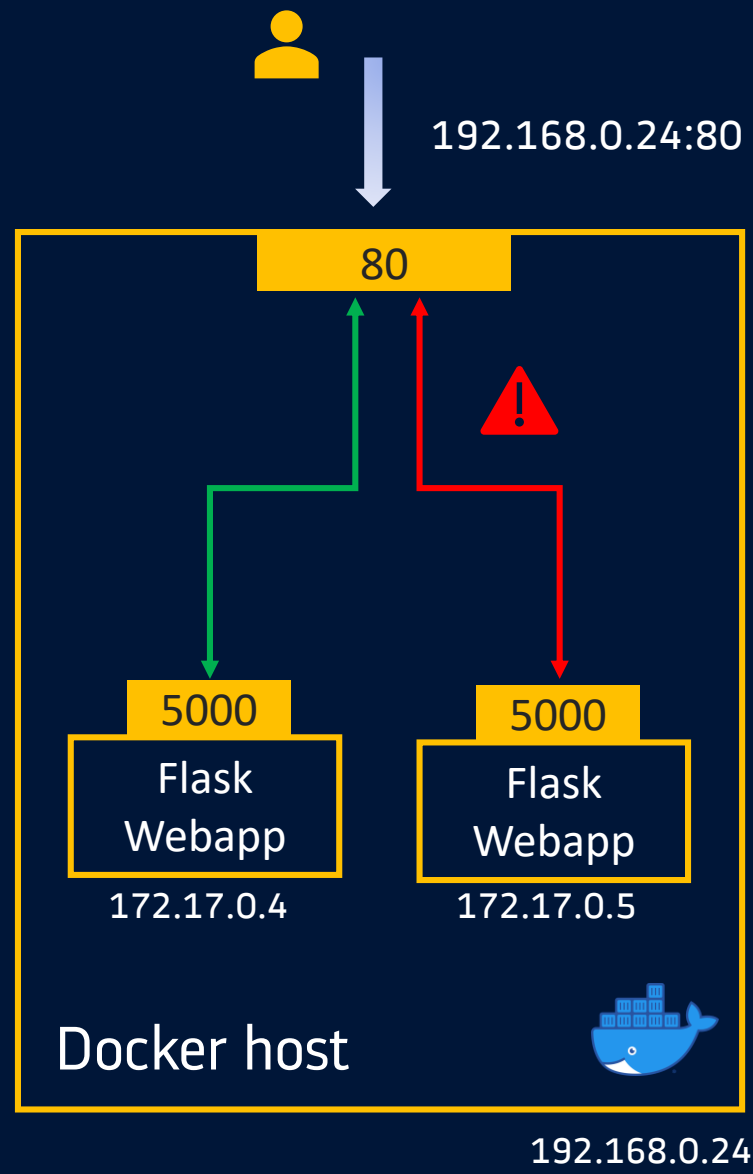
```
docker run –d –p 80:5000 --name web1 flask
docker run –d –p 80:5000 --name web2 flask
```

No!!! It results in port conflict. Docker daemon doesn't allow same host port for multiple containers in non swarm mode

Then how does docker allow running multiple instances of same application(with same host port mapping) in a node inside a swarm cluster?

```
docker service create --replicas 5 -p 80:5000 --name web flask
```

Ingress Routing Mesh

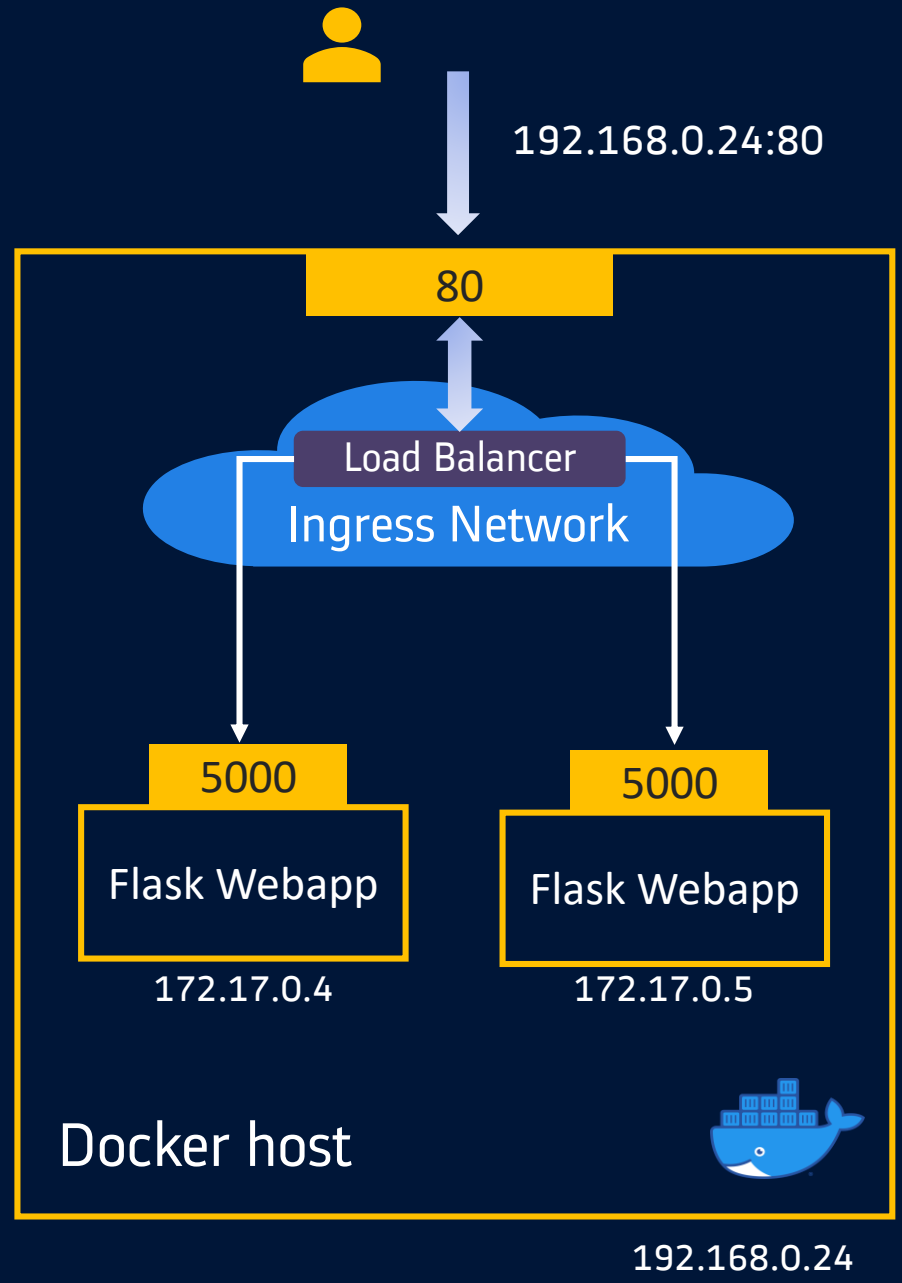


Docker Networking

Ingress Network

- Ingress network has a built in load balancer that redirects traffic from published port on host to the mapped container ports
- This allows multiple containers inside a docker host to use same host port without port collisions
- No external setup is needed, the load balancer works out of the box with the ingress network

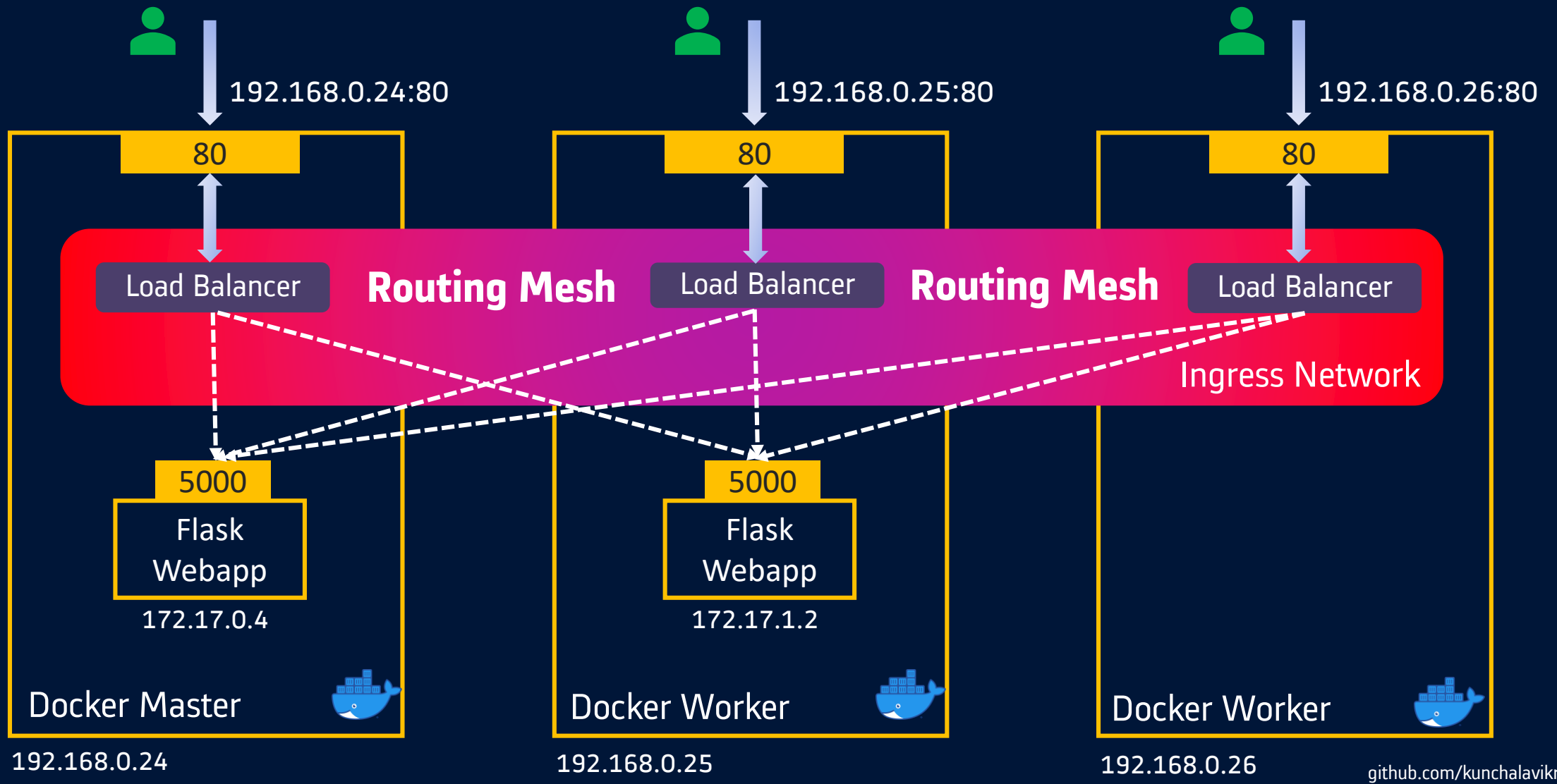
```
docker service create --replicas 5 -p
80:5000 --name web flask
```



Docker Networking

Ingress Routing Mesh

docker service create --replicas 2 -p 80:5000 --name web flask

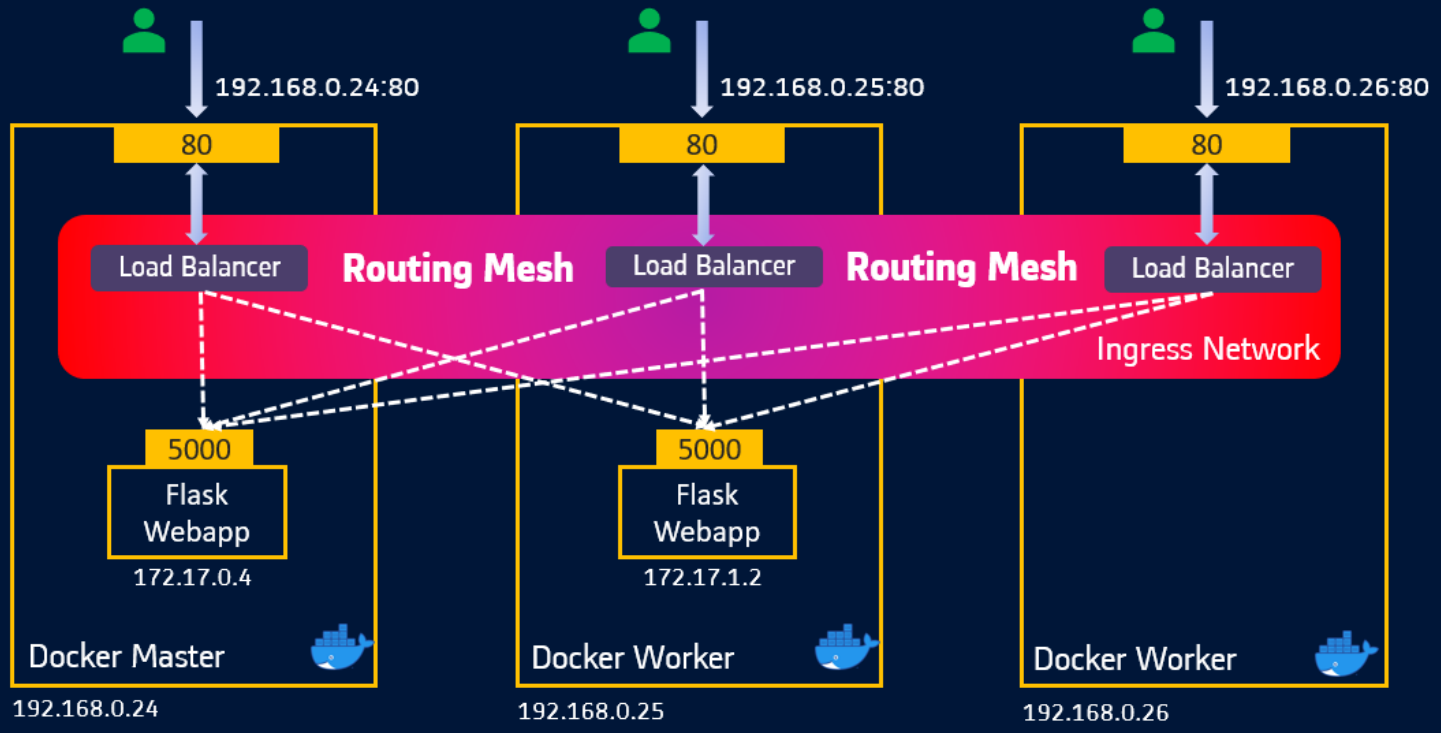


Docker Networking

Ingress Routing Mesh

The swarm internal networking mesh allows every node in the cluster to accept connections to any service port published in the swarm by routing all incoming requests to available nodes hosting a service with the published port.

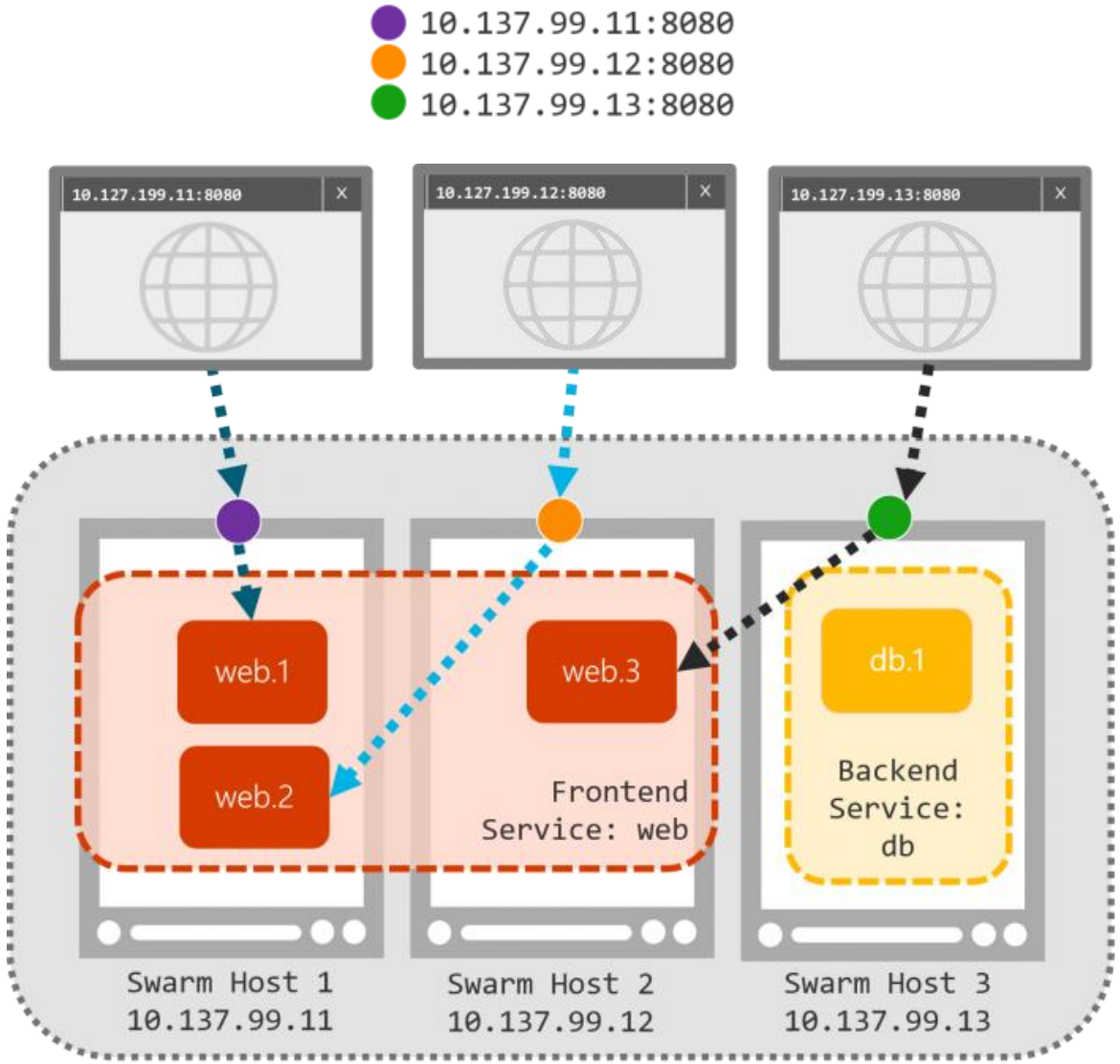
docker service create --replicas 2 -p 80:5000 --name web flask



Docker Service

Ingress Routing Mesh

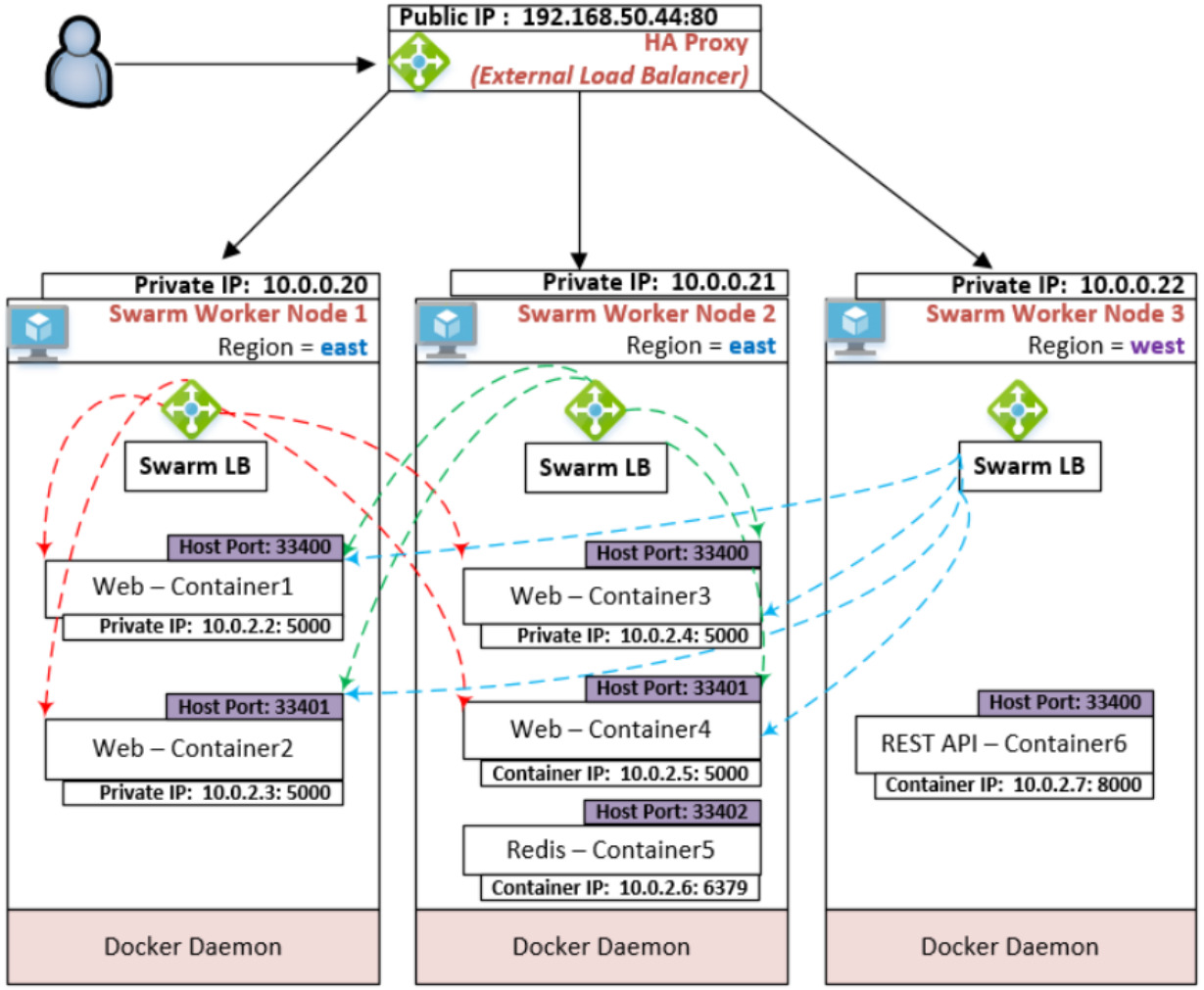
- All nodes participate in an **ingress routing mesh**. The routing mesh enables each node in the swarm to accept connections on published ports for any service running in the swarm, even if there's no task running on the node.
- The routing mesh routes all incoming requests to published ports on available nodes to an active container.
- In this scenario, we have 3 nodes and 4 replicas of web containers. The web containers are scheduled onto only Host 1 and Host 2. But still we can reach the web services from Host 3 because of swarm routing mesh



https://docs.docker.com/network/network-tutorial-overlay/
https://docs.docker.com/engine/swarm/ingress/

Docker Service

Load Balancer (ingress routing mesh)



The ingress network is a special overlay network that facilitates load balancing among a service's nodes. When any swarm node receives a request on a published port, it hands that request off to a module called IPVS. IPVS keeps track of all the IP addresses participating in that service, selects one of them, and routes the request to it, over the ingress network.

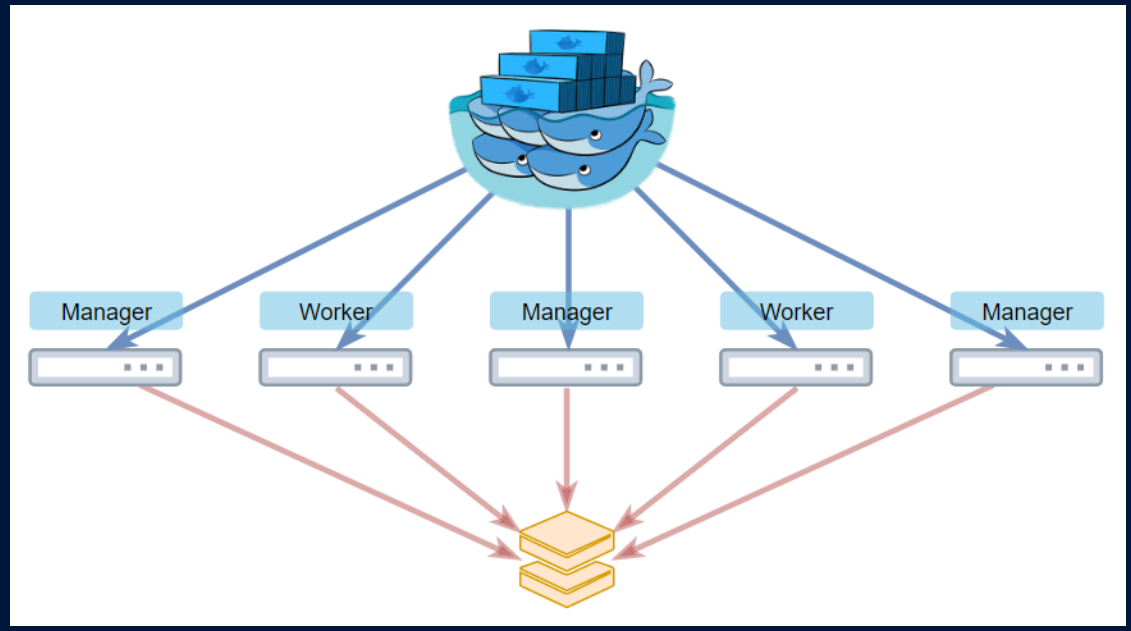
Docker Swarm

Persistent Storage

- Persistent storage is storing the data of the container beyond its lifecycle.
- For single host containers, Bind or Named mounts works fine, but in order to share storage volumes across multiple Docker hosts, such as a Swarm cluster, we need distributed FS or Network FS.

Use of Distributed/Network FS

- ✓ Because programs running on your cluster aren't guaranteed to run on a specific node, data can't be saved to any arbitrary place in the file system.
- ✓ If a program tries to save data to a file for later, but is then relocated onto a new node, the file will no longer be where the program expects it to be.
- ✓ Distributed/Network FS allows applications to have a common storage volume to store and retrieve the data beyond its lifecycle.



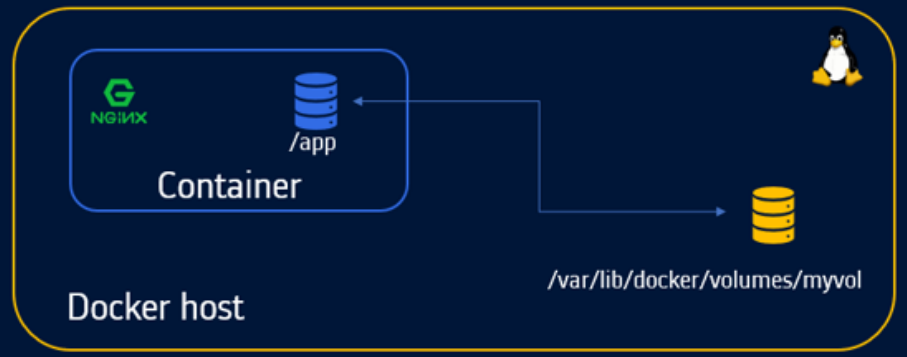
https://theworkaround.com/2019/05/15/docker-swarm-persistent-storage.html

Docker Swarm

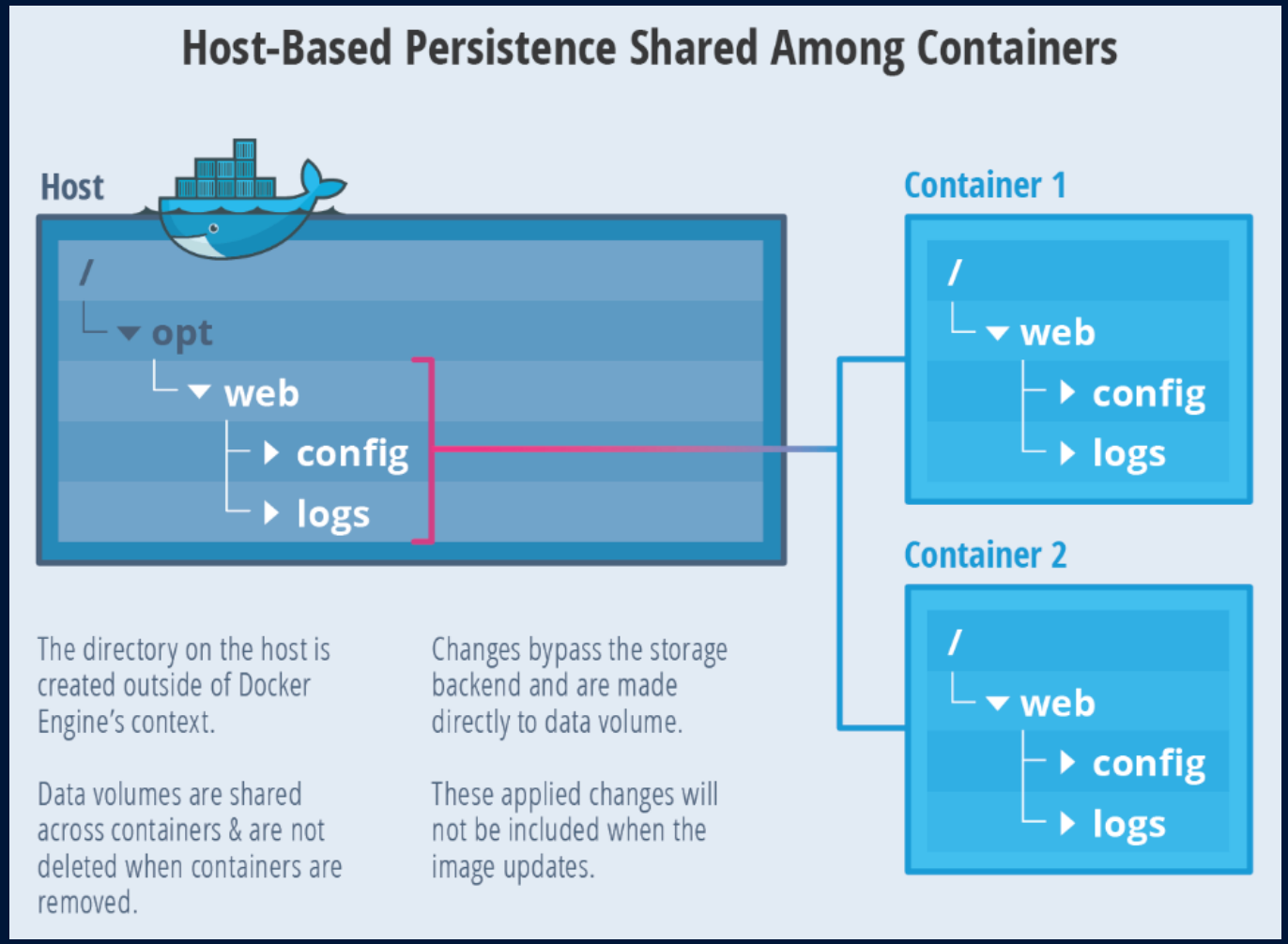
Persistent Storage in single node

- Named Mount
- Bind Mount

Bind Volume



Named Volume



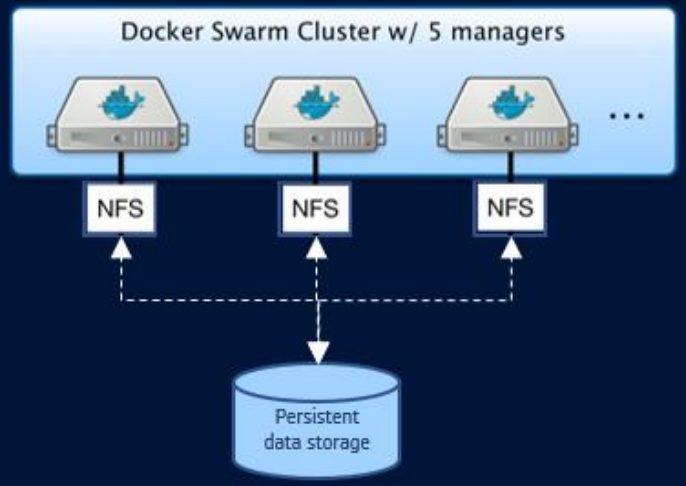
https://thenewstack.io/methods-dealing-container-storage/

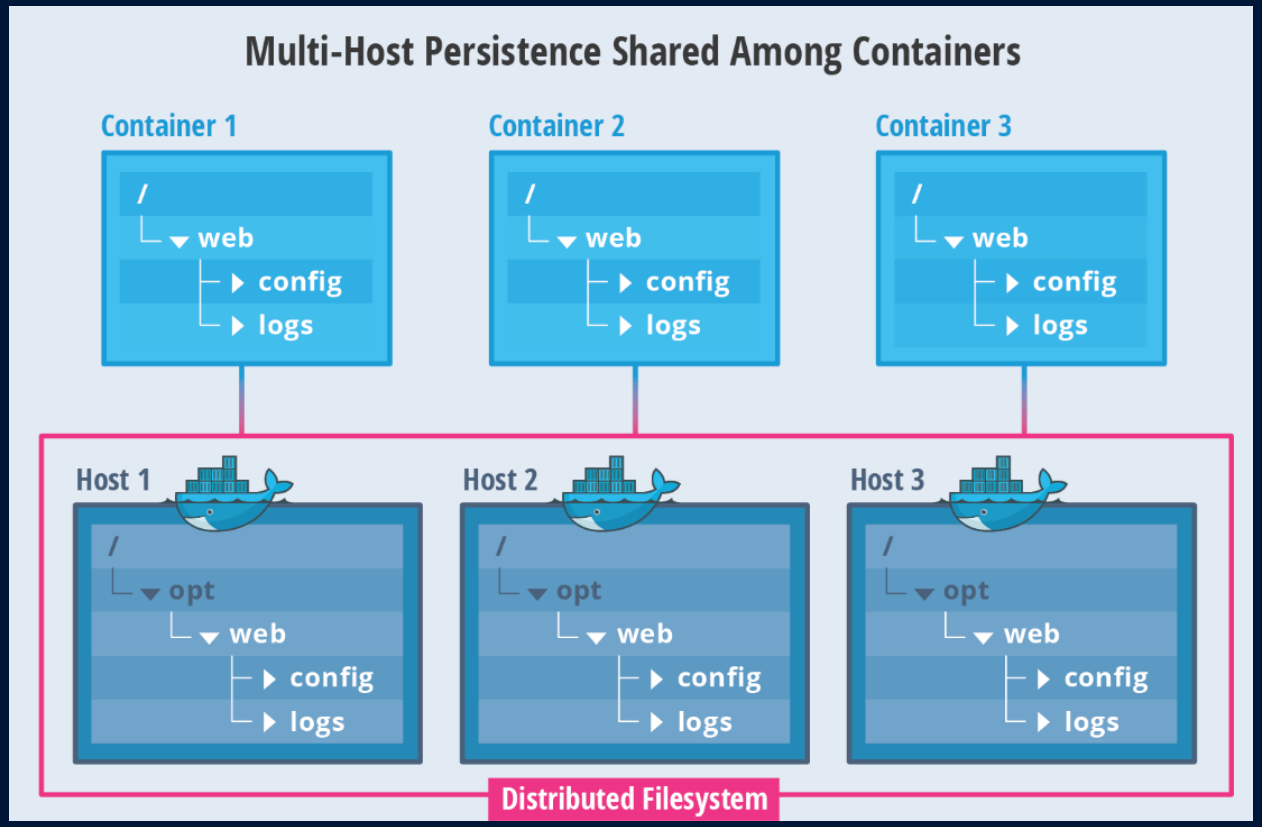
Docker Swarm

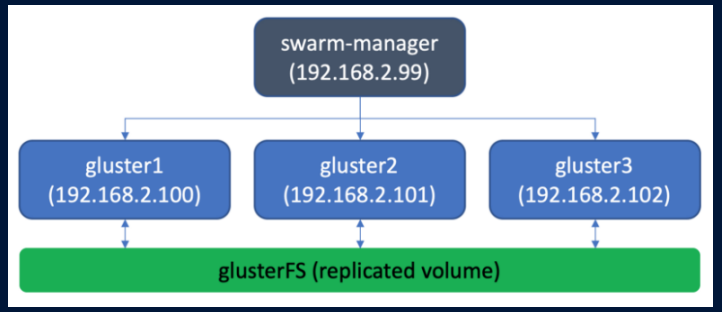
Persistent Storage across cluster

Requires distributed/network file storage

- NFS
- GlusterFS
- Ceph
- Convoy
- RexRay
- PortWorx
- StorageOS





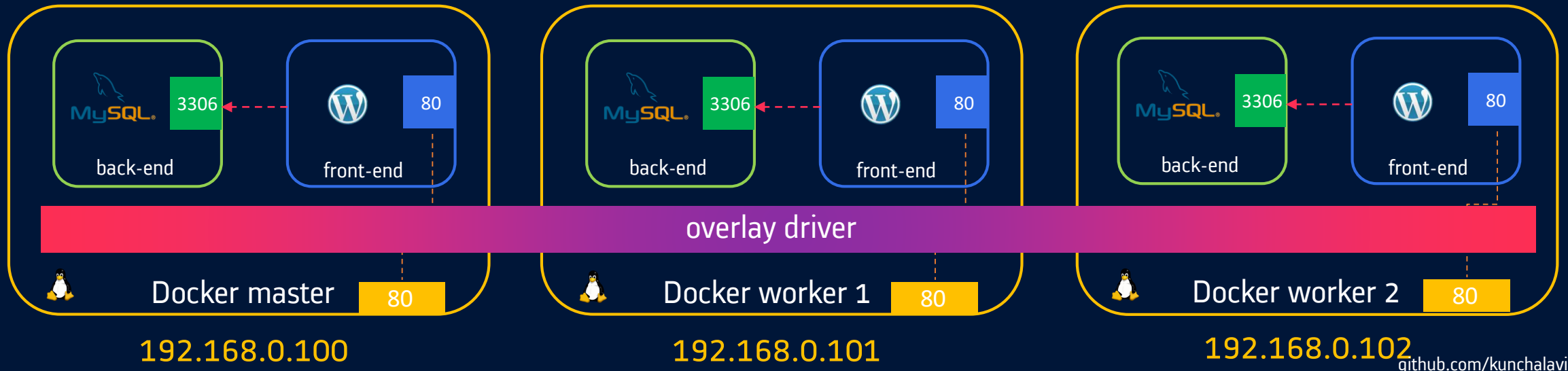


GlusterFS

https://theworkaround.com/2019/05/15/docker-swarm-persistent-storage.html

Docker Stacks

- Docker stack is used to deploy a complete application stack to the swarm.
- Production grade docker-compose
- Docker-compose is better suited for development scenarios and is a single host deployment solution
- Uses the same syntax as docker-compose file, but with a new section ‘deploy’ added to manifest file
- Docker stack ignores “build” instructions. You can’t build new images using the stack commands. It needs pre-built images to exist



Docker Stack

```yaml
version: '3.3'
services:
  wordpress:
    image: wordpress
    depends_on:
      - mysql
    ports:
      - 80:80
    deploy:
      replicas: 2
      placement:
        constraints:
          - node.role == manager
    environment:
      WORDPRESS_DB_HOST: mysql
      WORDPRESS_DB_NAME: wordpress
      WORDPRESS_DB_USER: wordpress
      WORDPRESS_DB_PASSWORD: wordpress
    volumes:
      - wordpress-data:/var/www/html
    networks:
      - my_net
  mysql:
    image: mariadb
    deploy:
      replicas: 1
      placement:
        constraints:
          - node.role == worker
    environment:
      MYSQL_ROOT_PASSWORD: wordpress
      MYSQL_DATABASE: wordpress
      MYSQL_USER: wordpress
      MYSQL_PASSWORD: wordpress
    volumes:
      - mysql-data:/var/lib/mysql
    networks:
      - my_net
networks:
  my_net:
volumes:
  mysql-data:
  wordpress-data:
```

```yaml
wordpress:
  image: wordpress
  depends_on:
    - mysql
  ports:
    - 80:80
  deploy:
    replicas: 2
    placement:
      constraints:
        - node.role == manager
  environment:
    WORDPRESS_DB_HOST: mysql
    WORDPRESS_DB_NAME: wordpress
    WORDPRESS_DB_USER: wordpress
    WORDPRESS_DB_PASSWORD: wordpress
  volumes:
    -  wordpress-data:/var/www/html
  networks:
    - my_net
```

Constraints allow containers to be bound to a specific node. When these containers are down because of an issue, swarm re-deploys another one in the same node thus ensuring data persistency all the time

```yaml
mysql:
  image: mariadb
  deploy:
    replicas: 1
    placement:
      constraints:
        - node.role == worker
  environment:
    MYSQL_ROOT_PASSWORD: wordpress
    MYSQL_DATABASE: wordpress
    MYSQL_USER: wordpress
    MYSQL_PASSWORD: wordpress
  volumes:
    - mysql-data:/var/lib/mysql
  networks:
    - my_net
```

```yaml
volumes:
  mysql-data:
  wordpress-data:
networks:
  my_net:
```

https://docs.docker.com/engine/swarm/services/#placement-constraints
https://thenewstack.io/methods-dealing-container-storage/

Docker Stacks

```yaml
version: '3.3'
services:
  wordpress:
    image: wordpress
    depends_on:
      - mysql
    ports:
      - 80:80
    deploy:
      replicas: 2
      placement:
        constraints:
          - node.role == manager
    environment:
      WORDPRESS_DB_HOST: mysql
      WORDPRESS_DB_NAME: wordpress
      WORDPRESS_DB_USER: wordpress
      WORDPRESS_DB_PASSWORD: wordpress
    volumes:
      - wordpress-data:/var/www/html
    networks:
      - my_net
  mysql:
    image: mariadb
    deploy:
      replicas: 1
      placement:
        constraints:
          - node.role == worker
    environment:
      MYSQL_ROOT_PASSWORD: wordpress
      MYSQL_DATABASE: wordpress
      MYSQL_USER: wordpress
      MYSQL_PASSWORD: wordpress
    volumes:
      - mysql-data:/var/lib/mysql
    networks:
      - my_net
networks:
  my_net:
volumes:
  mysql-data:
  wordpress-data:
```

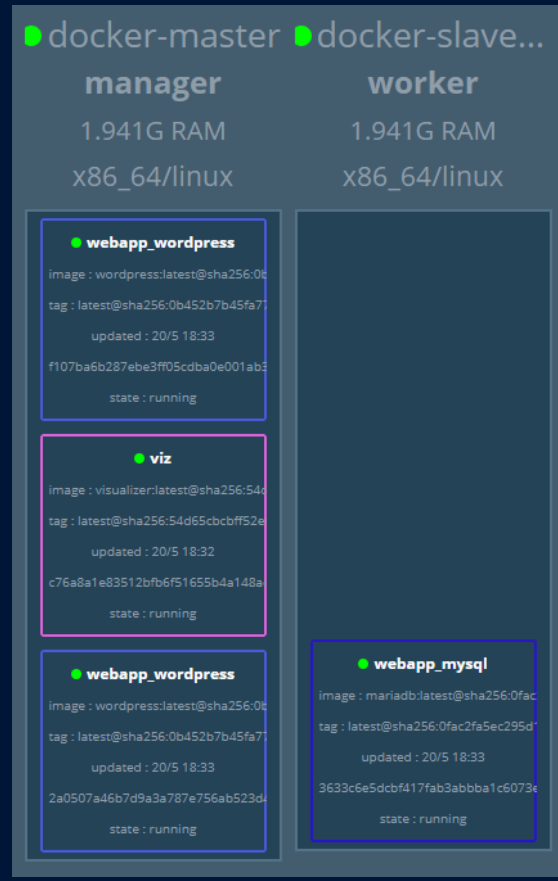
Deploy a stack: docker stack deploy -c <compose.yml> <stack-name>

List stacks: docker stack ls

List processes: docker stack ps <stack-name>

List services: docker service ls

Delete stack: docker stack rm <stack-name>

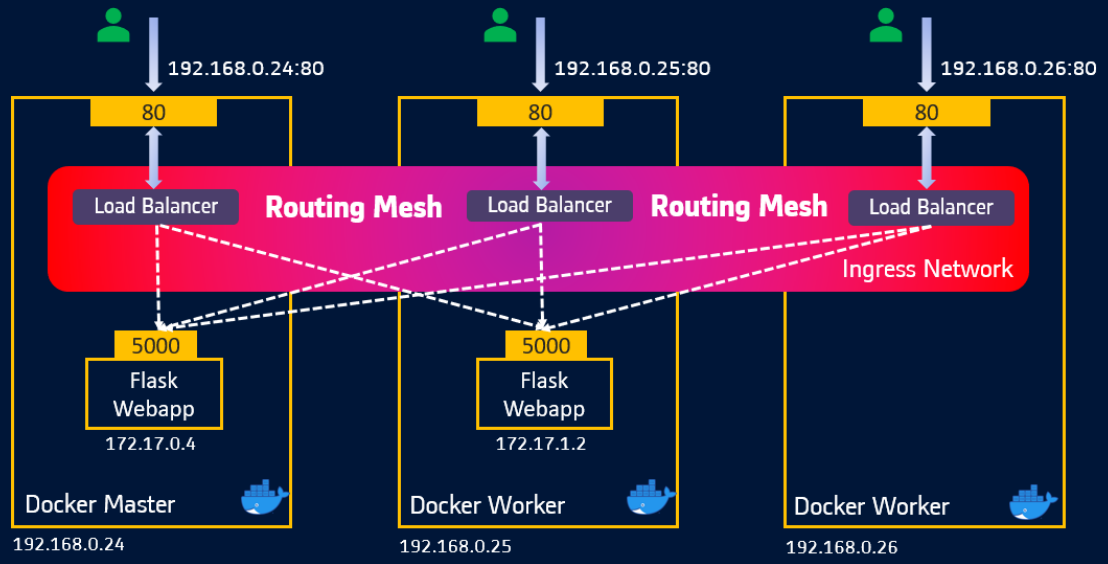


https://docs.docker.com/engine/swarm/services/#placement-constraints

Docker Swarm

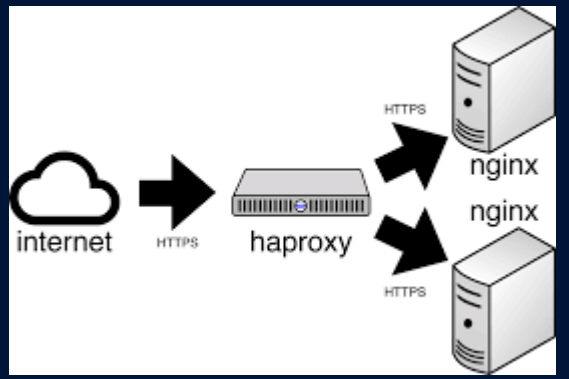
LoadBalancer

- One of the key benefits of docker swarm is increasing application availability through redundancy
- Application can be reached from all the available nodes in the cluster using <node-ip>:port
- But how do end user access this application thorough a common endpoint i.e., DNS name? It is through an External LoadBalancer/Reverse Proxy
- HAProxy load balancer routes external traffic into the cluster and load balancing its across the available nodes
- Other popular load balancers include Nginx and Traefik



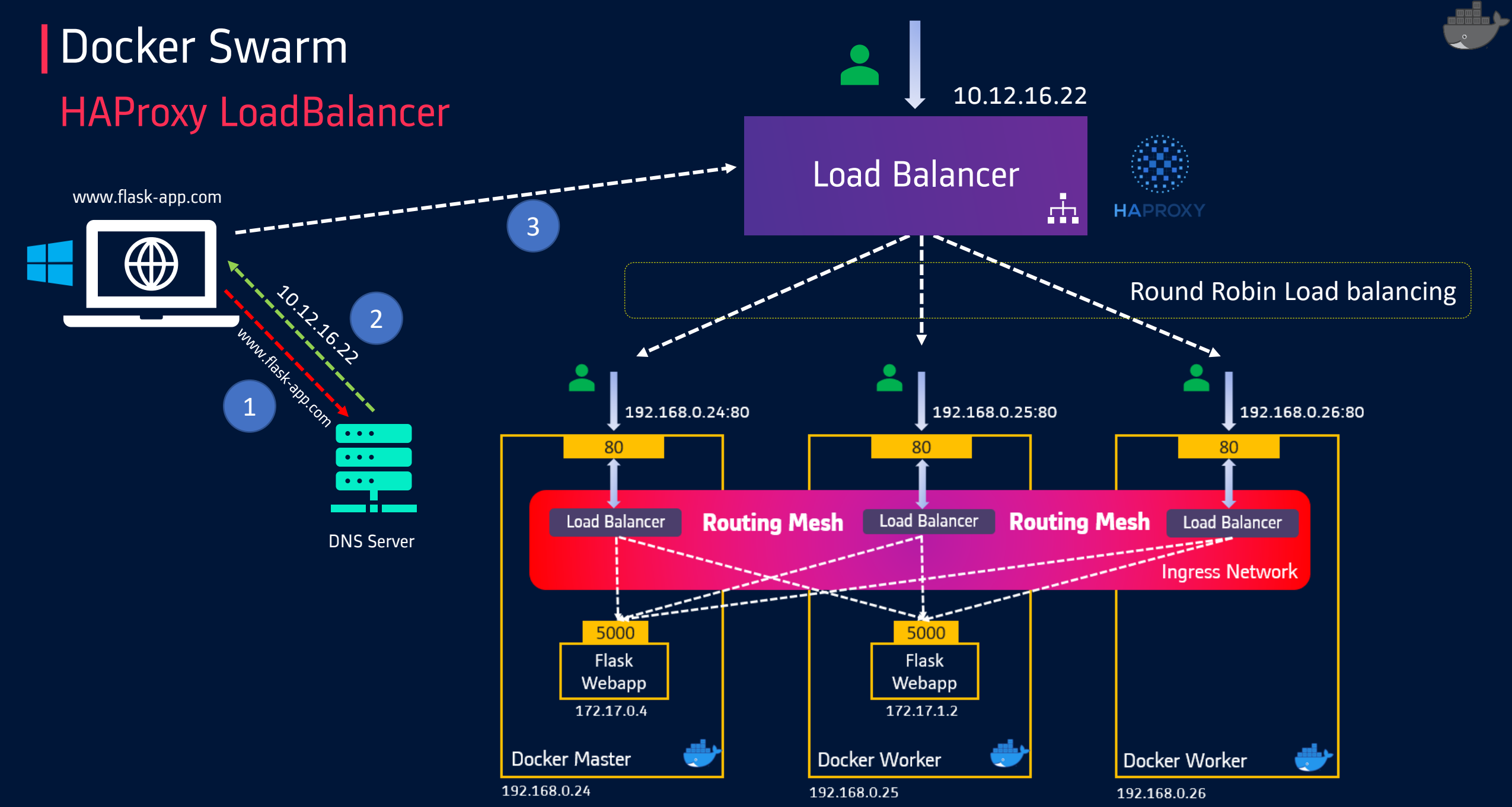
Traefik

NGINX



Docker Swarm

HAProxy LoadBalancer



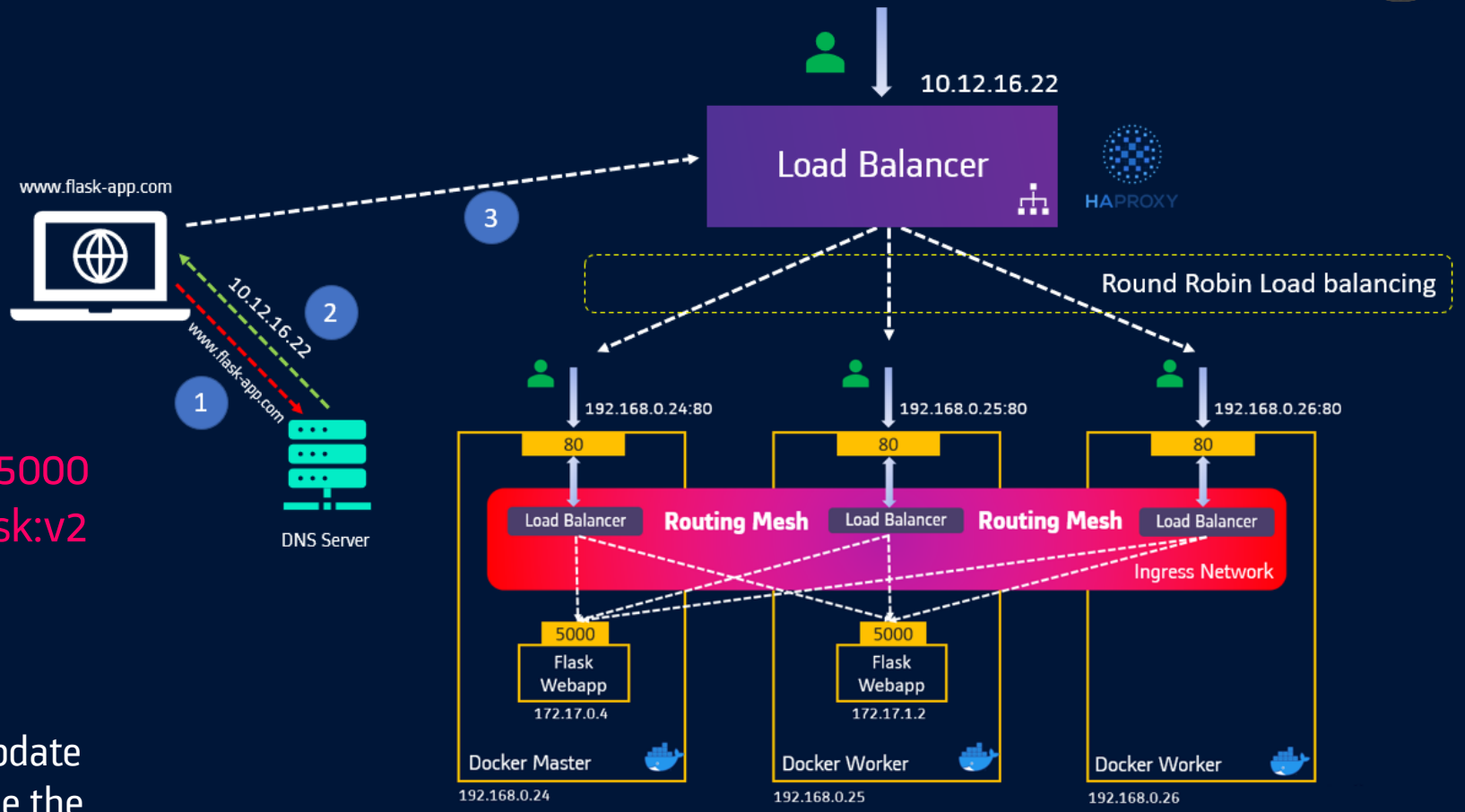
Docker Swarm

HAProxy LoadBalancer

Demo

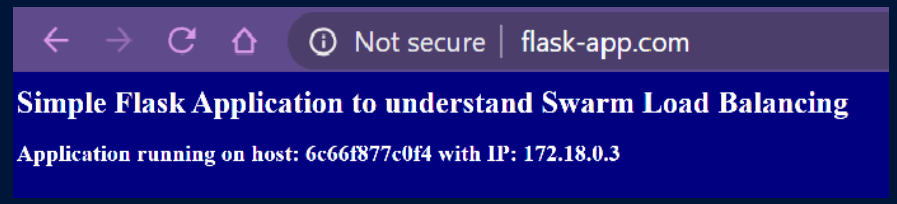
1. create a service of 5 flask replicas

```
docker service create --replicas 5 -p 80:5000
--name web kunchalavikram/sampleflask:v2
```



2. Create fake DNS entries

(since we don't have DNS server, we will update /etc/hosts file in any machine, outside/inside the cluster, with fake DNS name that points to IP of HAProxy server. We will use this name to access the flask application)



3VMs are used for swarm cluster and 1VM for running HAProxy. These are just demo considerations. Not intended for production

Docker Swarm

HAProxy LoadBalancer

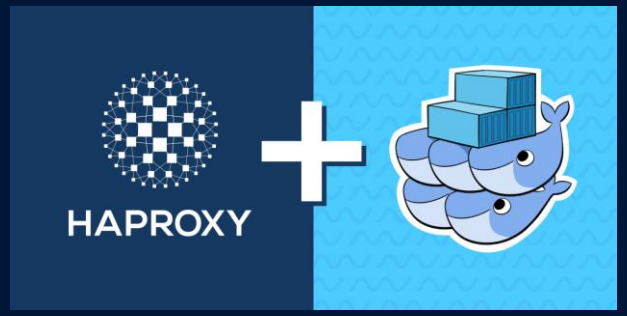
Setup HAProxy

- Provision a VM
- apt install haproxy –y
- systemctl stop haproxy
- add configuration to /etc/haproxy/haproxy.cfg
- systemctl start haproxy && systemctl enable haproxy

192.168.0.105 /etc/haproxy/haproxy.cfg

```
# Configure HAProxy to listen on port 80
frontend http_front
   bind *:80
   default_backend http_back

# Configure HAProxy to route requests to swarm nodes on port 8080
backend http_back
   balance roundrobin
   mode http
   server srv1 192.168.0.111:80
   server srv2 192.168.0.104:80
   server srv3 192.168.0.109:80
```

swarm nodes IP and application port



Create a Fake DNS Entry that points to IP of HAProxy Machine

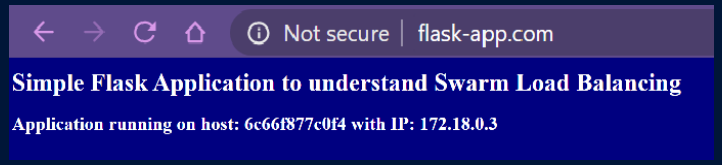
windows

C:\Windows\System32\drivers\etc\hosts
192.168.0.105 flask-app.com
ipconfig /flushdns

linux

/etc/hosts
192.168.0.105 flask-app.com

http://flaskapp.com

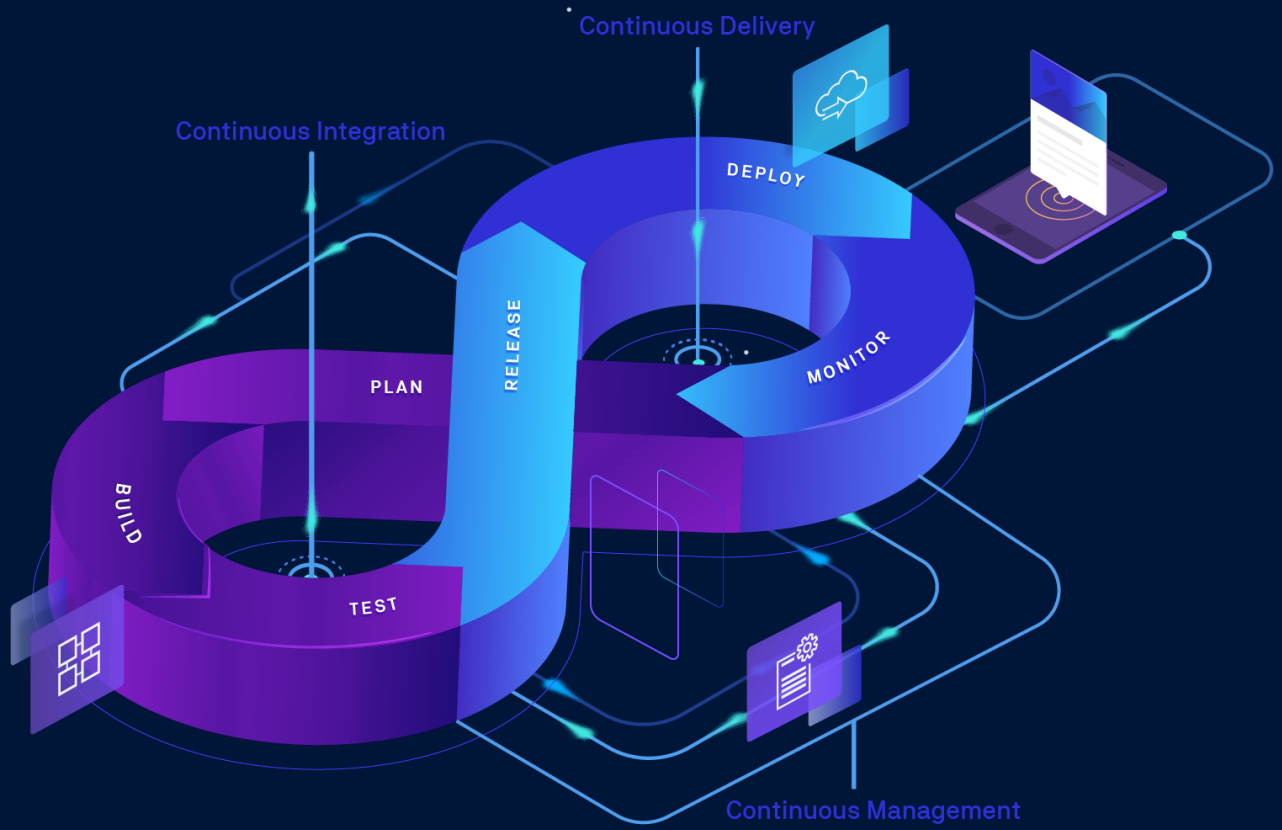


References

- Hypervisors
  https://phoenixnap.com/kb/what-is-hypervisor-type-1-2
- Docker
  https://takacsmark.com/getting-started-with-docker-in-your-project-step-by-step-tutorial/
  https://container.training/
- Docker-compose
  https://docs.docker.com/compose/compose-file/
  https://docs.docker.com/compose/compose-file/compose-versioning/
- Docker Swarm
  https://knowledgepill.it/posts/docker_swarm_compendium/?fbclid=IwAR2JWNRHrmfKlauScuEmcW1hwI7IxLJdf6_KUWN8GQiSqxc5pXmkw9RDEaM
  https://takacsmark.com/getting-started-with-docker-in-your-project-step-by-step-tutorial/
  https://container.training/
  https://rominirani.com/docker-swarm-tutorial-b67470cf8872
- Docker Networking
  https://docs.docker.com/network/network-tutorial-overlay/
  https://docs.docker.com/engine/swarm/ingress/
- Routing Mesh
  https://mstechbits.wordpress.com/2019/06/06/load-balancing-docker-containers/amp/
  https://techcommunity.microsoft.com/t5/containers/docker-s-routing-mesh-available-with-windows-server-version-1709/ba-p/382382#
  https://github.com/docker/docker.github.io/blob/master/engine/swarm/networking.md
- Placement Constraints in Swarm mode
  https://docs.docker.com/engine/swarm/services/#placement-constraints
- Traefik
  https://boxboat.com/2017/10/10/managing-multiple-microservices-with-traefik-in-docker-swarm/
  https://www.digitalocean.com/community/tutorials/how-to-use-traefik-as-a-reverse-proxy-for-docker-containers-on-ubuntu-18-04
- HAProxy
  https://www.haproxy.com/blog/haproxy-on-docker-swarm-load-balancing-and-dns-service-discovery/
- GlusterFS
  https://sysadmins.co.za/container-persistent-storage-for-docker-swarm-using-a-glusterfs-volume-plugin/
  https://medium.com/running-a-software-factory/setup-3-node-high-availability-cluster-with-glusterfs-and-docker-swarm-b4ff80c6b5c3
  https://blog.ruanbekker.com/blog/2019/03/05/setup-a-3-node-replicated-storage-volume-with-glusterfs/?referral=github.com
- Ceph
  https://ceph.io/

Credits

- All authors from references section
- Mumshad Mannambeth's Free Docker Course on Kodekloud https://kodekloud.com/p/docker-for-the-absolute-beginner-hands-on
- YouTube Videos
- Various blogs



github.com/kunchalavikram1427