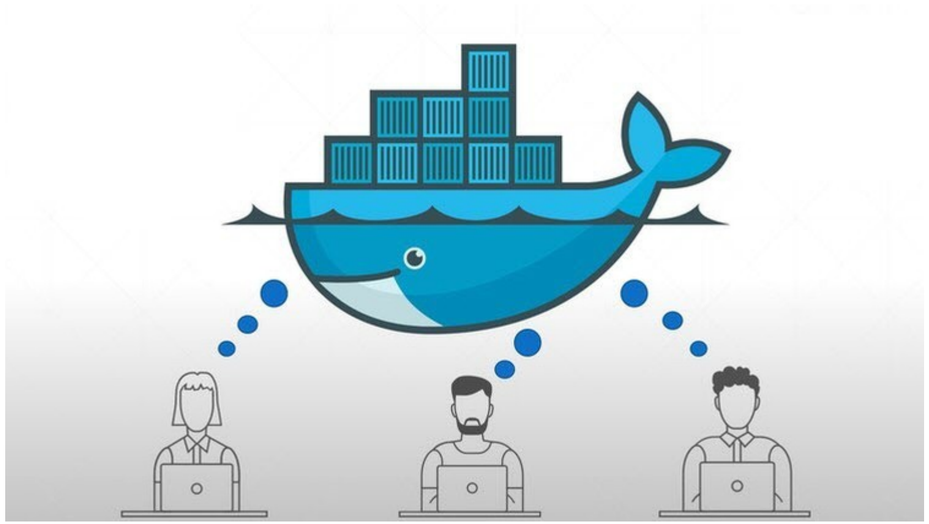
Introducción práctica a Docker

Juan Vicente Herrera Ruiz de Alejo



Add profile section | More...

Juan Vicente Herrera Ruiz de Alejo

Platform and automation engineer (DevOps culture) at X by Orange

X by Orange

Universidad Pontificia Co...

https://iuta.education/programas/master-en-industria-4-0/
http://madrid.devops.es/
https://www.linkedin.com/in/jvherrera/
http://www.juanvicenteherrera.es
twitter.com/jvicenteherrera
http://www.slideshare.net/juanvicenteherrera

Agenda

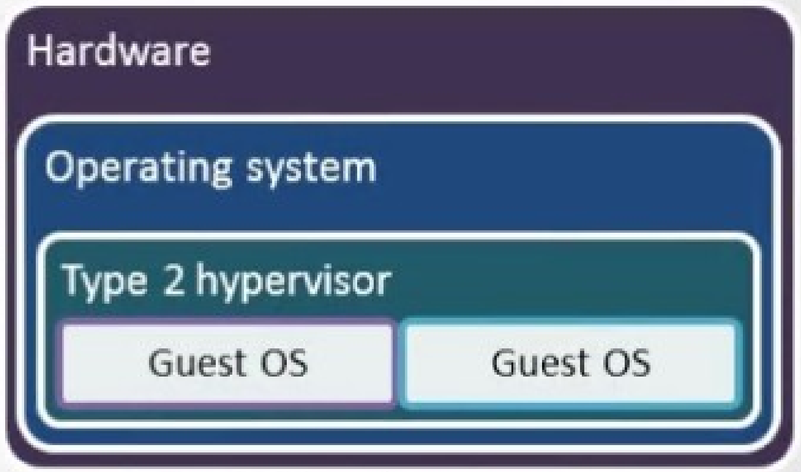
1) Introducción a Virtualización
2) Introducción a Containers
3) Docker Overview
4) Docker basics
5) Instalación de Docker Engine
6) Creación de Containers
7) Control de Containers en Docker
8) Contrucción de containers y administración de Dockerfiles
9) Administración Docker Volumes (almacenamiento)
10) Docker Hub
11) Gestión de red en Docker
12) Orquestación de contenedores con Docker Swarn
13) Hands on

Hypervisors

- Type 1
  - Bare metal installation over the hardware to control and monitor hardware and guest operating systems
  - VMware ESXi
  - Microsoft Hyper-V
    - Windows Server
  - Citrix Xen

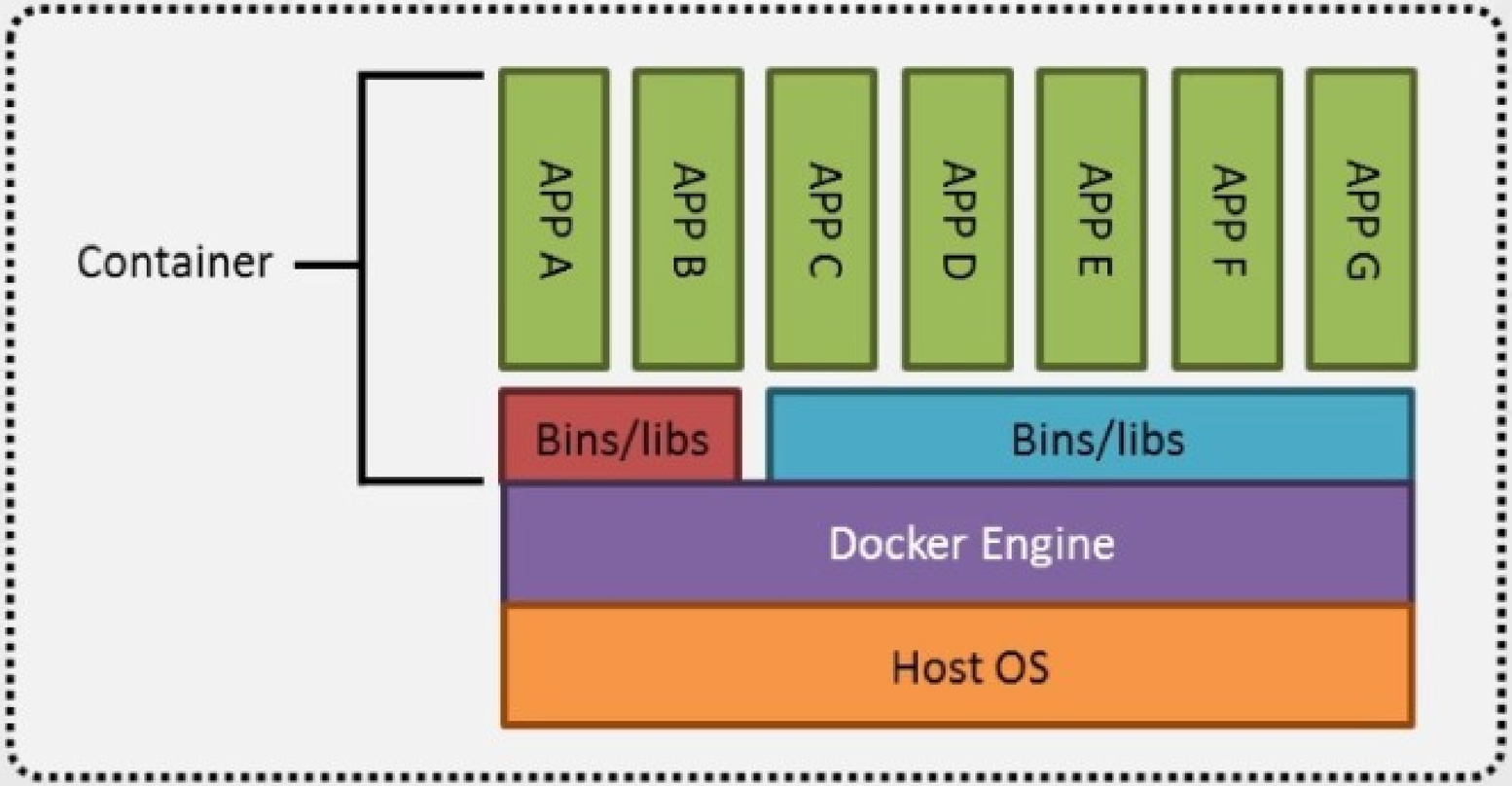
Hypervisors

- Type 2
  - Hypervisors are software applications that run inside of a traditional operating system
  - VMware Workstation
  - Microsoft Hyper-V
    - Windows 10
  - Oracle VM VirtualBox



Container Basics

- Isolated application runtime environment
- The OS and some binary libraries are shared by containers



Container Basics

- Containers are based on images
  - Images contain software and settings for running a container
  - Images contain metadata describing the image
  - A container is a runtime instance of an image
- Container contents
  - All elements needed to run an application
    - Software
    - Settings
    - App-specific libraries
    - Runtime environment
    - Tools

Container Basics

- Containers are an application sandboxing solution
  - Not tied to the host
  - Can be moved between Docker hosts
    - OS version on Docker host must be the same
    - Linux containers can run on Windows through Hyper-V
  - A reference image can be used for many application containers
    - E.g. Golden reference image with Python configured

Container Basics

- Container startup time
  - Very quick since the underlying OS kernel is already running
  - The container is an isolated OS kernel running process
- All application-specific elements
  - Stored within the container
  - In some cases, specific versions of binary libraries are required
    - Sometimes, these are not shared but stored in the container

Container Basics

- Data persistence
  - Application writes persist until the container is deleted
- Container scalability
  - Address peak application requests
  - Try not to run multiple services in the same container
- Container versioning
  - Only incremental changes are uploaded/downloaded

Container Basics

- Windows container runtime choices
  - Windows Server containers
    - Host OS kernel is shared among containers
    - The host computer can see the running process/container
  - Hyper-V containers
    - Uses a lightweight VM
    - Separate kernel for each container
    - The process is unknown to the host computer
    - This is how Windows 10 can run containers

Virtualization and Containers

- OS virtualization
  - The physical hypervisor hardware is virtualized
  - Starting the virtual machines means waiting for OS boot
  - The VM contains all OS files
- Containers
  - The OS is virtualized and not the hardware
  - Starts much more quickly than virtual machines
    - The OS kernel is already running
    - The OS is shared among multiple containers
  - Contains only app-related files

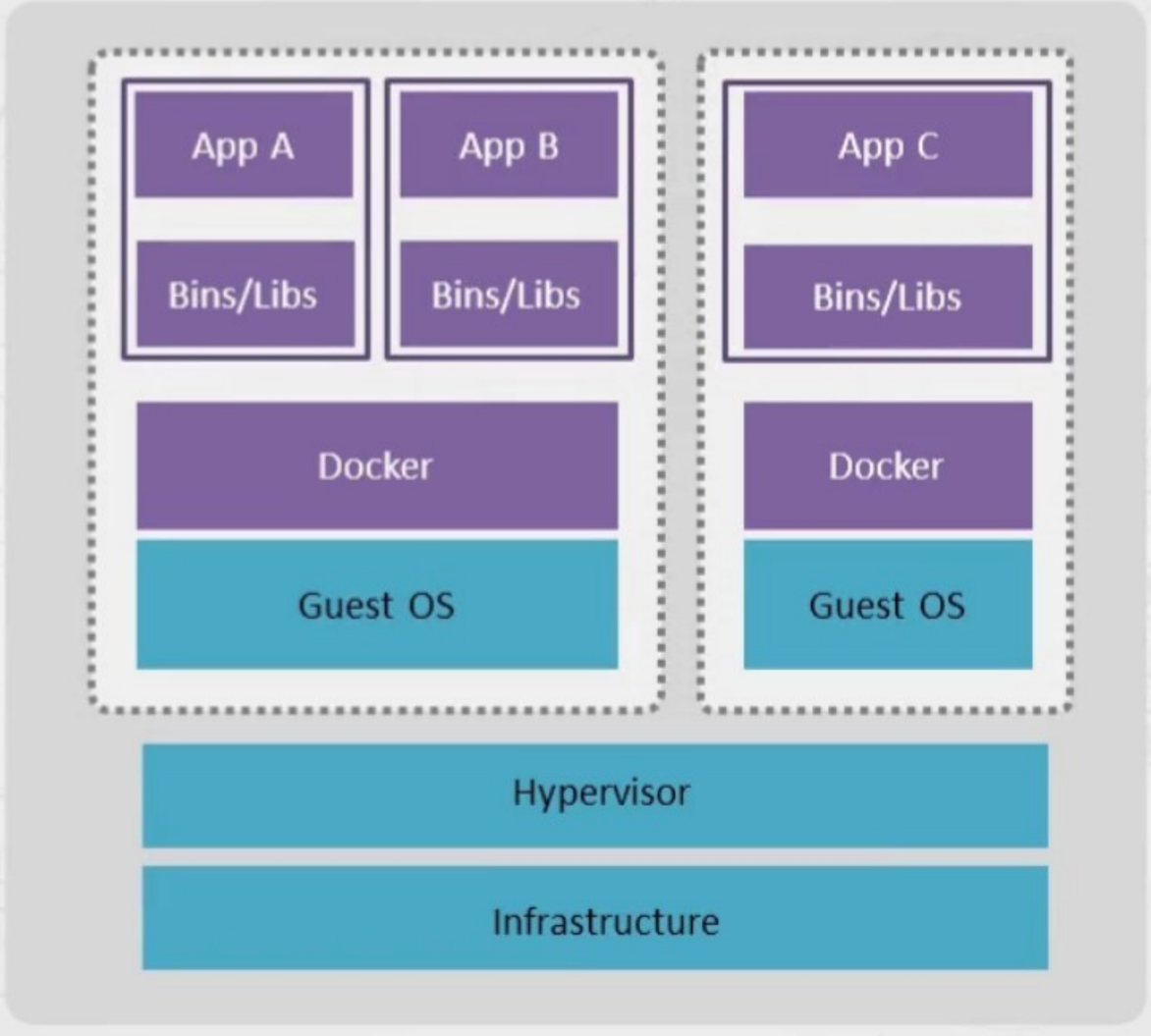
Virtualization and Containers

- Similarities
  - Portability
    - Move VMs or containers between hosts
  - Configurable resource limitations
    - E.g. Storage, CPUs
  - Both separate software from the underlying physical host

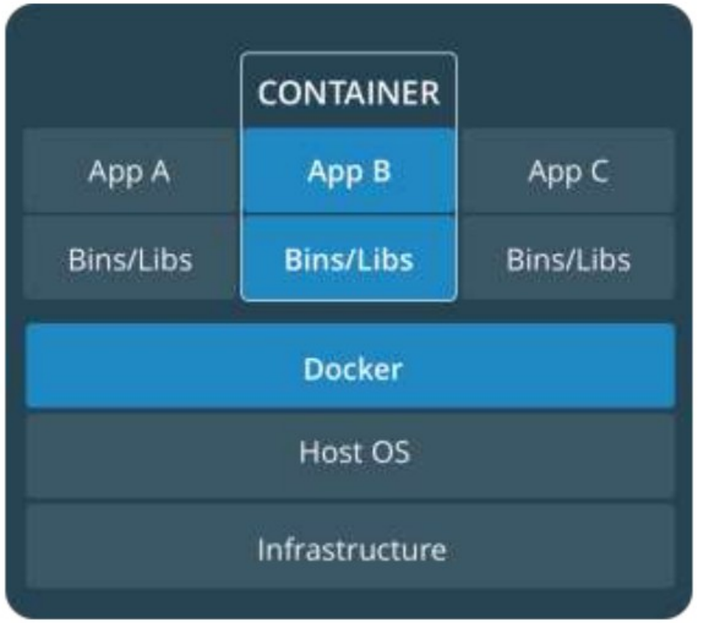
Virtualization and Containers

- Virtual machines provide
  - Complete isolation from physical host
  - Can be configured with guaranteed hardware resources
- Windows Server Containers provide
  - Less isolation than VMs
  - Process (application) isolation
  - The ability to run large numbers of isolated apps on a single host
- Containers can run within a VM
  - These are called Hyper-V Containers
    - As opposed to Windows Server Containers

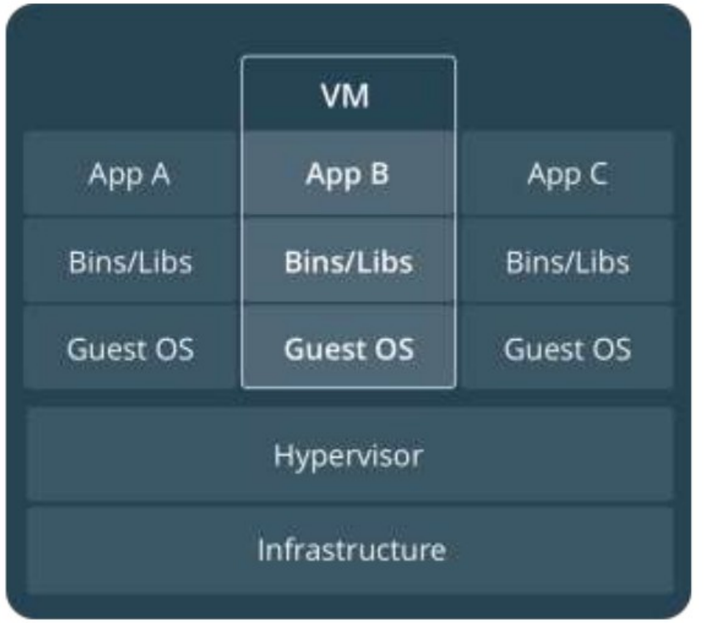
Hyper-V Containers



Comparing Containers and VMs



Containers are an app level construct



VMs are an infrastructure level construct to turn one machine into many servers

Container Use Cases

- Container benefits
  - Portability
    - Windows, Linux, MacOS
  - Rapid startup time
    - E.g. Newly hired developers receive a container created from a relevant base image that already includes required software, settings, and tools
  - Many containers can be based on the same image
  - Developers focus their time on the app, not the supporting infrastructure

Container Use Cases

- Sample Scenario 1: Web App Dev Testing
  - Create an image from pristine web app/database configuration
  - Spin up multiple containers from this image
  - Run multiple read/write tests concurrently
    - Saves from having to keep resetting the environment for each test
    - Uses much less space/resources than VM clones
    - Executes more quickly than using separate VMs

Container Use Cases

- Sample Scenario 2: Legacy Applications
  - Migrate existing legacy apps into containers
  - Container can be moved between
    - Docker hosts
    - VMs
    - On-premises
    - Cloud
  - Cost savings due to reduced app maintenance time

Container Use Cases

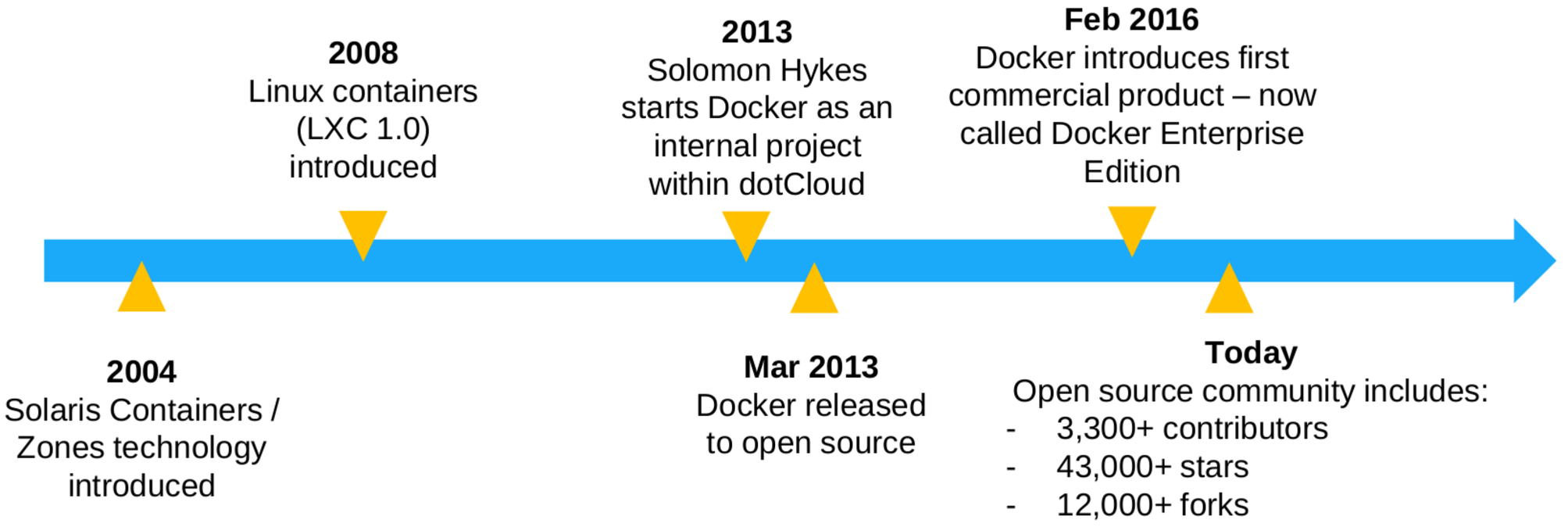
- Sample Scenario 3: App Cloud Migration
  - Migrate on-premises apps to a cloud container
  - Containers are much smaller than VMs
  - Most public cloud providers support Docker containers
    - If not, deploy a Windows Server 2016 VM with Docker
  - Containers are portable
    - This, however, does not mean there are no configuration changes required
      - E.g. Underlying host OS may need updates applied

Docker Overview

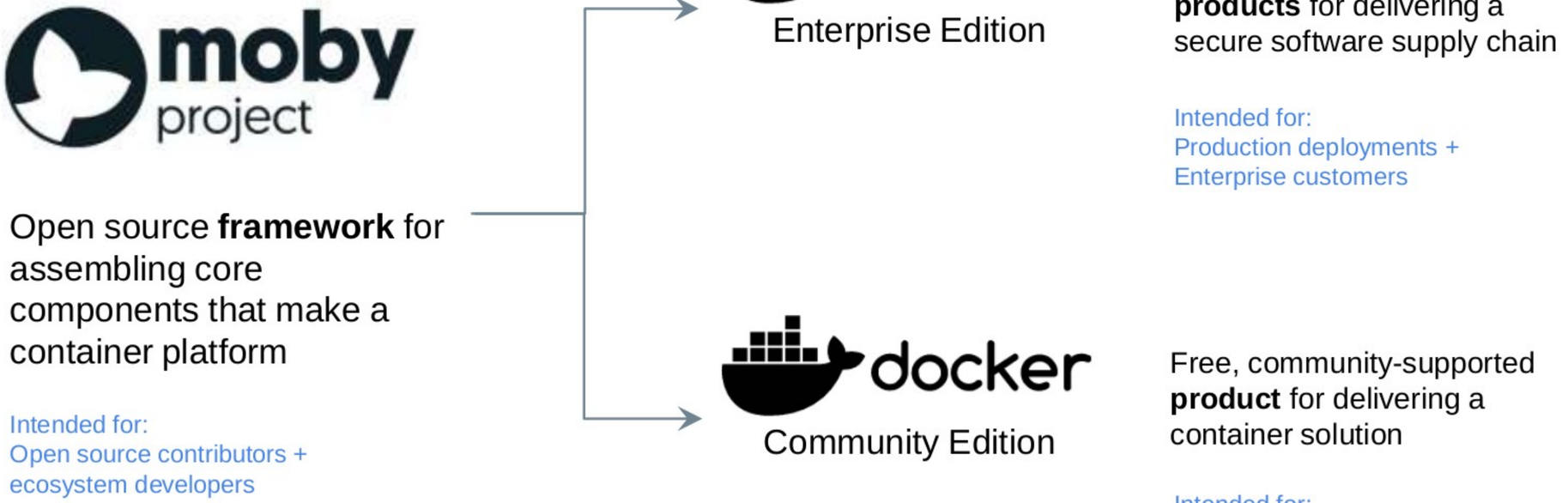
- Docker was introduced in 2013
- Open-source software

Similar Predecessors	
Unix V7	1979
Oracle Solaris Containers	2004
Google Process Containers	2006
Linux Containers (LXC)	2008

History of Docker

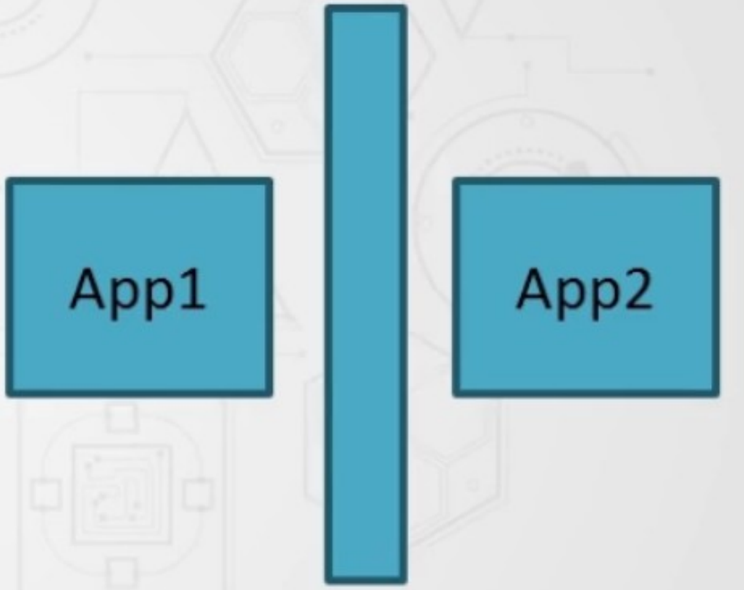


The Docker Family Tree



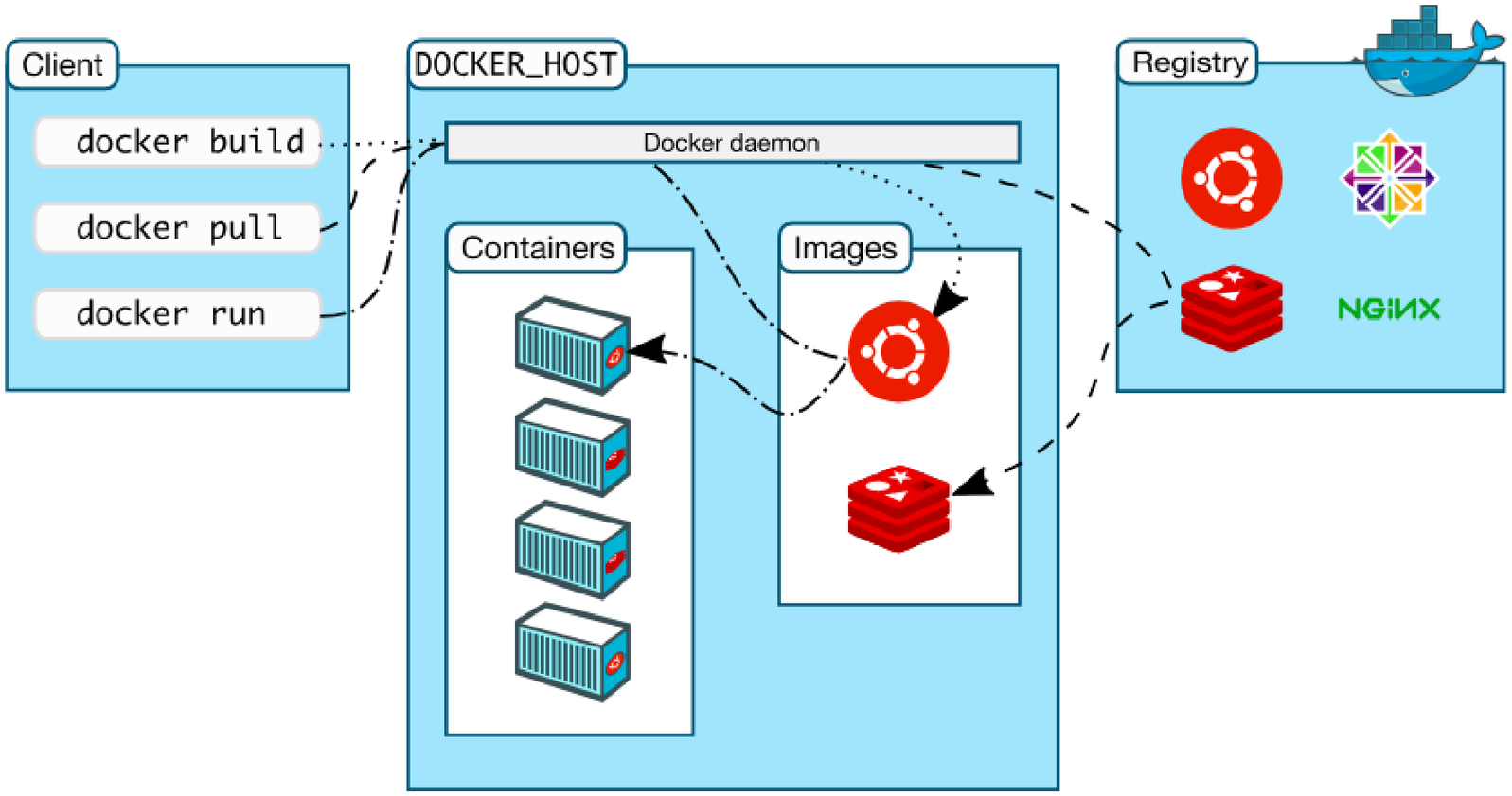
Docker Overview

- Highly appealing to software developers
  - Application isolation environment (a container)
  - Rapid creation and execution of containers
  - Container portability
    - Between hosts
    - Between OSs



Docker architecture

Docker uses a client-server architecture. The Docker *client* talks to the Docker *daemon*, which does the heavy lifting of building, running, and distributing your Docker containers. The Docker client and daemon *can* run on the same system, or you can connect a Docker client to a remote Docker daemon. The Docker client and daemon communicate using a REST API, over UNIX sockets or a network interface.



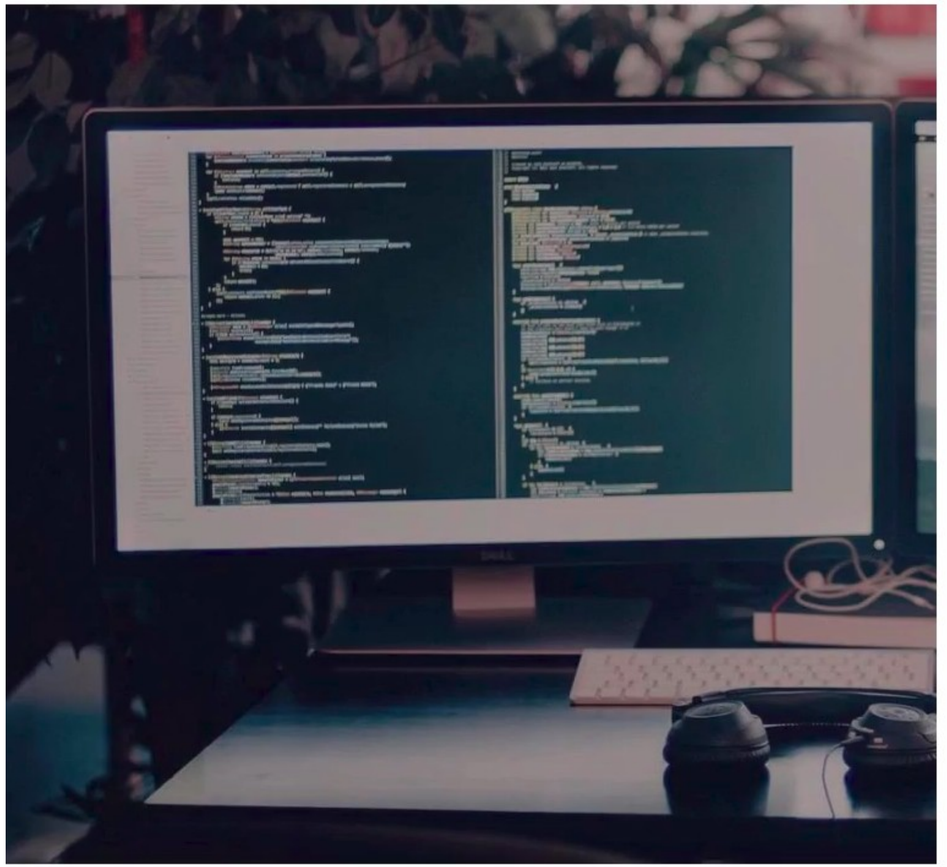
Docker Overview

- Docker components
  - Docker registry
    - Collection of Docker images
  - Docker daemon
    - Service responsible for running containers
      - Runs on Windows 10 and Windows Server 2016 as a service
  - Docker client
    - CLI that talks to the Docker daemon
  - Docker daemon + Docker client = Docker Engine

Docker installation

https://docs.docker.com/install/

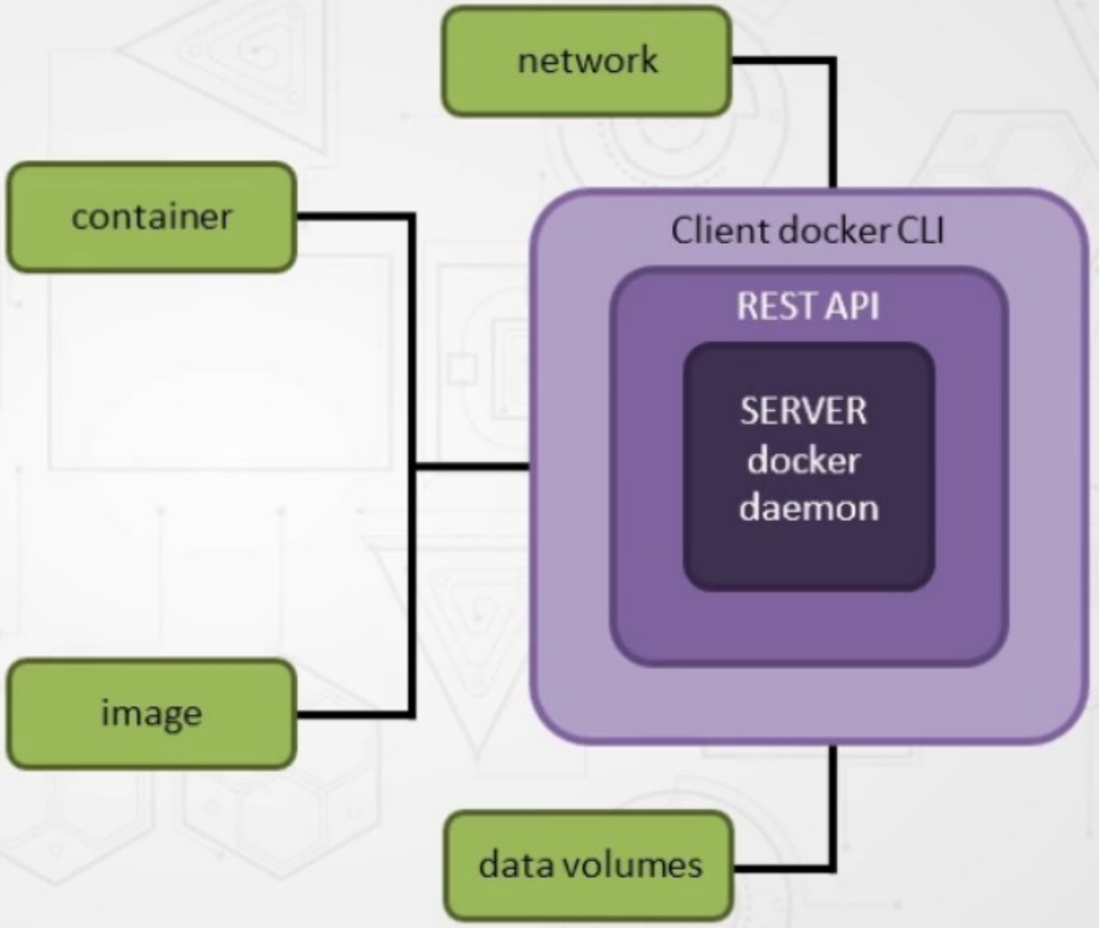
- Ubuntu → https://docs.docker.com/install/linux/docker-ce/ubuntu/
- Mac Os X →https://docs.docker.com/docker-for-mac/install/
- Windows 10 →https://docs.docker.com/docker-for-windows/install/



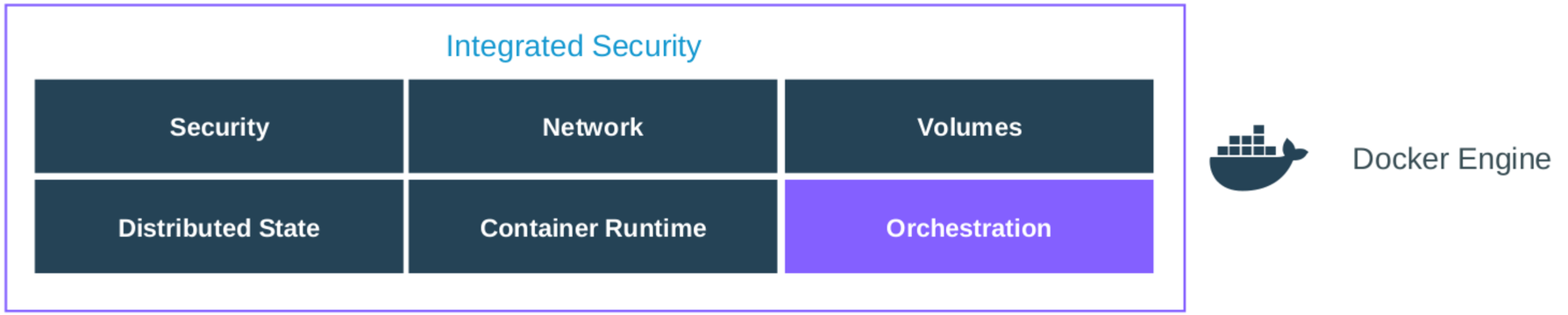
- Docker Images
- Docker Container
- Docker Registry
- Docker Swarm
- Docker Hub
- Docker Compose
- Docker File
- Docker Daemon

Docker Overview

- Docker Engine

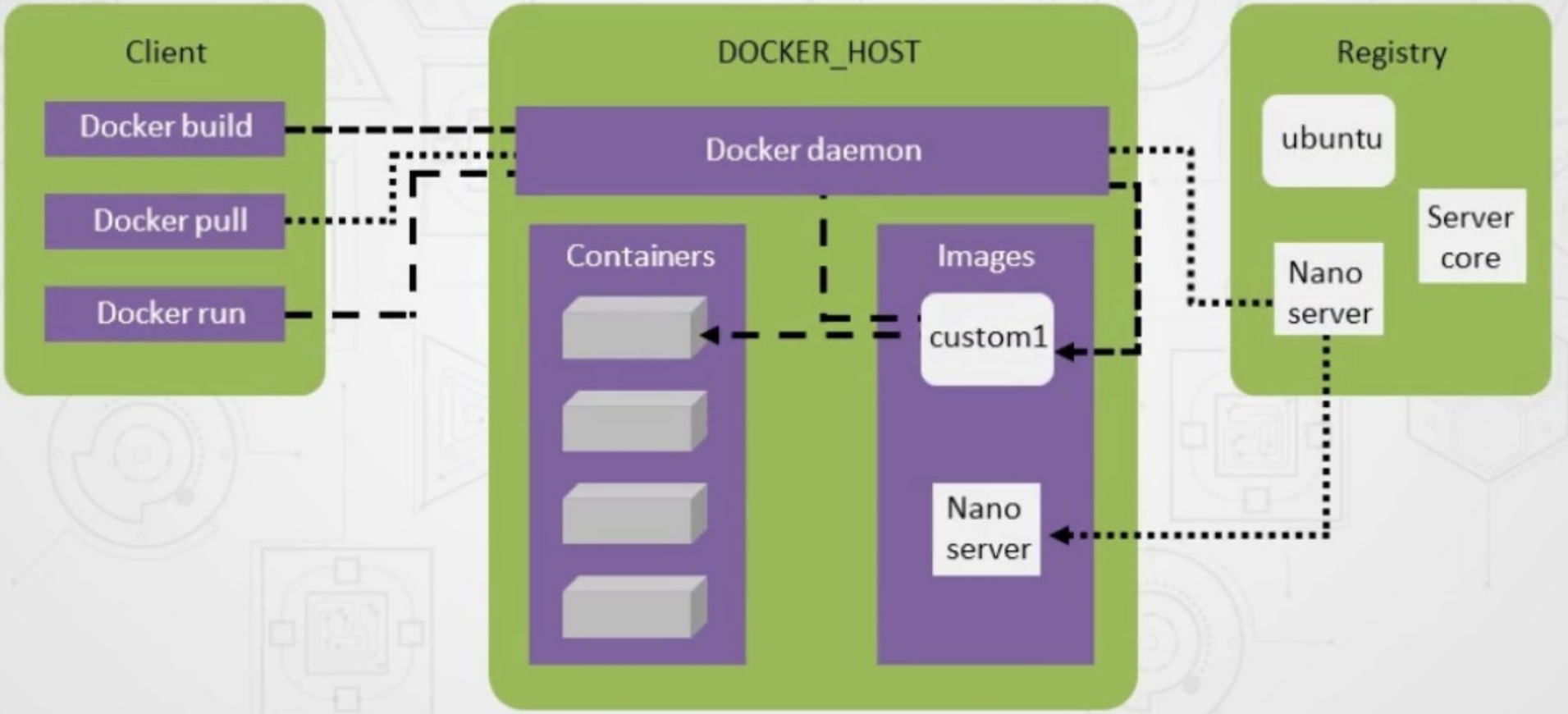


Foundation: Docker Engine

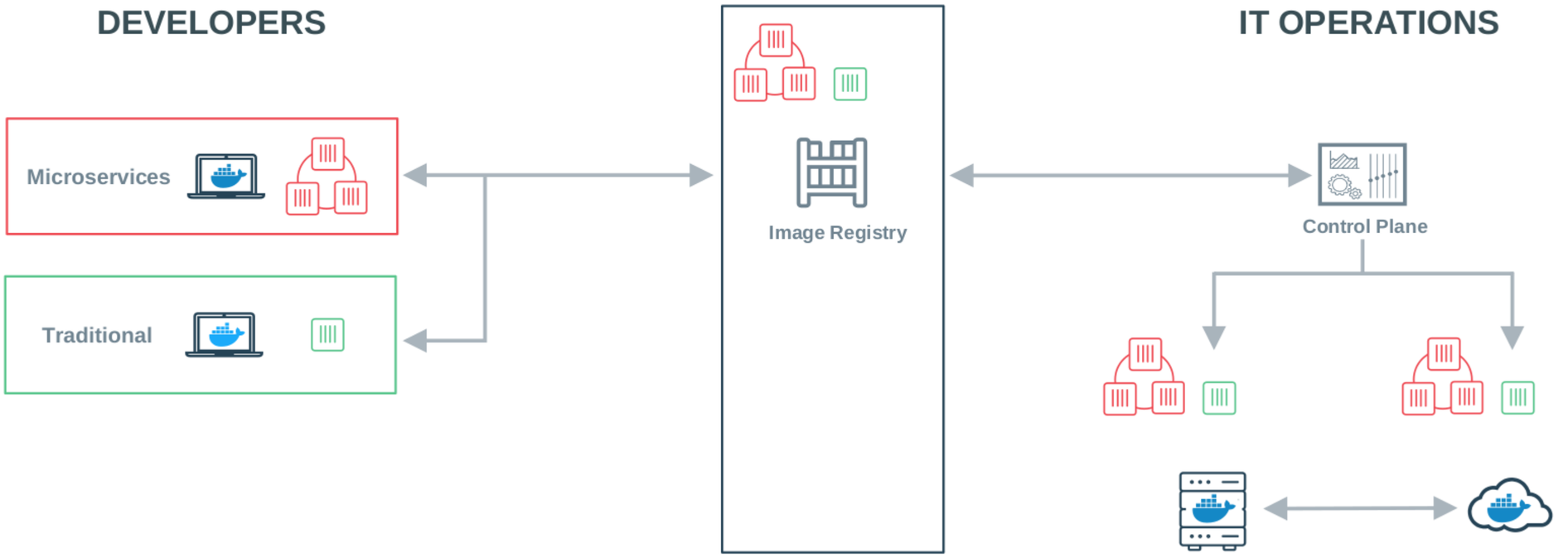


Docker Overview

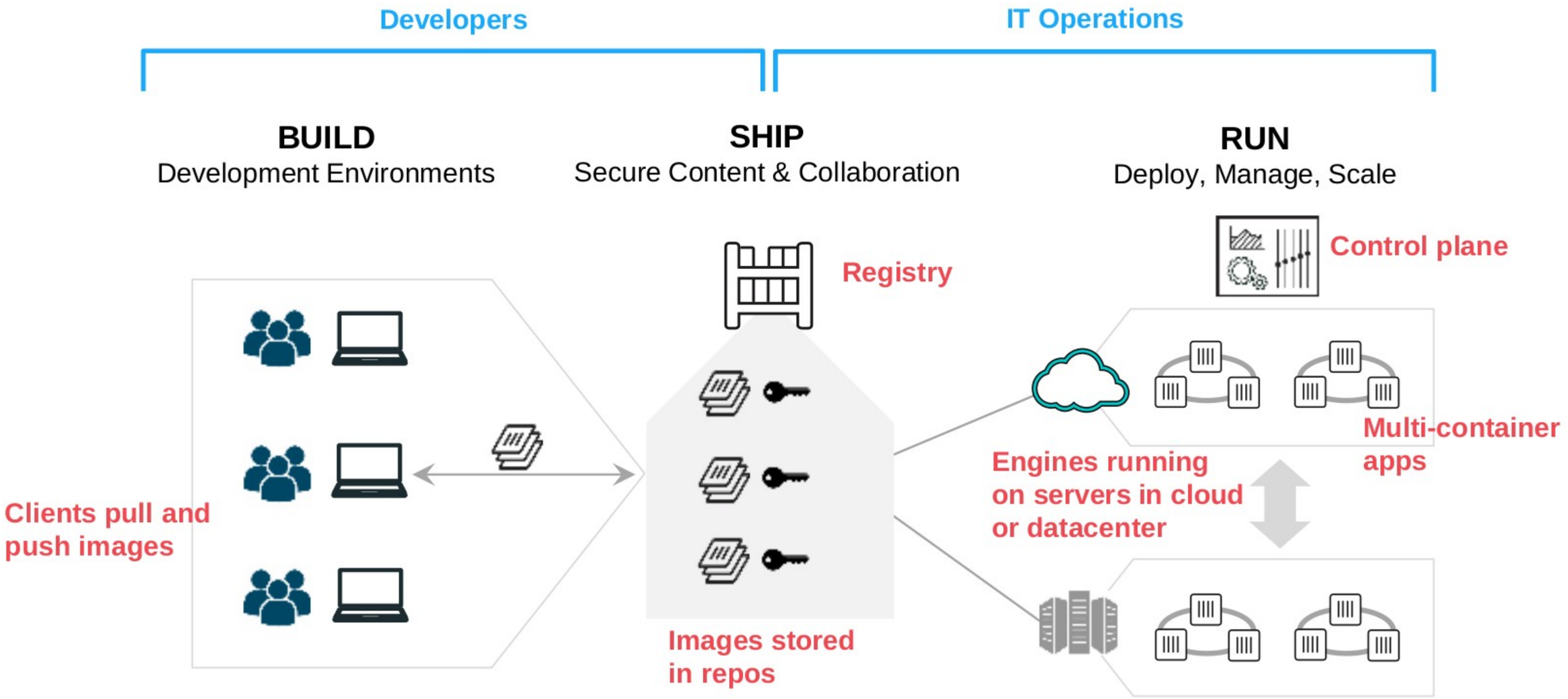
- Docker Ecosystem



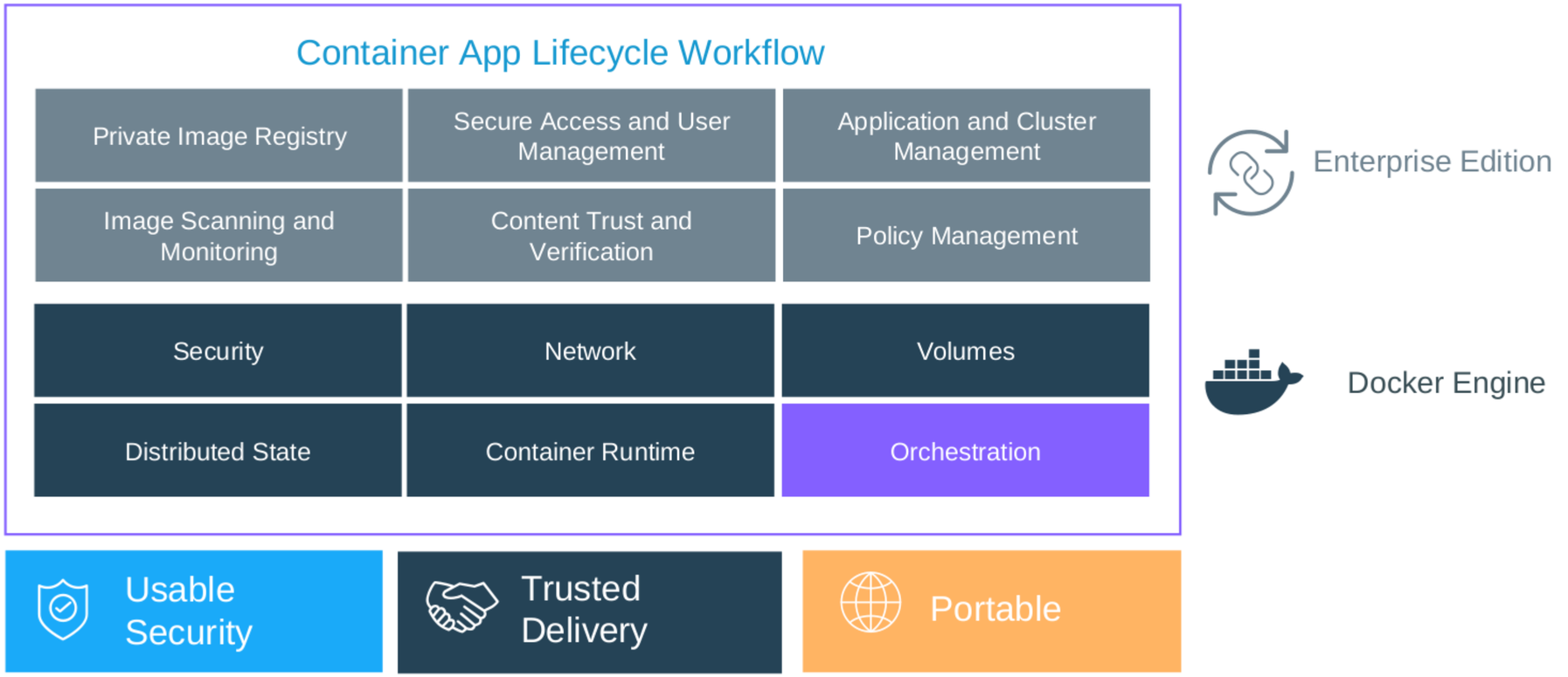
Building a Software Supply Chain



Containers as a Service

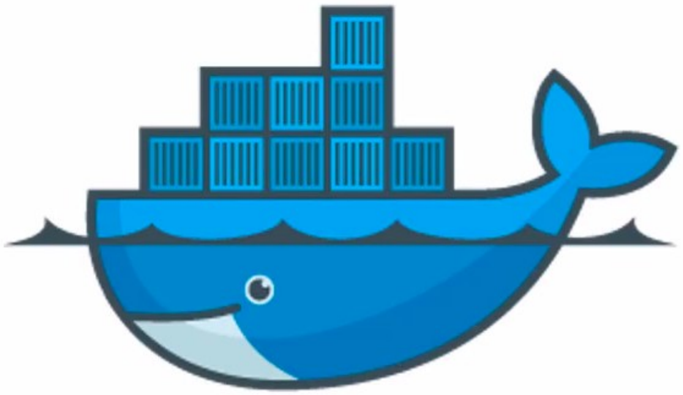


Building a Secure Supply Chain



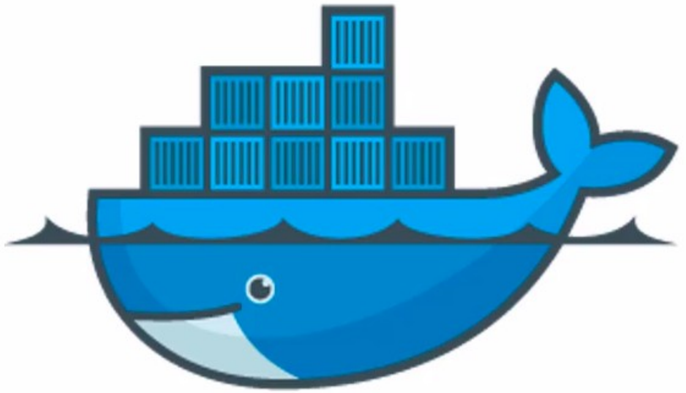
Docker Container

Docker container is a standardized unit of software with all the dependencies. Each container is based on a Docker image.



Docker Daemon

The Docker daemon, or engine is a thin layer between the containers and the Linux kernel. The Docker daemon is the persistent runtime environment that manages application containers.



Command	Description
docker start	Starts a container
docker stop	Stops a running container
docker run	Creates and starts a container
docker attach	Connects to a running container
docker ps	Lists running containers
docker exec	Execute a command in a running container
docker images	Shows all the available images
docker search	Searches registry for image
docker pull	Pulls an image from registry

Dockerfiles

- Dockerfiles are used to build Docker images
  - "docker build" command
    - Builds Docker image from the Dockerfile and context
  - Context
    - Set of files within the local file system
    - Do **not** use C:\ or / as the context!
      - Recursion
    - Git repository

Dockerfiles

- Dockerfile
  - It is a text file
    - Normally named Dockerfile
    - Normally exists in the root of the context
      - Otherwise, specify path when using "docker build –f <path to Dockerfile>"
    - Contains directives for images
  - Dockerfile directives
    - Each one creates a "layer" within the image
    - Rebuilding an image applies only to changed layers

Dockerfiles

Dockerfile Syntax	
Backslash (\)	escape character (spanning multiple lines)
Backtick (`)	escape character (spanning multiple lines)
FROM	Base image from which new image will be created
COPY	Copy folders or files to container filesystem
ADD	Similar to COPY; can also pull items from a URL

Dockerfiles

Dockerfile Syntax	
WORKDIR	Sets the working directory
CMD	Runs a command when container is launched
RUN	Runs commands whose results become a part of the image

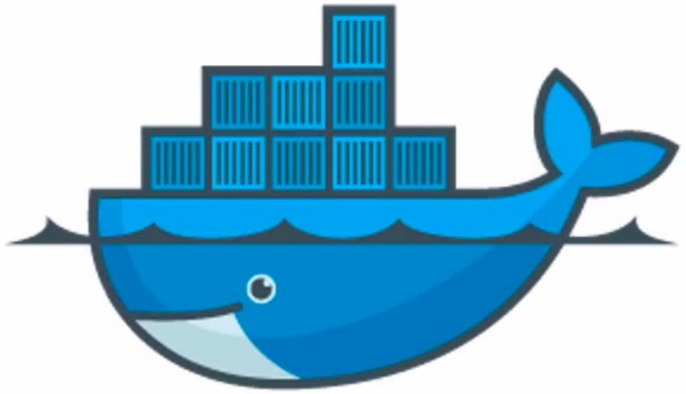
Dockerfiles

- .dockerignore file
  - Normally in the context root directory
  - Is not required
  - Used to exclude files and directories
    - Uses pattern matching
    - Useful to prevent large files from being added to Docker images
  - Consider how the .dockerignore file is created
    - PowerShell redirected output in UTF-16; encoding is problematic
    - In Windows, use a period (.) as the filename suffix; otherwise, file creation fails due to the leading dot

Dockerfiles

.dockerignore Syntax	
#	Comment
*	One or more characters wildcard
?	One character wildcard
[ ]	Range of characters
!	Exception (not)

Docker Images

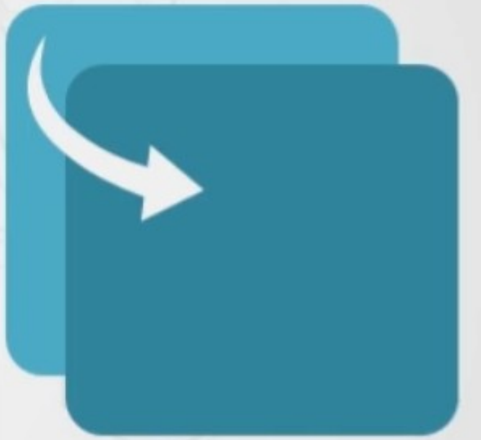


Docker images are templates or the basis of containers. You can deploy these images to run your container on top of the Docker engine.

A container always starts with and is an instantiation of that image. An image is a static specification of what the container should be in runtime. Docker images contain read-only layers, which means that once an image is created it is never modified.

Docker Images

- Used as a basis to create containers
  - Serves as a template
- Custom and public images
  - Custom images require the creation of a Dockerfile
  - Public images are pulled from a repository
- Existing images can be used to create custom images
  - Add or remove components as required



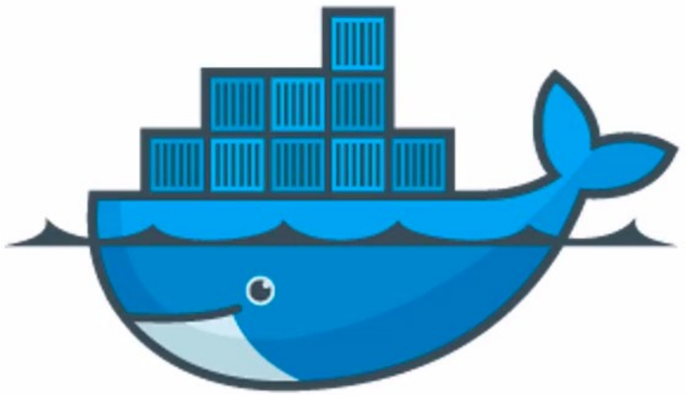
Docker Images

- Images are built using the "docker build" command
  - Requires a Dockerfile with the correct syntax
- A running image is called a container
  - "docker run --name container1 -it sample_image“
  - Running containers are assigned a unique ID
  - Multiple containers from the same image can be run concurrently
- List images
  - "docker images"

Docker Images

- Common images
  - nanoserver
  - windowsservercore
  - iis
  - mssql-server-windows-express
  - centos
  - debian
  - dotnet

Docker Registry



Docker registry is a searchable catalog or registry application which lets us distribute and find Docker images. It is where container images are published and stored.

A registry can be remote or on premises. It can be public or private, or a set of users.

Docker Trusted Registry

- Called a DTR
- Docker image
  - Software, tools, and settings for running a container
- Registry
  - Hosts Docker images
- Repository
  - Related images
  - Different versions of the same image
  - Each image uses unique tag identifiers

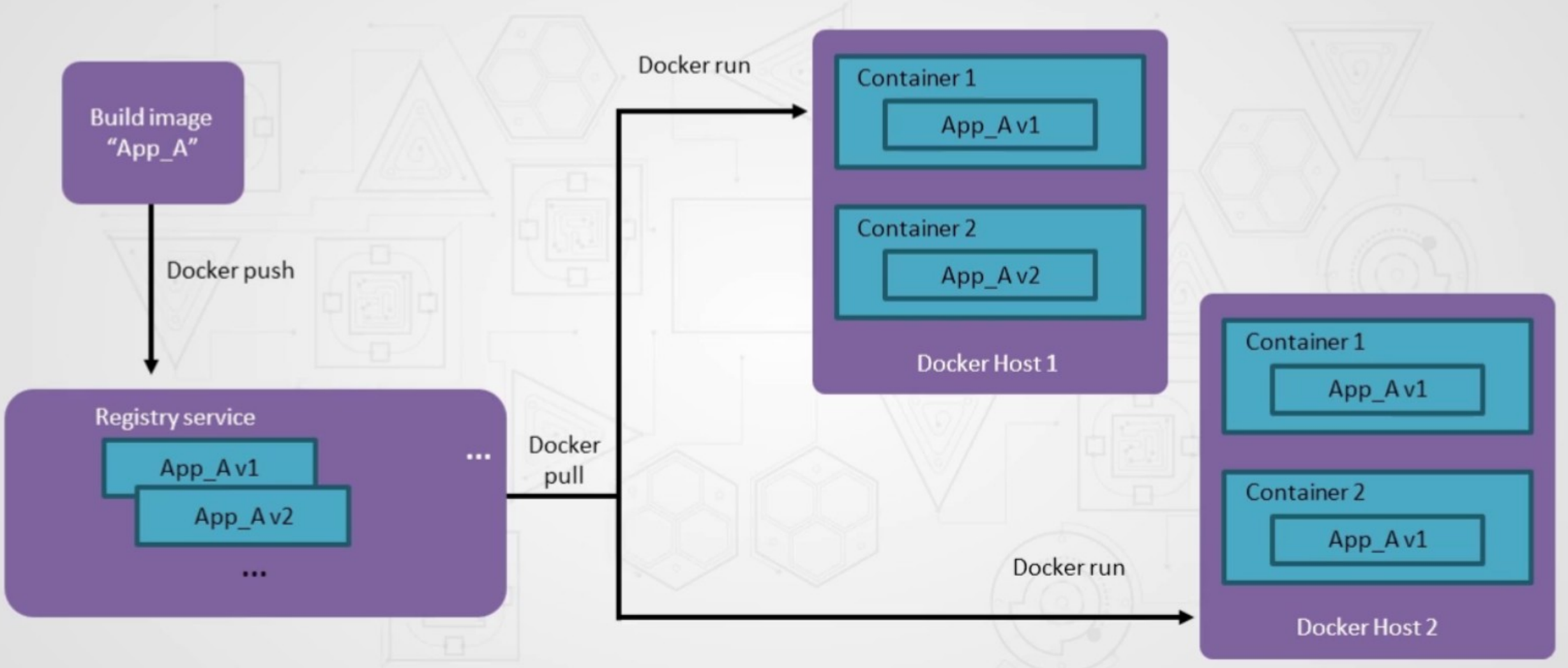
Docker Trusted Registry

- Docker Hub
- Docker Cloud
- Google Container Registry
- Amazon EC2 Container Registry
- Azure Container Registry
- VMware Harbor

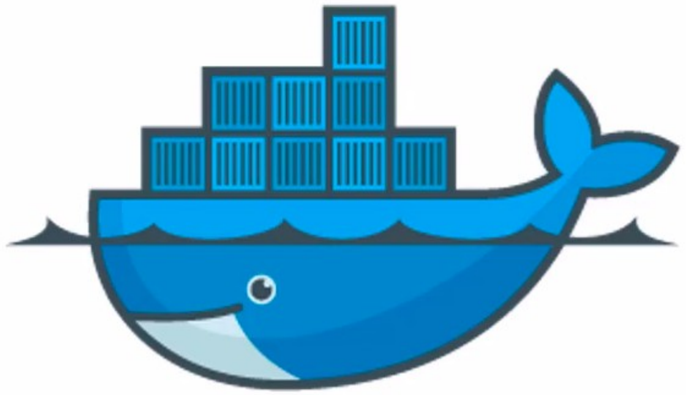
Docker Trusted Registry

- Docker pull
  - Bring down a Docker image from a repository or registry
- Docker push
  - Place a Docker image in a repository or registry
- Docker search
  - Find public images on Docker Hub

Docker Trusted Registry



Docker Hub



Docker Hub is a cloud hosted service from Docker that provides registry capabilities for public and private content. It provides image discovery, distribution, and collaboration , in addition to workflow support.

Docker Hub also has a set of official certified images from known software publishers such as Canonical, Red Hat, and MongoDB. You can use these official images as a basis for building your own images or applications.

Docker Hub

- Docker Hub is a service provided by Docker for finding and sharing container images with your team. It provides the following major features:
- Repositories: Push and pull container images.
- Teams & Organizations: Manage access to private repositories of container images.
- Official Images: Pull and use high-quality container images provided by Docker.
- Publisher Images: Pull and use high-quality container images provided by external vendors. Certified images also include support and guarantee compatibility with Docker Enterprise.
- Builds: Automatically build container images from GitHub and Bitbucket and push them to Docker Hub
- Webhooks: Trigger actions after a successful push to a repository to integrate Docker Hub with other services.

Docker Hub

Step 1: Sign up for Docker Hub

Start by creating an account.

Step 2: Sign in for Docker Hub

jvherrera@juanvi-HP-ZBook-14u-G5:~$ docker login -u juanviz

Password:

WARNING! Your password will be stored unencrypted in /home/jvherrera/.docker/config.json.

Configure a credential helper to remove this warning. See

https://docs.docker.com/engine/reference/commandline/login/#credentials-store

Login Succeeded

For real applications IT users and app teams need more sophisticated tools.

- Docker supplies two such tools:
- **Docker Compose**
- **Docker Swarm Mode**

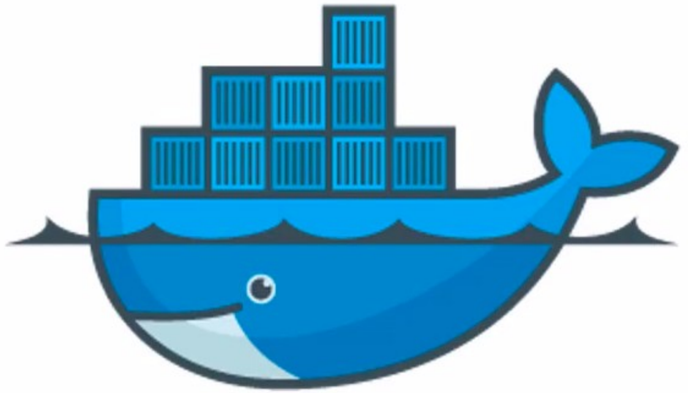
The two tools have some similarities but some important differences:

Compose is used to control multiple containers on a single system. Much like the Dockerfile we looked at to build an image, there is a text file that describes the application: which images to use, how many instances, the network connections, etc. But Compose only runs on a single system so while it is useful, we are going to skip Compose1 and go straight to Docker Swarm Mode.

Swarm Mode tells Docker that you will be running many Docker engines and you want to coordinate operations across all of them. Swarm mode combines the ability to not only define the application architecture, like Compose, but to define and maintain high availability levels, scaling, load balancing, and more. With all this functionality, Swarm mode is used more often in production environments than it's more simplistic cousin, Compose.

Docker Compose

Docker Compose is a tool for defining and running complex applications with Docker. With compose, you define a multi-container application in a single file, then spin your application up in a single command which does everything that needs to be done to get it running.



Docker Compose

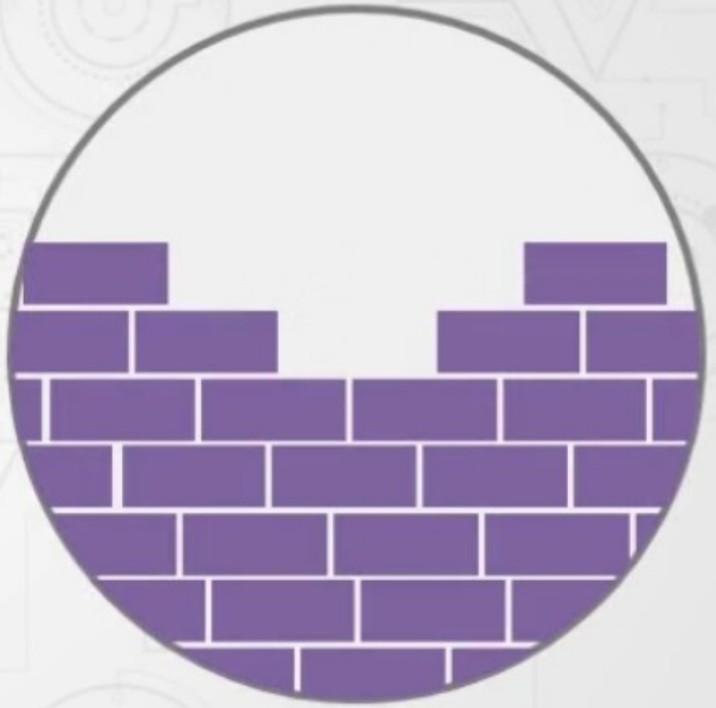
- Configure and run Docker applications
  - Specifically, multi-container applications
- Multi-container application potential components
  - Web servers
  - Database servers
  - Message queues
- Service configuration
  - YAML file

Docker Compose

- Benefits
  - Restarting an unchanged service uses the existing container
  - Variables can be configured to support multiple environments
  - Traffic targeted to the multi-container service is round-robined

Docker Compose

- Usage steps
  1. Configure a Dockerfile
  2. Configure docker-compose.yml
  3. Run the "docker-compose up" command



Docker-compose.yml Part 1

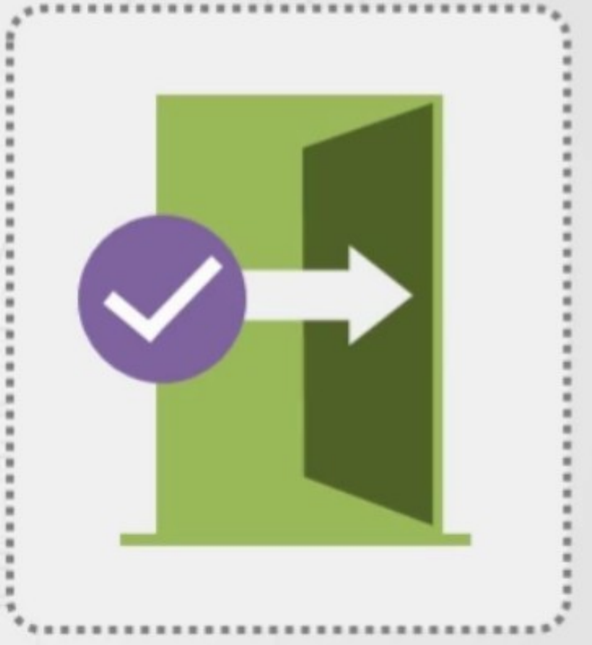
```
services:
  db:
    image: microsoft/mssql-server-windows-express
    environment:
      sa_password: "Pa$$w0rd"
  web:
    build:
      context: .
      dockerfile: Dockerfile
    environment:

…
```

Docker-compose.yml Part 2

```
"Data:DefaultConnection:ConnectionString=dbServer=db,1433;Database=DB1;User
Id=sa;Password=Pa$$w0rd;MultipleActiveResultSets=True"
    depends_on:
      - "db"
    ports:
      - "5000:5000"
networks:
  default:
    external:
      name: ext_nat_net1
```

Continuous Integration and Deployment

- Referred to as CI/CD
- Multiple developers
  - Code check-in/check-out
  - Code changes are integrated quickly into the app
- New code changes
  - Can trigger immediate testing on new code changes
    - Testing environment must be the same as production
  - Upon positive results, push current/new image to a Docker registry



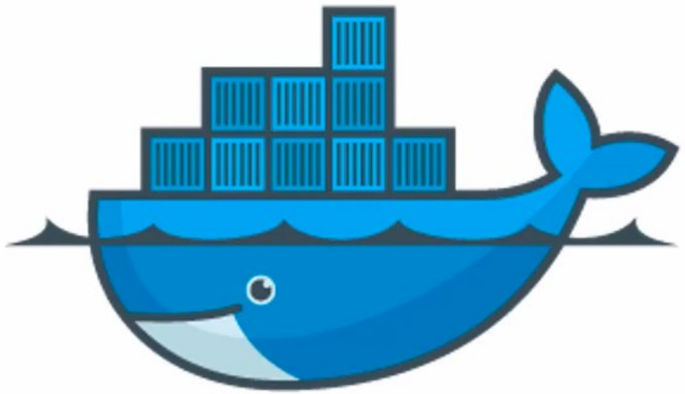
Continuous Integration and Deployment

- CI best practices
  - Use build servers that automatically test code changes
  - Developers should check-in code changes multiple times daily
  - Every change committed to the baseline should be built
  - Automated testing should also include
    - Statistics
    - Reports
    - Generated installation media, such as a .sh or .msi file

Continuous Integration and Deployment

- Common developer CI tools
  - Jenkins
  - Travis
  - TeamCity
  - CircleCI
  - Codeship
  - GitLab CI
  - Visual Studio Team Foundation Server

Docker Swarm



Docker Swarm is native clustering for Docker. It turns a pool of Docker hosts into a single, virtual Docker host. Because Docker Swarm serves the standard Docker API, any tool that already communicates with a Docker daemon can use Swarm to transparently scale to multiple hosts. Supported tools include, but are not limited to, the following:

- Dokku
- Docker Compose
- Docker Machine
- Jenkins

And of course, the Docker client itself is also supported.

Docker Swarm

- Swarm host types
  1. Worker nodes
     - These become swarm members using a join token created by a manager node
     - Actions are directed by manager nodes
     - Execute tasks
  2. Manager nodes
     - Swarm configuration
     - Docker CLI commands are run here
     - A swarm must have at least one of these
     - Monitor and maintain swarm state

Docker Swarm

- What is Swarm?
  - Docker host clustering
  - Container orchestration
  - Container workload scheduling
- Docker hosts are configured to run in swarm mode

Docker Swarm

- Swarm requirements
  - Windows 10 or Windows Server 2016
    - At least two physical or virtual machines for clustering
  - Docker Engine v1.13.0 or later
  - Open ports
    - TCP 2377 for cluster management
    - TCP/UDP 7946 for inter-node communication
    - UDP 4789 for container ingress network traffic

Docker Swarm

3. Create an overlay network
   - "docker network create"
4. Deploy services to the swarm
   - "docker service create"
5. Scale the service
   - "docker service scale"

- List swarm nodes
  - "docker node ls"

Universal Control Plane (UCP)

- Manage
  - Cluster nodes
  - Volumes
  - Networks
  - Images
  - Containers
  - Apps

Universal Control Plane (UCP)

- UCP security
  - Controls cluster management access
  - LDAP
  - Microsoft Active Directory
  - RBAC
  - Integrates with Docker Trusted Registry (DTR)
  - Images can be digitally signed

Storage Overview

By default all files created inside a container are stored on a writable container layer. This means that:

- The data doesn’t persist when that container no longer exists, and it can be difficult to get the data out of the container if another process needs it.
- A container’s writable layer is tightly coupled to the host machine where the container is running. You can’t easily move the data somewhere else.
- Writing into a container’s writable layer requires a storage driver to manage the filesystem. The storage driver provides a union filesystem, using the Linux kernel. This extra abstraction reduces performance as compared to using data volumes, which write directly to the host filesystem.

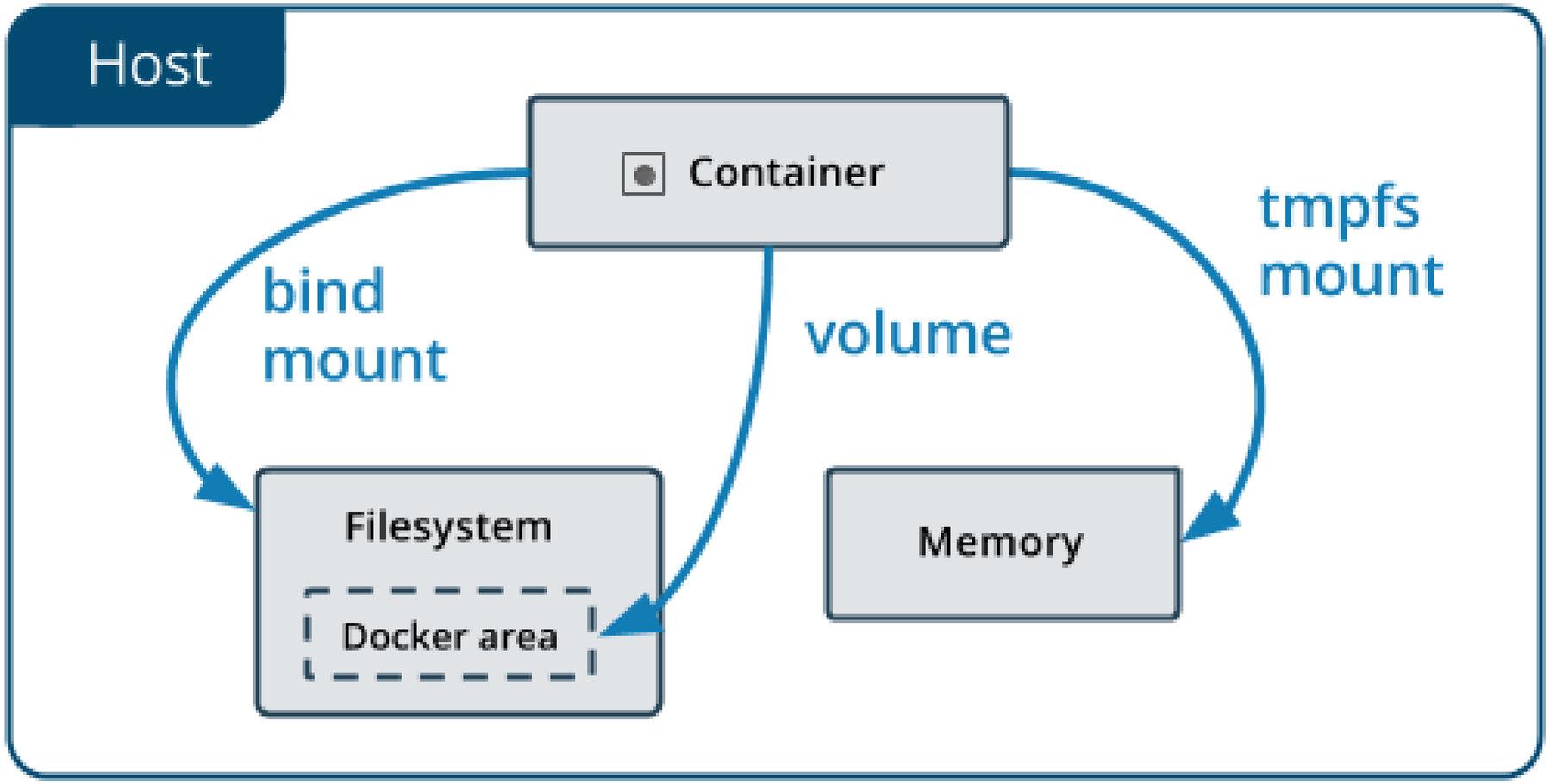
Docker has two options for containers to store files in the host machine, so that the files are persisted even after the container stops: **volumes**, and **bind mounts**.

If you’re running Docker on Linux you can also use a tmpfs mount.

- Volumes are stored in a part of the host filesystem which is managed by Docker (/var/lib/docker/volumes/ on Linux). Non-Docker processes should not modify this part of the filesystem. Volumes are the best way to persist data in Docker.

- Bind mounts may be stored anywhere on the host system. They may even be important system files or directories. Non-Docker processes on the Docker host or a Docker container can modify them at any time.

- tmpfs mounts are stored in the host system’s memory only, and are never written to the host system’s filesystem.


Host
Container
bind
mount
volume
tmpfs
mount
Filesystem
Docker area
Memory

More details about mount types

Volumes:

- Created and managed by Docker. You can create a volume explicitly using the docker volume create command, or Docker can create a volume during container or service creation.
- When you create a volume, it is stored within a directory on the Docker host.
- When you mount the volume into a container, this directory is what is mounted into the container.
- This is similar to the way that bind mounts work, except that volumes are managed by Docker and are isolated from the core functionality of the host machine.

More details about mount types

- **A given volume can be mounted into multiple containers simultaneously**. When no running container is using a volume, the volume is still available to Docker and is not removed automatically. **You can remove unused volumes using docker volume prune.**

- When you mount a volume, it may be named or anonymous. Anonymous volumes are not given an explicit name when they are first mounted into a container, so Docker gives them a random name that is guaranteed to be unique within a given Docker host. Besides the name, named and anonymous volumes behave in the same ways.

- Volumes also support the use of volume drivers, which allow you to store your data on remote hosts or cloud providers, among other possibilities.

More details about mount types

Bind mounts: Available since the early days of Docker.

- Bind mounts have limited functionality compared to volumes.
- When you use a bind mount, a file or directory on the host machine is mounted into a container.
- The file or directory is referenced by its full path on the host machine. The file or directory does not need to exist on the Docker host already.
- It is created on demand if it does not yet exist. Bind mounts are very performant, but they rely on the host machine’s filesystem having a specific directory structure available.
- If you are developing new Docker applications, consider using named volumes instead. You can’t use Docker CLI commands to directly manage bind mounts

More details about mount types

tmpfs mounts:

- A tmpfs mount is not persisted on disk, either on the Docker host or within a container.
- It can be used by a container during the lifetime of the container, to store non-persistent state or sensitive information.
- For instance, internally, swarm services use tmpfs mounts to mount secrets into a service’s containers.

Good use cases for volumes

Volumes are the preferred way to persist data in Docker containers and services. Some use cases for volumes include:

- **Sharing data among multiple running containers**. If you don’t explicitly create it, a volume is created the first time it is mounted into a container.

- **When that container stops or is removed, the volume still exists**. Multiple containers can mount the same volume simultaneously, either read-write or read-only. Volumes are only removed when you explicitly remove them.

- **When the Docker host is not guaranteed to have a given directory or file structure.** Volumes help you decouple the configuration of the Docker host from the container runtime.

- **When you want to store your container’s data on a remote host or a cloud provider, rather than locally.**
- **When you need to back up, restore, or migrate data from one Docker host to another,** volumes are a better choice. You can stop containers using the volume, then back up the volume’s directory (such as /var/lib/docker/volumes/<volume-name>).

Good use cases for bind mounts

In general, you should use volumes where possible. Bind mounts are appropriate for the following types of use case:

- **Sharing configuration files from the host machine to containers.** This is how Docker provides DNS resolution to containers by default, by mounting /etc/resolv.conf from the host machine into each container.

- **Sharing source code or build artifacts between a development environment on the Docker host and a container**. For instance, you may mount a Maven target/ directory into a container, and each time you build the Maven project on the Docker host, the container gets access to the rebuilt artifacts.

- **When the file or directory structure of the Docker host is guaranteed to be consistent** with the bind mounts the containers require.

Good use cases for tmpfs mounts

tmpfs mounts are best used for cases when you do not want the data to persist either on the host machine or within the container.

This may be for security reasons or to protect the performance of the container when your application needs to write a large volume of non-persistent state data.

Tips for using bind mounts or volumes

If you use either bind mounts or volumes, keep the following in mind:

- If you mount an empty volume into a directory in the container in which files or directories exist, these files or directories are propagated (copied) into the volume. Similarly, if you start a container and specify a volume which does not already exist, an empty volume is created for you. This is a good way to pre-populate data that another container needs.

- If you mount a bind mount or non-empty volume into a directory in the container in which some files or directories exist, these files or directories are obscured by the mount, just as if you saved files into /mnt on a Linux host and then mounted a USB drive into /mnt. The contents of /mnt would be obscured by the contents of the USB drive until the USB drive were unmounted. The obscured files are not removed or altered, but are not accessible while the bind mount or volume is mounted.

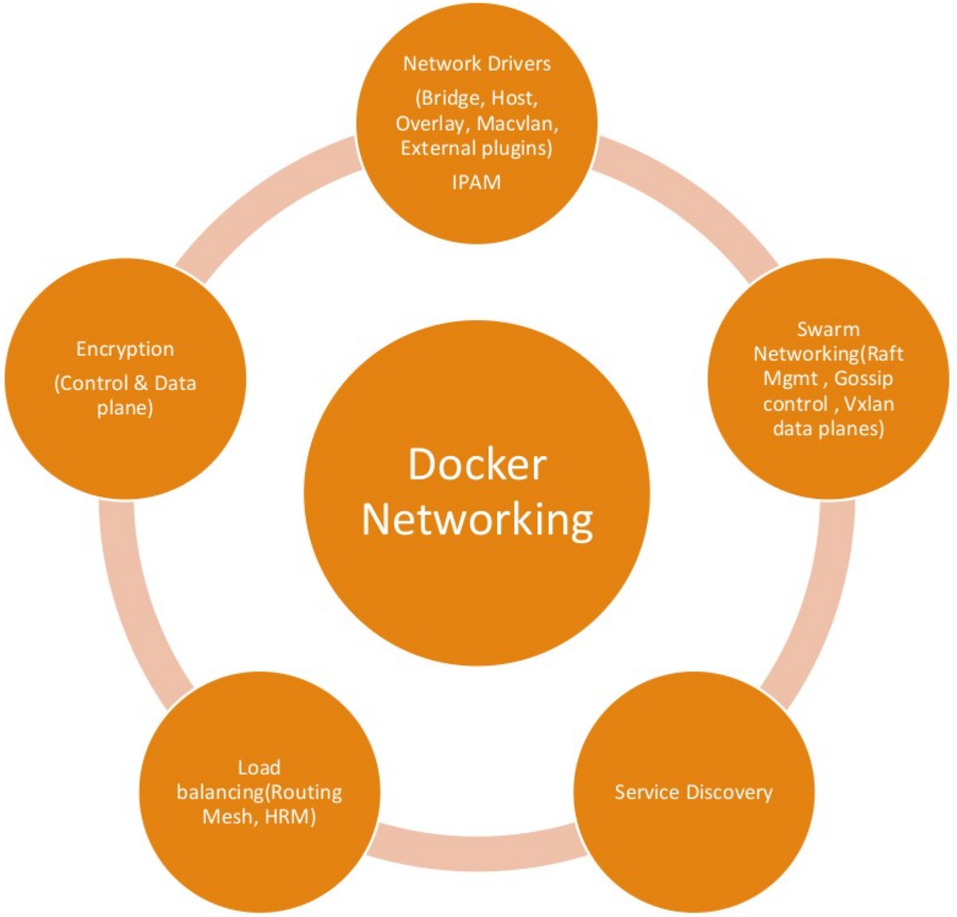
Why we need Container Networking?

- Containers need to talk to external world.
- Reach Containers from external world to use the service that Containers provides.
- Allows Containers to talk to host machine.
- Inter-container connectivity in same host and across hosts.
- Discover services provided by containers automatically.
- Load balance traffic between different containers in a service
- Provide secure multi-tenant services

Compare Container Networking with VM Networking

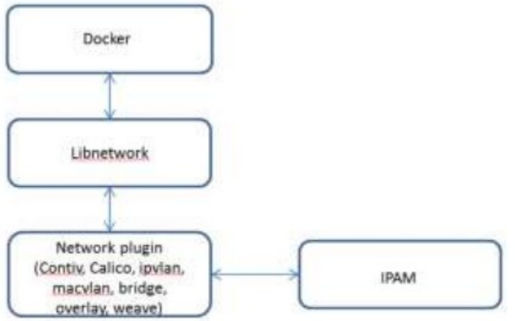
Feature	Container	VM
Isolation	Network isolation achieved using Network namespace.	Separate networking stack per VM
Service	Typically, Services gets separate IP and maps to multiple containers	Multiple services runs in a single VM
Service Discovery and Load balancing	Microservices done as Containers puts more emphasis on integrated Service discovery	Service Discovery and Load balancing typically done outside
Scale	As Container scale on a single host can run to hundreds, host networking has to be very scalable.	Host networking scale needs are not as high
Implementation	Docker Engine and Linux bridge	Hypervisor and Linux/OVS bridge

Docker Networking components



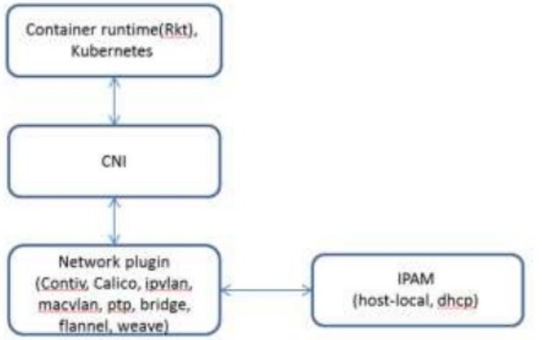
CNI and CNM – Standards for Container Networking

CNM



- Project started by Docker.
- Keep networking as a library separate from the Container runtime.
- Networking implementation will be done as a plugin implemented by drivers.
- IP address assignment for the Containers is done using local IPAM drivers and plugins.
- Supported local drivers are bridge, overlay, macvlan, ipvlan. Supported remote drivers are Weave, Calico, Contiv etc.

CNI



- Project started by CoreOS. Used by Cloudfoundry, Mesos and Kubernetes.
- The CNI interface calls the API of the CNI plugin to set up Container networking.
- The CNI plugin calls the IPAM plugin to set up the IP address for the container.
- Available CNI plugins are Bridge, macvlan, ipvlan, and ptp. Available IPAM plugins are host-local and DHCP.
- External CNI plugins examples – Flannel, Weave, Contiv etc

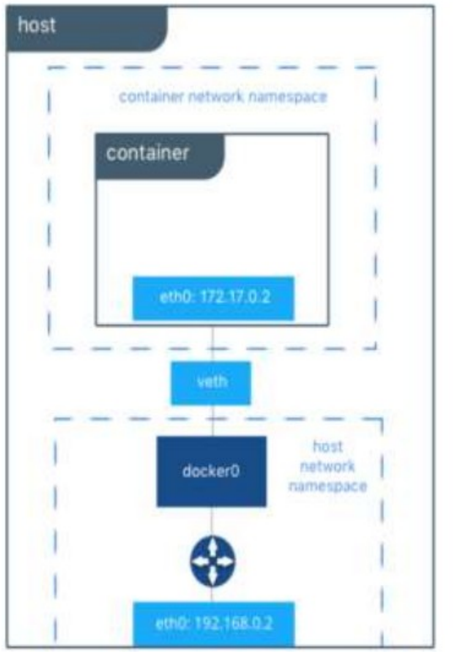
Compare Docker Network driver types

Driver/ Features	Bridge	User defined bridge	Host	Overlay	Macvlan/ipvlan
Connectivity	Same host	Same host	Same host	Multi-host	Multi-host
Service Discovery and DNS	Using “links”. DNS using /etc/hosts	Done using DNS server in Docker engine	Done using DNS server in Docker engine	Done using DNS server in Docker engine	Done using DNS server in Docker engine
External connectivity	NAT	NAT	Use Host gateway	No external connectivity	Uses underlay gateway
Namespace	Separate	Separate	Same as host	Separate	Separate
Swarm mode [1]	No support yet	No support yet	No support yet	Supported	No support yet
Encapsulation	No double encap	No double encap	No double encap	Double encap using Vxlan	No double encap
Application	North, South external access	North, South external access	Need full networking control, isolation not needed	Container connectivity across hosts	Containers needing direct underlay networking

Bridge Driver

- Used by “docker0” bridge and user-defined bridges.
- “docker0” bridge is created by default. User has the choice to change “docker0” bridge options by specifying them in Docker daemon config.
- User-defined bridges can be created using “docker network create” with “bridge” driver.
- Used for connectivity between containers in same host and for external North<->South connectivity.
- Services running inside Containers can be exposed by NAT/port forwarding.
- External access is provided by masquerading.

docker run -d -p 8080:80 --network bridge --name web nginx

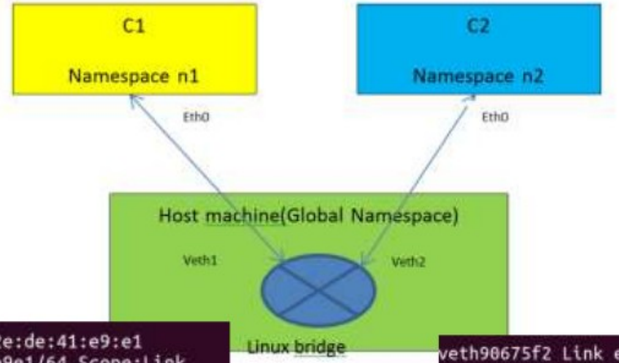


Picture from Docker white paper

Docker Container Networking – Bridge driver

```
root@bd212dd98815:/# ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:ac:11:2a:02
          inet addr:172.17.42.2  Bcast:0.0.0.0  Mask:255.255.255.0
          inet6 addr: fe80::42:acff:fe11:2a02/64 Scope:Link
          UP BROADCAST RUNNING  MTU:1500  Metric:1
          RX packets:6 errors:0 dropped:0 overruns:0 frame:0
          TX packets:7 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:508 (508.0 B)  TX bytes:598 (598.0 B)
```

```
eth0      Link encap:Ethernet  HWaddr 02:42:ac:11:2a:03
          inet addr:172.17.42.3  Bcast:0.0.0.0  Mask:255.255.255.0
          inet6 addr: fe80::42:acff:fe11:2a03/64 Scope:Link
          UP BROADCAST RUNNING  MTU:1500  Metric:1
          RX packets:3 errors:0 dropped:0 overruns:0 frame:0
          TX packets:4 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:258 (258.0 B)  TX bytes:348 (348.0 B)
```




```
veth3f5b988 Link encap:Ethernet  HWaddr 8e:2e:de:41:e9:e1
          inet6 addr: fe80::8c2e:deff:fe41:e9e1/64 Scope:Link
          UP BROADCAST RUNNING  MTU:1500  Metric:1
          RX packets:9 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:738 (738.0 B)  TX bytes:648 (648.0 B)
```

```
veth90675f2 Link encap:Ethernet  HWaddr 9a:bc:0a:4a:fd:b4
          inet6 addr: fe80::98bc:aff:fe4a:fdb4/64 Scope:Link
          UP BROADCAST RUNNING  MTU:1500  Metric:1
          RX packets:9 errors:0 dropped:0 overruns:0 frame:0
          TX packets:17 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:738 (738.0 B)  TX bytes:1386 (1.3 KB)
```

```
sreeni@ubuntu:~$ sudo brctl show
bridge name     bridge id               STP enabled     interfaces
docker0         8000.8e2ede41e9e1       no              veth3f5b988
                                                        veth90675f2

sreeni@ubuntu:~$
```

```
docker0   Link encap:Ethernet  HWaddr 26:90:8f:3a:1f:d5
          inet addr:172.17.42.1  Bcast:0.0.0.0  Mask:255.255.255.0
          UP BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

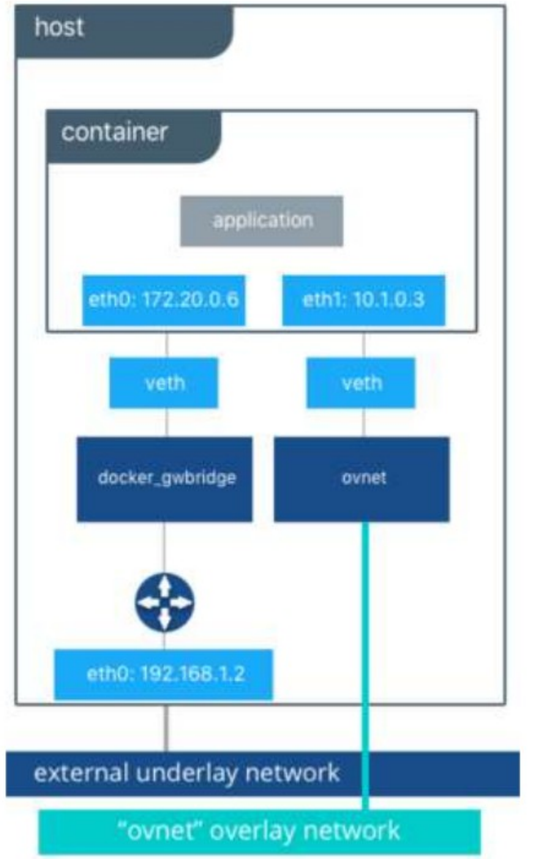
Overlay Driver

- Used for container connectivity across hosts.
- Before Docker 1.12 version, Overlay driver needed external KV store. After 1.12, external KV store is not needed.
- Containers connected to overlay network also get connected to “docker_bwbridge” for external access.
- Vxlan is used for encapsulation.

docker network create --driver overlay onet

docker run -ti --name client --network onet smakam/myubuntu:v4 bash

docker run -d --name web --network onet nginx



Picture from Docker white paper

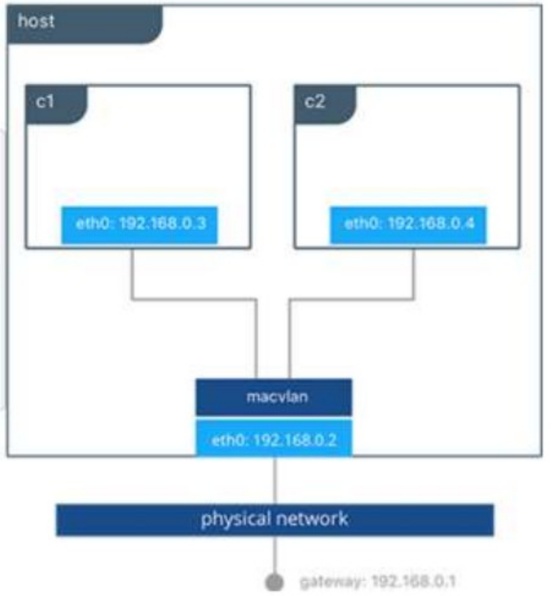
Macvlan driver

- Macvlan allows a single physical interface to have multiple mac and ip addresses using macvlan sub-Interfaces
- Macvlan driver allows for containers to directly connect to underlay network. Works well for connectivity to legacy applications.
- Provides connectivity within a single host as well as across hosts.

docker network create -d macvlan --subnet=192.168.0.0/16 --ip-range=192.168.2.0/24 -o macvlan_mode=bridge -o parent=eth1 macvlan1

docker run -d --name web1 --network macvlan1 nginx

docker run -d --name web2 --network macvlan1 nginx



Picture from Docker white paper

Docker Network plugins

- Extends functionality of Docker networking by using plugins to implement networking control and data plane.
- Docker provides batteries included approach where user has a choice of using Docker network drivers or plugins provided by other vendors.
- Using plugins, switch vendors can get Docker integrated with their custom switches having special features or with custom features like policy based networking.
- Docker network plugins follow CNM(libnetwork) model.
- Docker 1.13.1+ included support for global scoped network plugins that allows network plugins to work in Swarm mode.
- Following network plugins are available now:
  - Contiv – Network plugin from Cisco. Supports L2 and L3 physical topology. Integrates with Cisco ACI. Also provides policy based networking.
  - Calico – Follows Layer3 rather than overlay approach. Uses policy based networking.
  - Weave – Follows Overlay approach
  - Kuryr – Uses openstack Neutron to provide container networking

IP Address management

- Docker does the IP address management by providing subnets for networks and IP addresses for containers.
- For default “bridge” network, custom subnet can be specified in Docker daemon options.
- Users can specify their own subnet while creating networks and specify IP when creating containers. Following example illustrates this.

  docker network create --subnet=172.19.0.0/16 mynet

  docker run --ip 172.19.0.22 -it --network mynet smakam/myubuntu:v4 bash
- Using remote IPAM plugin, IP addresses can be managed by external application instead by Docker.
- Remote IPAM plugin can be specified using “--ipam-driver” option while creating Docker network. Infoblox is an example of external Docker IPAM plugin.
- Docker also supports assignment of IPV6 addresses for Containers.

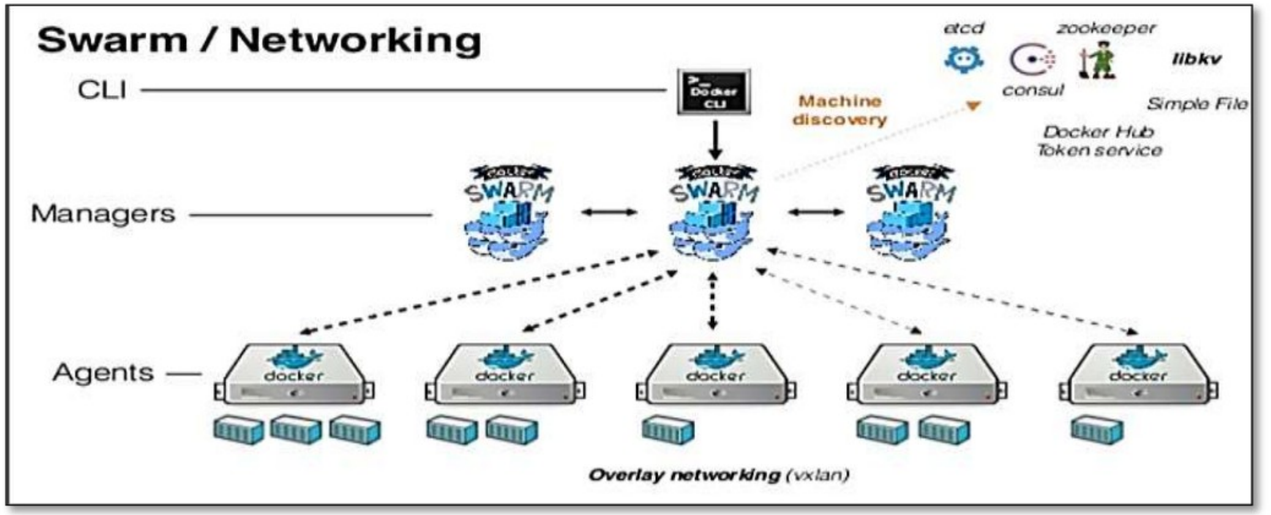
Default Networks created by Docker

```
jvherrera@juanvi-HP-ZBook-14u-G5:~$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
d62821368957        bridge              bridge              local
c6e08e54428b        docker_gwbridge     bridge              local
d80564d14448        efk_default         bridge              local
303a1d9ef12e        host                host                local
ouuex7p627ti        ingress             overlay             swarm
372b87a7cff2        none                null                local
```

- “bridge” is the default Bridge network
- “docker_gwbridge” is used by multi-host networks to connect to outside world
- “host” network is used for having containers in host namespace
- “ingress” network is used for routing mesh
- “none” network is used when Containers don’t need any networking
- “Scope” signifies if the network is local to host or across the Swarm cluster

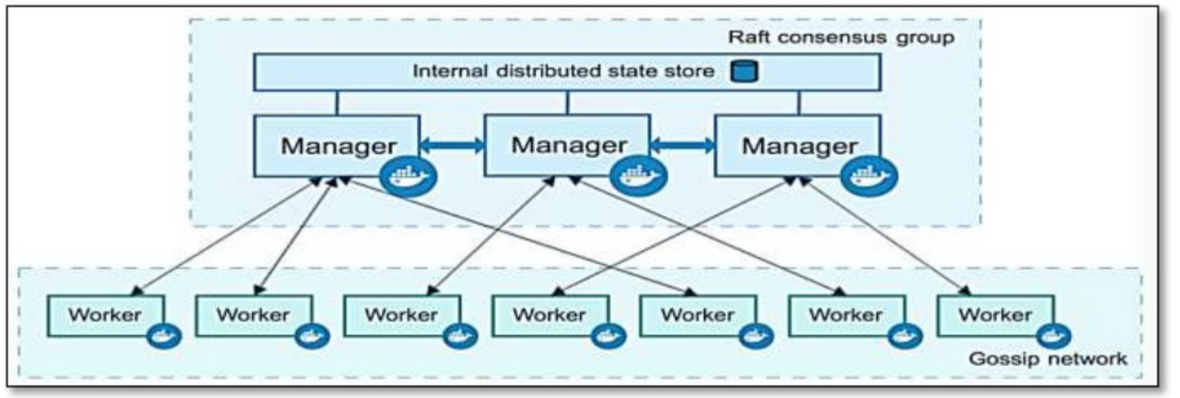
Legacy Swarm mode

- In this mode, Swarm is not integrated with Docker engine and it runs as a separate container.
- Needs separate KV store like Consul, etcd.
- Supported in Docker prior to version 1.12
- This is a legacy mode that is deprecated currently



Swarm Mode

- Orchestration supported by Docker from 1.12 version.
- Using Raft protocol, Managers maintain state of Swarm nodes as well as services running on them.
- Gossip protocol is used by workers to establish control plane between them. Only workers in same network exchange state associated with that network.
- Control plane is encrypted by default. Data plane can be optionally encrypted using “--opt encrypted” when creating network.
- No separate KV store is needed with Swarm mode.
- Prior to Docker 17.06, Swarm mode was supported only with Overlay driver. Post 17.06, all network drivers are supported with Swarm mode.

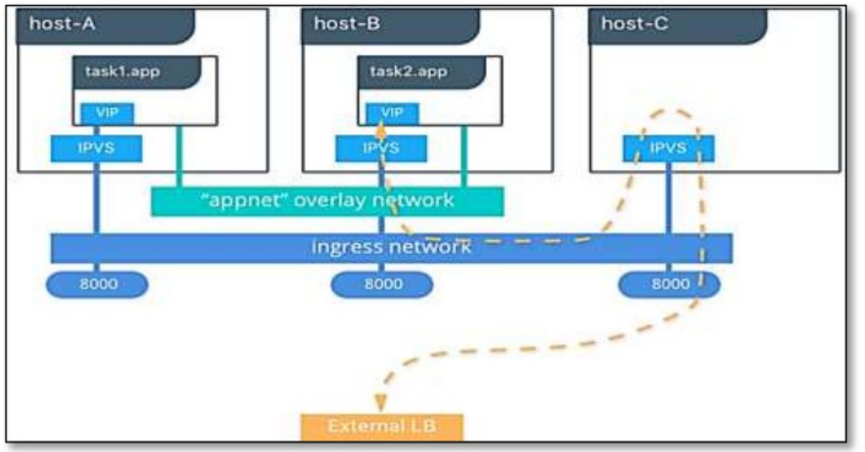


Service Discovery

- Service discovery is provided by DNS server available in Docker engine.
- For unmanaged containers, container name resolves to container IP. Alias names can be also be used.
- For services using service IP(endpoint mode=vip), service name resolves to service IP which in turn forwards the request to containers. In this case, ipvs based L4 load balancing is done.
- For services using direct DNS(endpoint mode=dnsrr), service name directly resolves to container IP. In this case, DNS round robin load balancing is done.
- Service Discovery is network scoped. Only containers in same network can discover each other.

Load balancing

- For unmanaged containers, load balancing is done using simple round robin load balancing. Using aliases, a single alias can load balance to multiple unmanaged containers .
- Docker takes care of load balancing internal services to the containers associated with the services.
- For services using service IP(endpoint mode=vip), ipvs and iptables are used to load balance. This provides L4 based load balancing. Ipvs is Linux kernel load balancing feature.
- For services using direct DNS(endpoint mode=dnsrr), DNS round robin balancing is used.
- For services exposed externally, Docker uses routing mesh to expose the service on all Swarm nodes. Routing mesh uses “ingress” network to connect all nodes.
- For HTTP based load balancing, HRM(HTTP Routing mesh) can be used. This is supported only with Docker EE.



Picture from Docker white paper

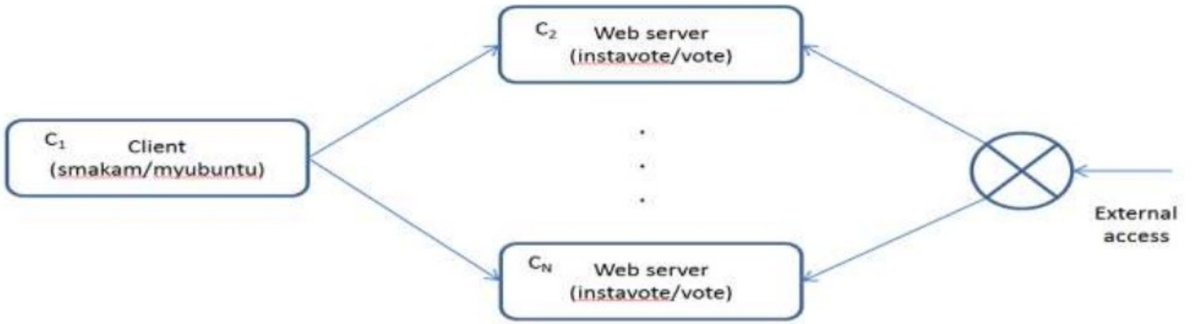
Swarm Networking - Sample application detail

- The application will be deployed in 2 node Swarm cluster.
- "client" service has 1 client container task. "vote" service has multiple vote container tasks. Client service is used to access multi-container voting service. This application is deployed in a multi-node Swarm cluster.
- "vote" services can be accessed from "client" service as well as from outside the swarm cluster.

docker network create -d overlay overlay1

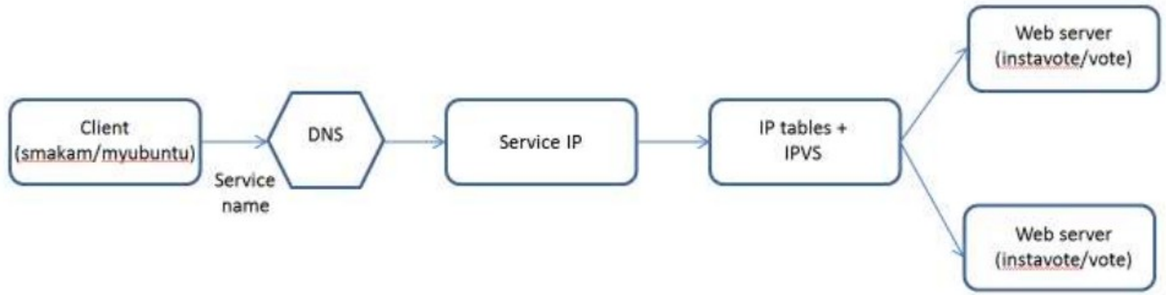
docker service create --replicas 1 --name client --network overlay1 smakam/myubuntu:v4 sleep infinity

docker service create --name vote --network overlay1 --mode replicated --replicas 2 --publish mode=ingress,target=80,published=8080 instavote/vote

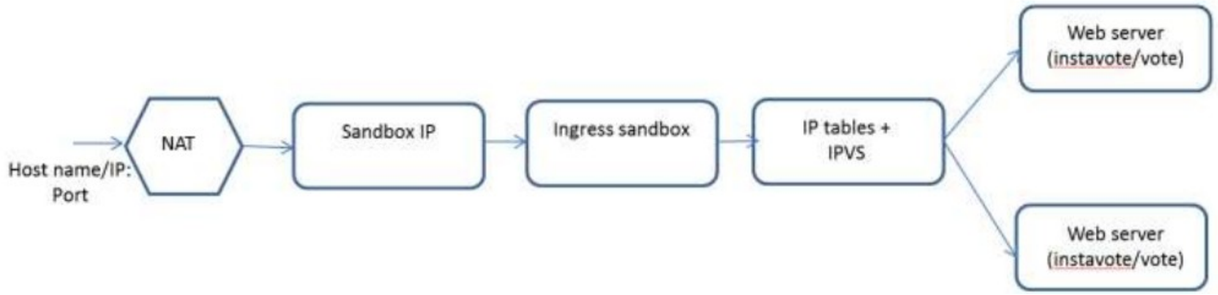


Swarm Networking - Application access flow

“Client” service accessing “vote” service using “overlay” network

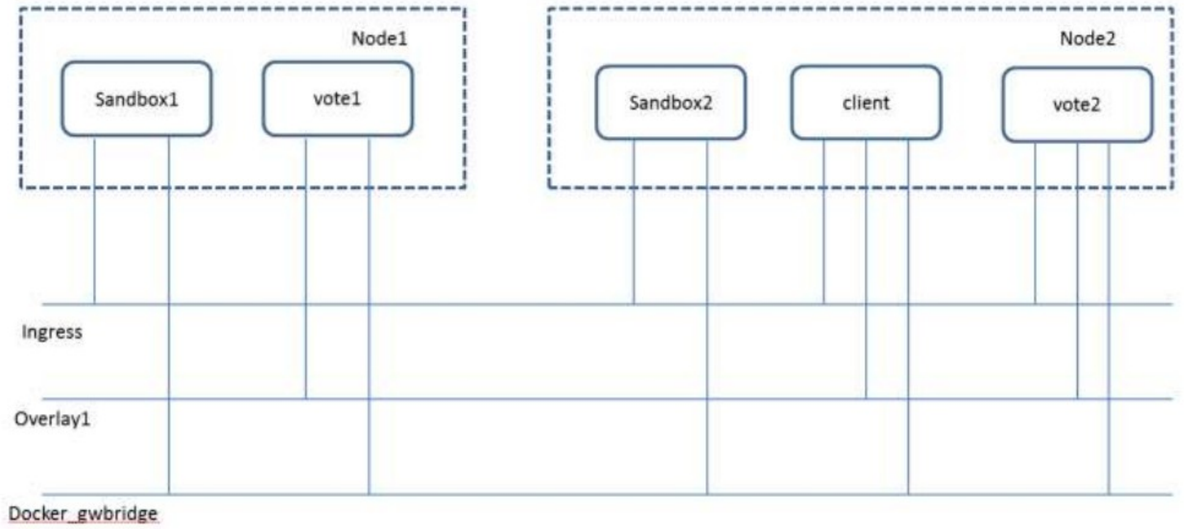


Accessing “vote” service using “ingress” network externally



Swarm Application - Networking detail

- Sandboxes and “vote” containers are part of “ingress” network and it helps in routing mesh.
- “client” and “vote” containers are part of “overlay1” network and it helps in service connectivity.
- All containers are part of the default “docker_gwbridge” network. This helps for external access when services gets exposed using publish mode “host”



Docker Network debug commands

- Basic Swarm debugging:

Docker node ls

- Service and Container debugging:

Docker service logs <service name/id>

Docker service inspect <service name/id>

Docker container logs <container name/id>

Docker container inspect <container name/id>

- Network debugging:

Docker network inspect <network name/id>

Use "-v" option for verbose output

Troubleshooting using debug container

- All Linux networking tools are packaged inside “nicolaka/netshoot”(https://github.com/nicolaka/netshoot) container. This can be used for debugging.
- Using this debug container avoids installation of any debug tools inside the container or host.
- Linux networking tools like tcpdump, netstat can be accessed from container namespace or host namespace.

Capture port 80 packets in the Container:

docker run -ti --net container:<containerid> nicolaka/netshoot

tcpdump –i eth0 –n port 80

Capture vxlan packets in the host:

docker run -ti --net host nicolaka/netshoot

tcpdump –i eth1 –n port 4789

- Debug container can also be used to get inside container namespace, network namespace and do debugging. Inside the namespace, we can run commands like “ifconfig”, “ip route”, “brctl show” to debug further.

Starting nsenter using debug container:

docker run -it --rm -v /var/run/docker/netns:/var/run/docker/netns --privileged=true nicolaka/netshoot

Getting inside container or network namespace:

nsenter –net /var/run/docker/netns/<networkid> sh

Compare Docker and Kubernetes Networking

Feature	Docker	Kubernetes
Abstraction	Container	Pod
Standard	CNM	CNI
Service discovery	Embedded DNS	Kube-dns
Internal load balancing	Iptables and ipvs	Iptables and Kube-proxy
External load balancing	Routing mesh	Nodeport
External plugins	Weave, Calico, Contiv	Flannel, Weave, Calico, Contiv

In this exercise, you will

- Differentiate
  - OS virtualization and Docker containers
  - Docker images and containers
- Explain
  - Dockerfiles
  - Docker Compose
  - Docker Swarm

OS Virtualization and Containers

- OS virtualization
  - The physical hypervisor hardware is virtualized
  - Starting the virtual machines means waiting for OS boot
- Containers
  - The OS is virtualized and not hardware
  - Start much more quickly than virtual machines
    - The OS kernel is already running
    - The OS is shared among multiple containers

Docker Images Vs Container

- Containers are based on images
  - Images contain software and settings for running a container
  - Images contain metadata describing the image
  - A container is a runtime instance of an image
- Container contents
  - All elements need to run an application
    - Software
    - Settings
    - App-specific libraries
    - Runtime environment
    - Tools

Dockerfiles

- Dockerfile
  - Used to create a Docker image
  - It is a text file
    - Normally named Dockerfile
    - Normally exists in the root of the context
      - Otherwise, specify path when using "docker build –f <path to Dockerfile>"
    - Contains directives for images

Docker Compose

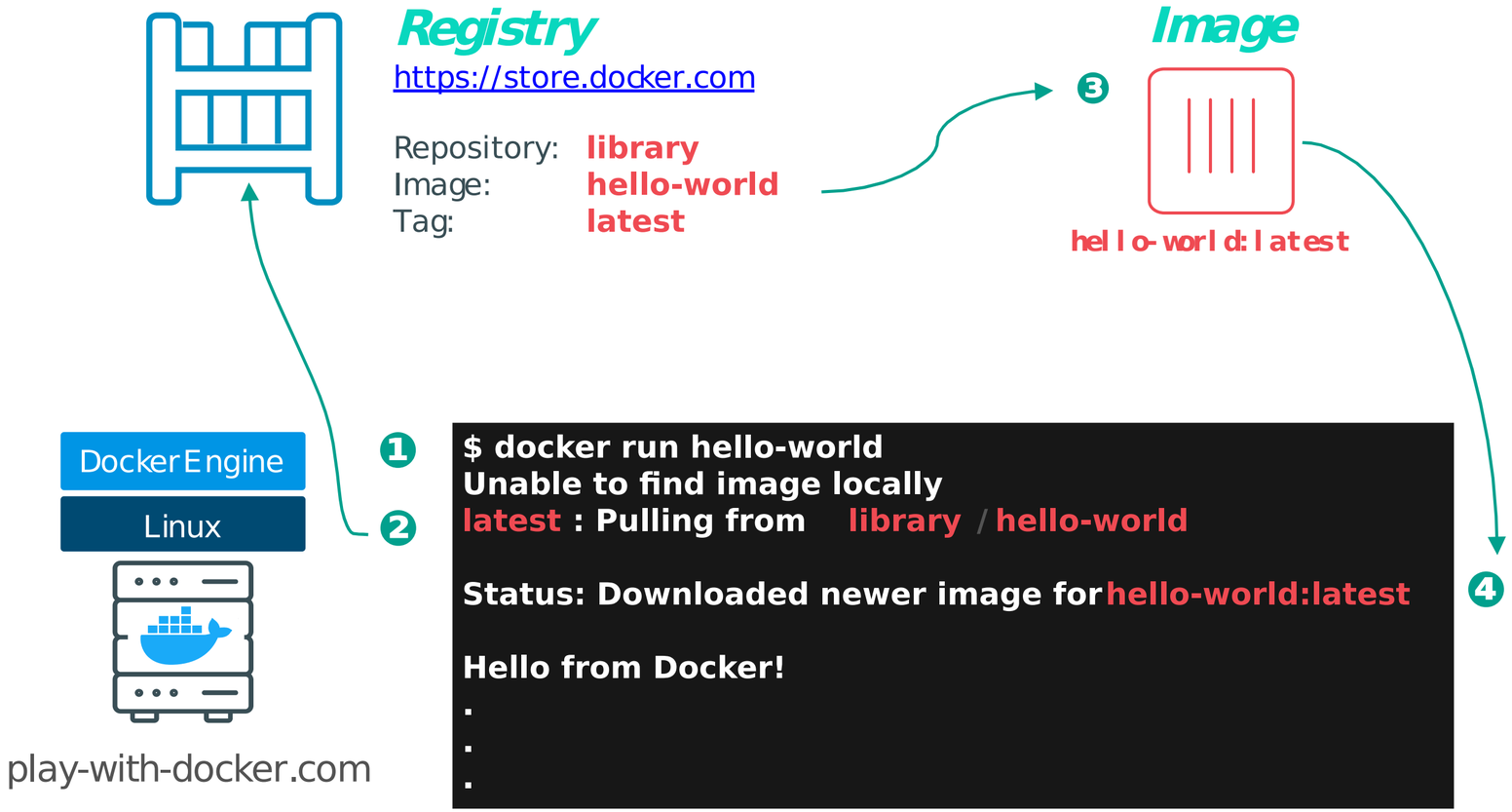
- Configure and run multi-container applications
- Multi-container application potential components
  - Web servers
  - Database servers
  - Message queues
- Service configuration
  - YAML file
- Runs on MacOS, Linux, Windows

Docker Swarm

- Swarm
  - Docker host clustering
  - Container orchestration
  - Container workload scheduling
- Docker hosts are configured to run in swarm mode

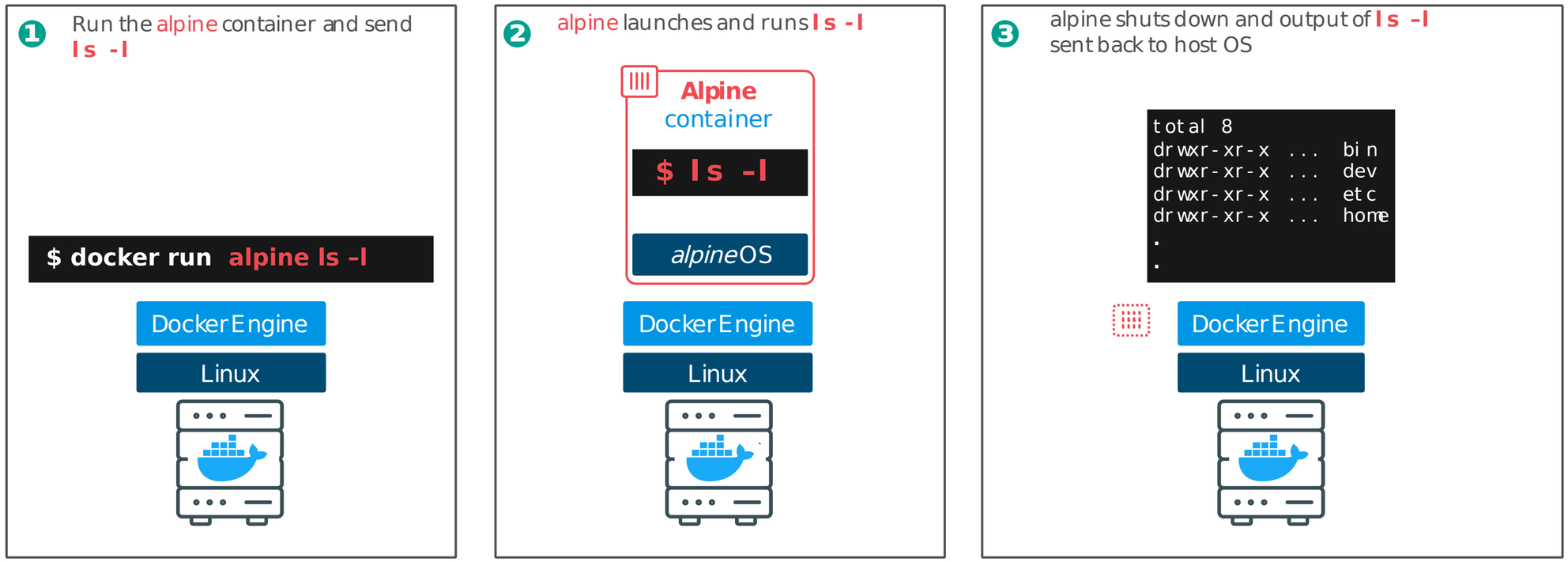
Hands on

Hello World: What Happened?



Hands on

docker run Details

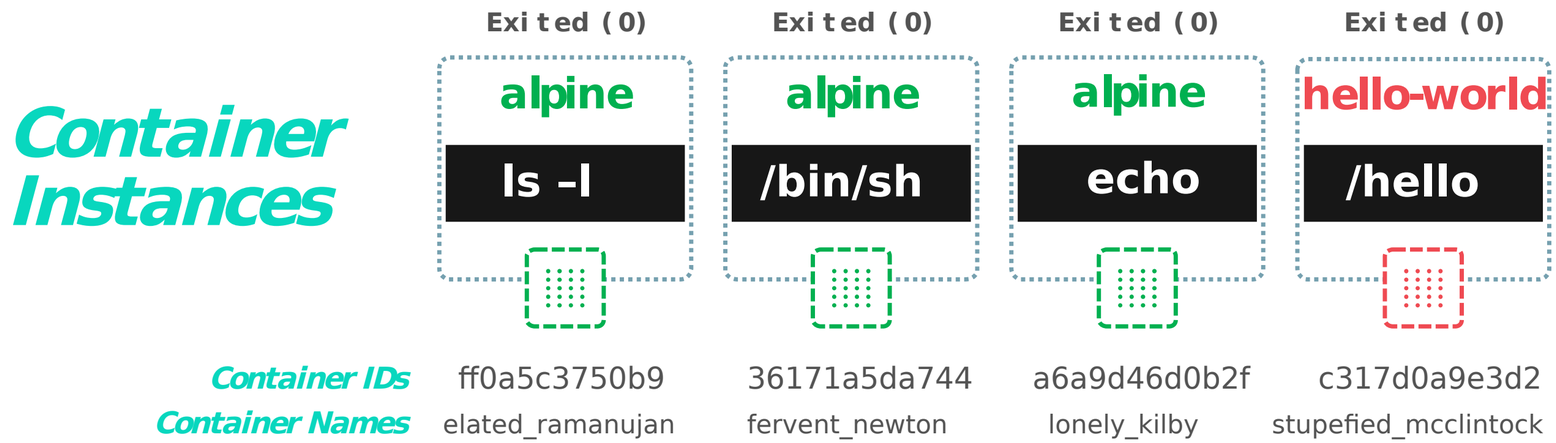


Hands on

```
docker container run alpine echo "hello from alpine"
docker container run alpine /bin/sh
docker container run -it alpine /bin/sh
```

Docker Container Instances

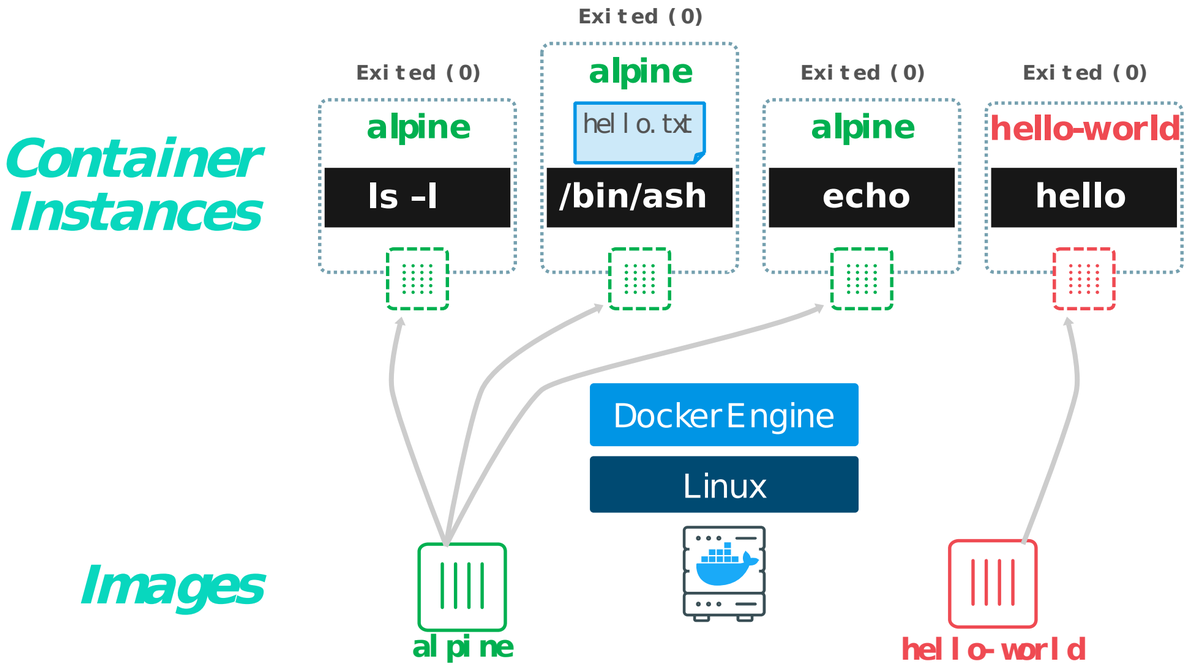
Output of docker container ls -a



Hands on

```
docker container run -it alpine /bin/ash
echo "hello world" > hello.txt
docker container run alpine ls
docker container ls -a
```

Docker Container Isolation

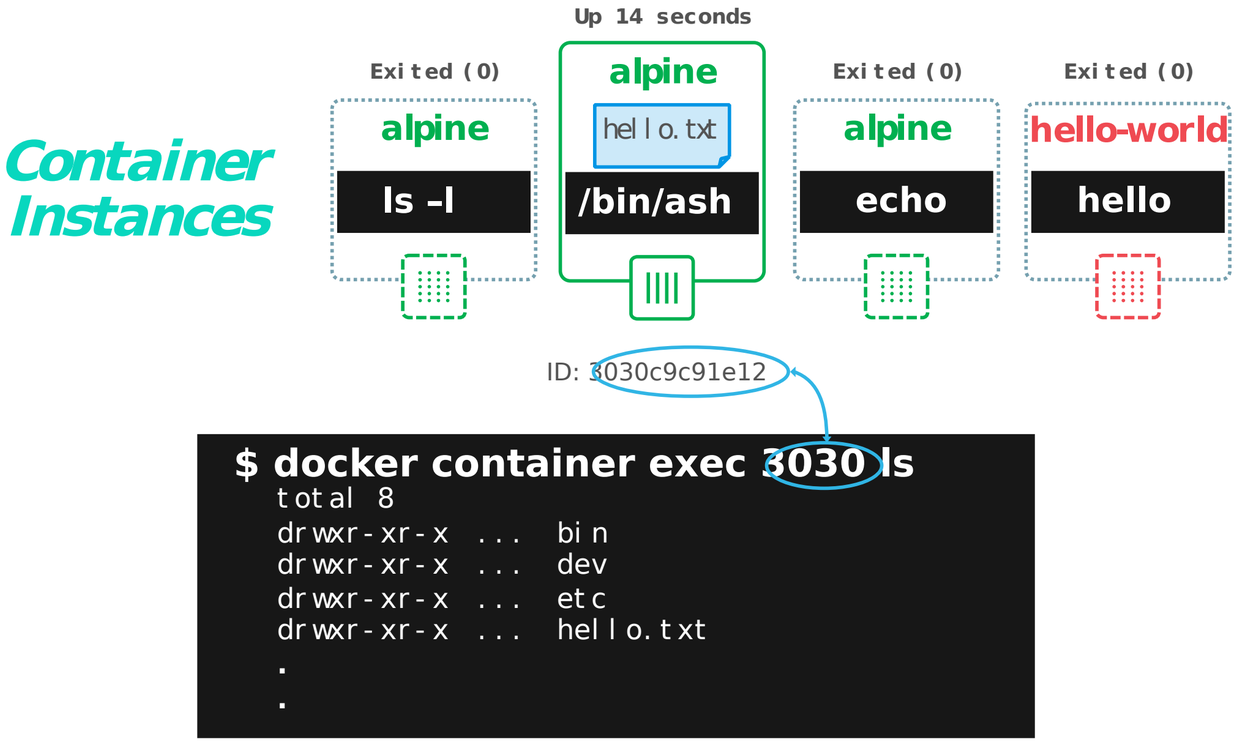


Hands on

Now use the docker container ls command again to list the running containers.
docker container start <container ID>
docker container exec <container ID> ls

---

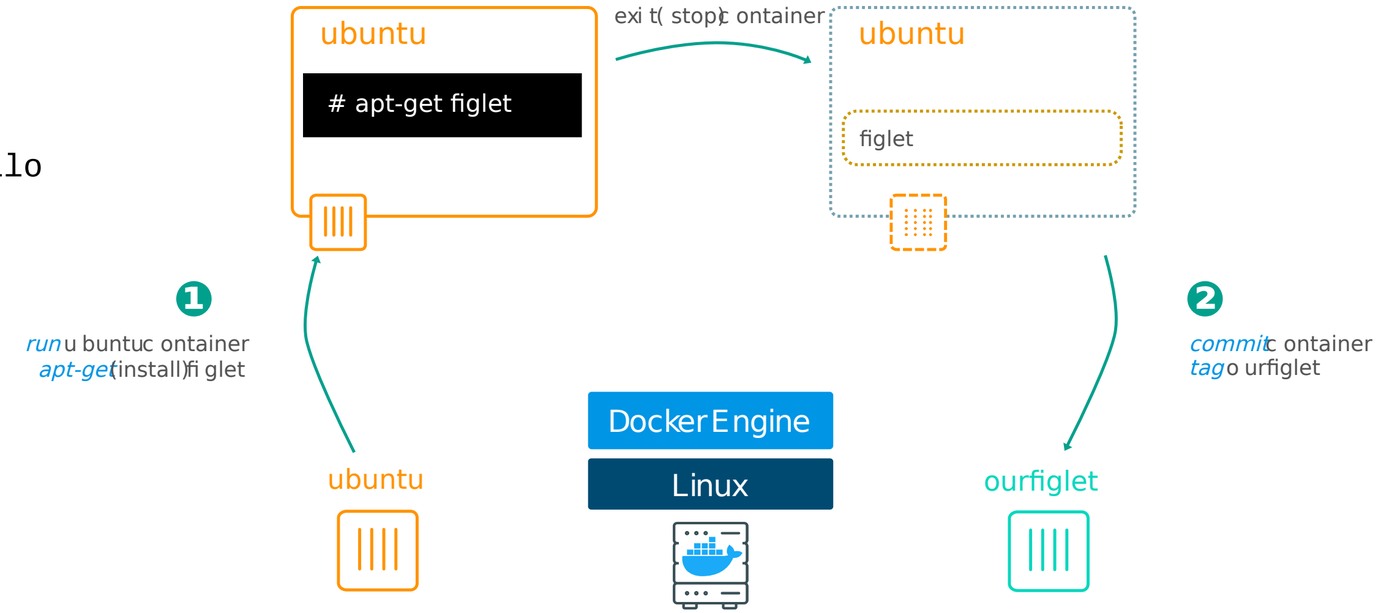
docker container exec



Hands on

```
docker container run -ti ubuntu bash
apt-get update
apt-get install -y figlet
figlet "hello docker"
exit
docker container ls -a
docker container commit CONTAINER_ID
docker image ls
docker image tag <IMAGE_ID> ourfiglet
docker image ls
docker container run ourfiglet figlet hello
```

Image Creation: Instance Promotion



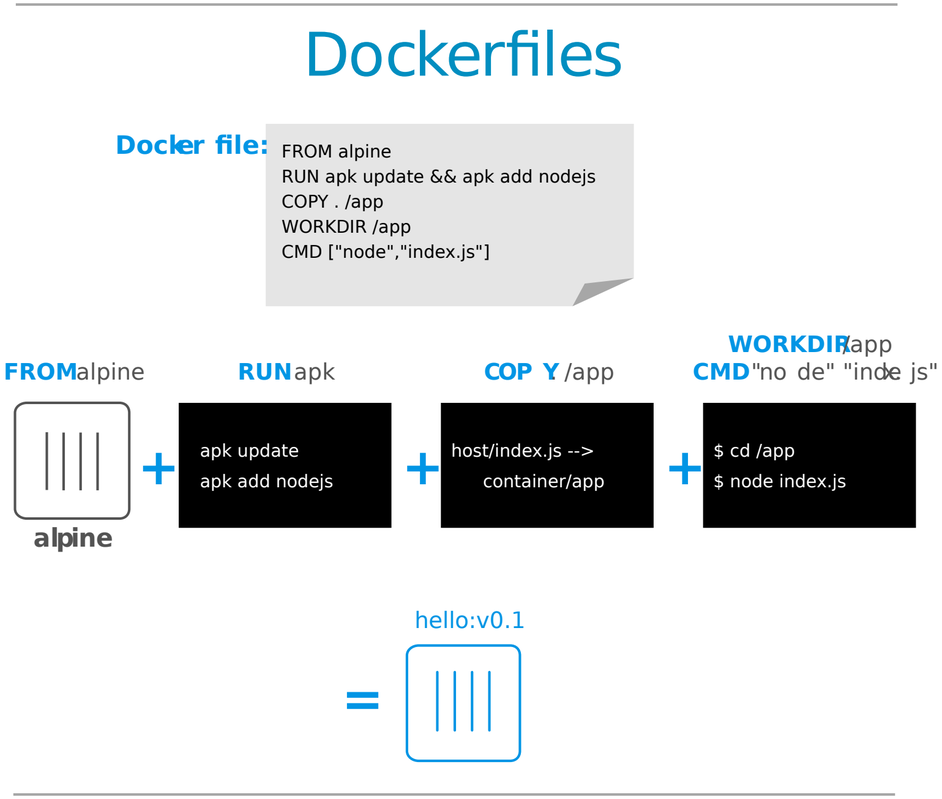
Hands on

Create a new file “index.js” with the following content:

```
var os = require("os");
var hostname = os.hostname();
console.log("hello from " + hostname)
```

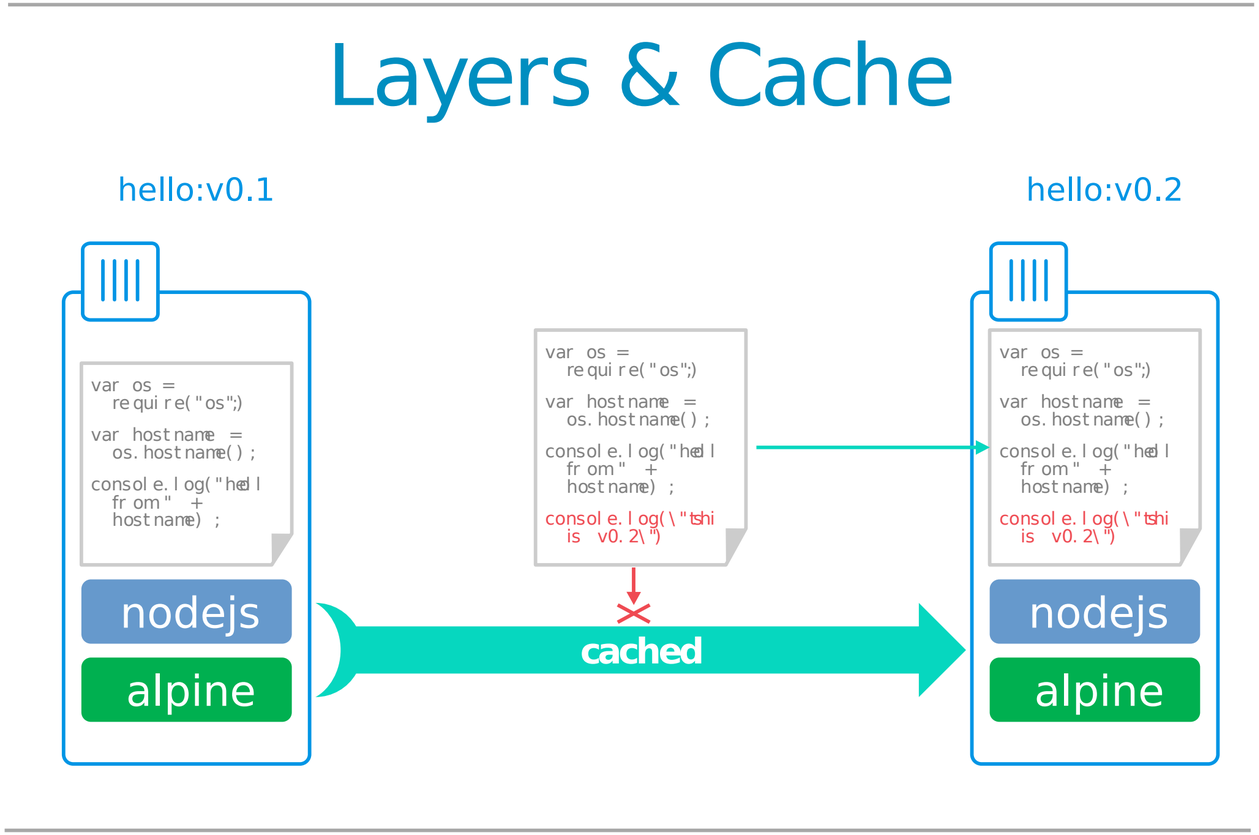
Create a new file called “Dockerfile” with the following content:

```
FROM alpine
RUN apk update && apk add nodejs
COPY . /app
WORKDIR /app
CMD ["node","index.js"]

docker image build -t hello:v0.1 .
docker container run hello:v0.1


echo "console.log(\"this is v0.2\");" >> index.js
docker image build -t hello:v0.2 .
```



Hands on

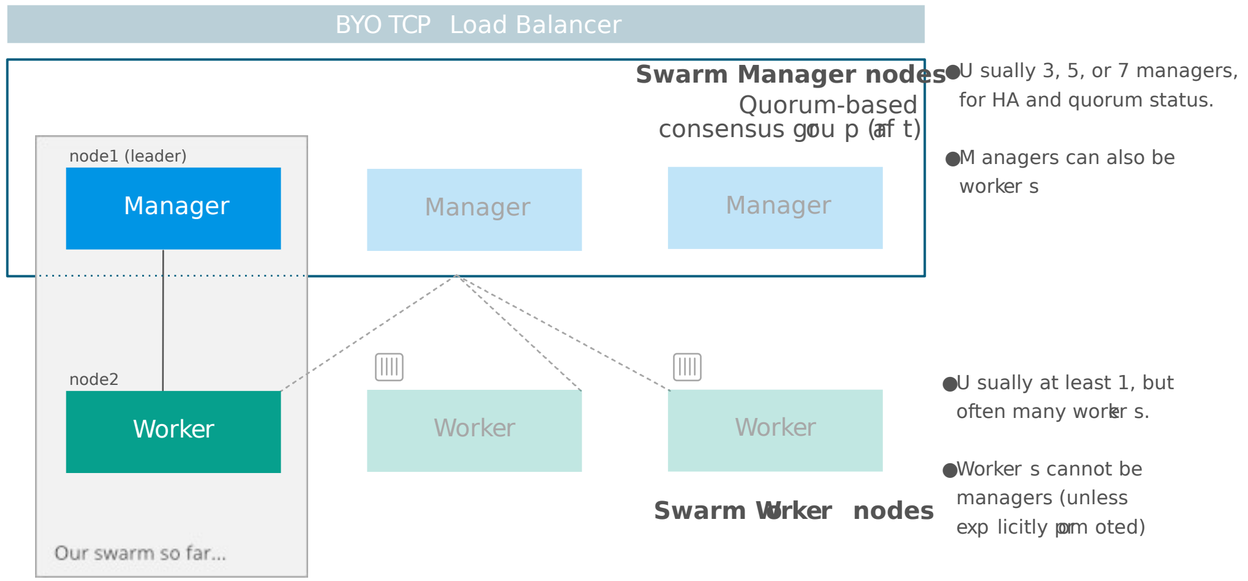
```
echo "console.log(\"this is v0.2\");" >> index.js
docker image build -t hello:v0.2 .
```



Hands on

```
docker swarm init --advertise-addr $(hostname -i)
docker node ls
git clone https://github.com/docker/example-voting-app
cd example-voting-app
cat docker-stack.yml
docker stack deploy --compose-file=docker-stack.yml voting_stack
docker stack ls
docker stack services voting_stack
docker service ps voting_stack_vote
```

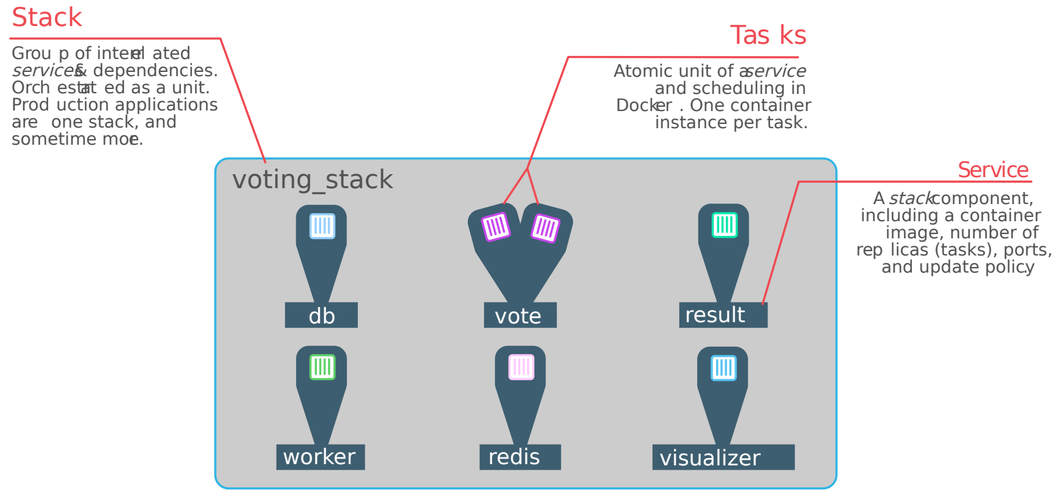
Docker Enterprise Edition: Swarm Architecture



Hands on

```
docker swarm init --advertise-addr $(hostname -i)
docker node ls
git clone https://github.com/docker/example-voting-app
cd example-voting-app
cat docker-stack.yml
docker stack deploy --compose-file=docker-stack.yml voting_stack
docker stack ls
docker stack services voting_stack
docker service ps voting_stack_vote
```

Swarm: Stacks, Services & Tasks

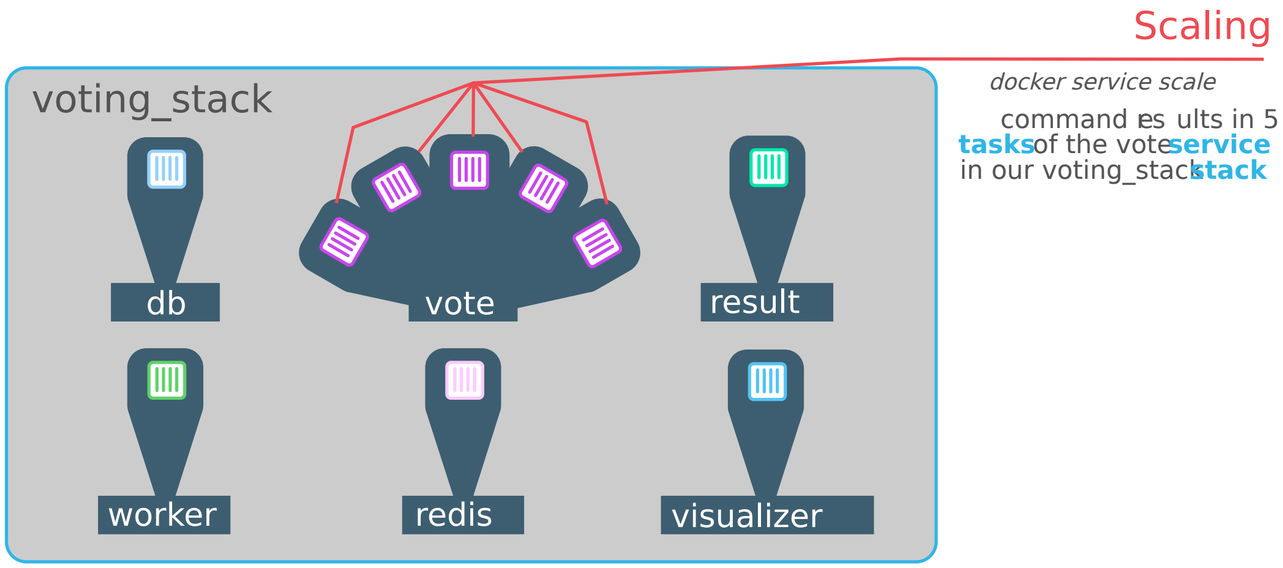


Voting App Architecture

Hands on

```
docker service scale voting_stack_vote=5
docker stack services voting_stack
```

Scaling Services



Voting App Architecture

Hands on for review

Source code: https://github.com/juanviz/devacademyDocker.git

0. Download a container image from Docker Hub
1. List image containers downloaded
2. Create a container from just downloaded container image of Ubuntu and run an interactive shell
3. Update repositories package list
4. Finish shell session and check the status of the container
5. Sign in in Docker Hub
6. Commit the change to the just created and updated Ubuntu container
7. Push the change to Your Docker Hub account
8. Create a Docker Swarm cluster with your local node
9. List nodes of Docker Swarm Cluster
10. Create a service for expose nginx webserver
11. List Docker Swarm services created
12. Scale up nginx services to 4 replicas
13. List details about services running
14. Clear all the files created previously
15. Download course repository for run flask-app container
16. Edit Dockerfile for ensure all of the configuration is fine
17. Create the image with flask-app
18. Upload the image just created to Docker Hub

Hands on for review

Source code: https://github.com/juanviz/devacademyDocker.git

19. Check networking od your Docker node
20. Start two alpine containers running ash, which is Alpine’s default shell rather than bash.
21. Inspect the bridge network to see what containers are connected to it
22. The containers are running in the background. Use the docker attach command to connect to alpine1.
23. This succeeds. Next, try pinging the alpine2 container by container name. This will fail.
24. Detach from alpine1 without stopping it by using the detach sequence
25. In this example, we again start two alpine containers, but attach them to a user-defined
26. Inspect the alpine-net network.
27. Create your four containers.
28. Inspect the bridge network and the alpine-net network again
29. Let’s connect to alpine1 and test this out.
30. From alpine1, you should not be able to connect to alpine3 at all, since it is not on the alpine-net network.
31. Attach to it and run the tests.
32. As a final test, make sure your containers can all connect to the internet by pinging google.com.
33. Stop and remove all containers and the alpine-net network.
34. Create and start the container as a detached process. The --rm option means to remove the container once it exits/stops. The -d flag means to start the container detached (in the background).

Hands on for review

Source code: https://github.com/juanviz/devacademyDocker.git
35. Examine all network interfaces and verify that a new one was not created.
36. Stop the container. It will be removed automatically as it was started using the --rm option.
37. Open a terminal window. List current networks before you do anything else. Here's what you should see if you've never added a network or initialized a swarm on this Docker daemon. You may see different networks, but you should at least see these (the network IDs will be different

38. Start two alpine containers running ash, which is Alpine's default shell rather than bash.
39. Inspect the bridge network to see what containers are connected to it.
40. The containers are running in the background. Use the docker attach command to connect to alpine1.
41. Stop and remove both containers.
42. Create a macvlan network called my-macvlan-net. Modify the subnet, gateway, and parent values to values that make sense in your environment.
43. Start an alpine container and attach it to the my-macvlan-net network. The -dit flags start the container in the background but allow you to attach to it. The --rm flag means the container is removed when it is stopped.
44. Inspect the my-macvlan-alpine container and notice the MacAddress key within the Networks key
45. 45. Check out how the container sees its own network interfaces by running a couple of docker exec commands.

Hands on for review

Source code: https://github.com/juanviz/devacademyDocker.git

46. Stop the container (Docker removes it because of the --rm flag), and remove the network.
47. Create a macvlan network called my-8021q-macvlan-net. Modify the subnet, gateway, and parent values to values that make sense in your environment.
48. Start an alpine container and attach it to the my-8021q-macvlan-net network. The -dit flags start the container in the background but allow you to attach to it. The --rm flag means the container is removed when it is stopped.
49. Inspect the my-second-macvlan-alpine container and notice the MacAddress key within the Networks key
50. Check out how the container sees its own network interfaces by running a couple of docker exec commands.
51. Stop the container (Docker removes it because of the --rm flag), and remove the network.

References

https://github.com/juanviz/devacademyDocker

https://www.docker.com/

https://hub.docker.com/

https://play-with-docker.com