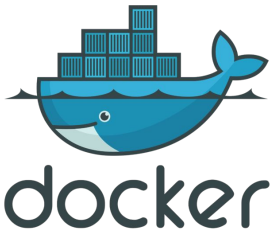
Docker Certified Associate

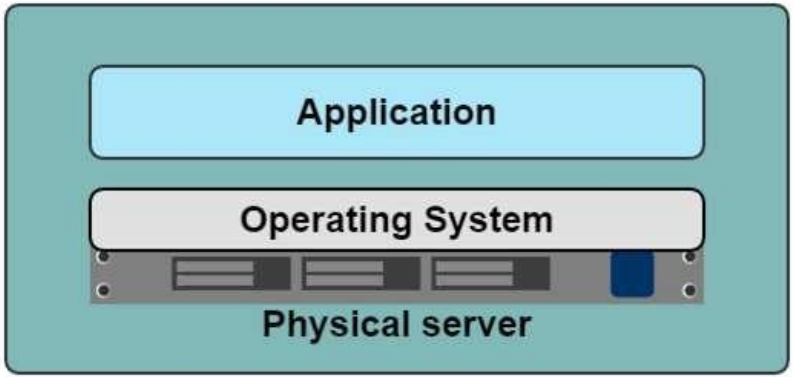
ITLearn 360

Introduction to Docker



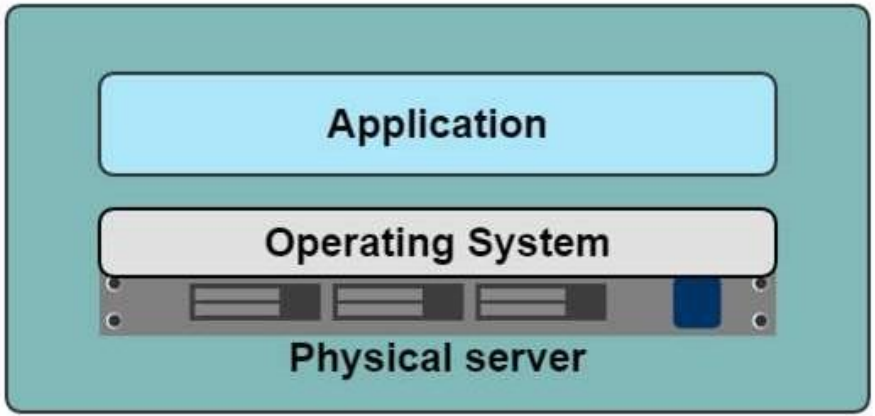
A History Lesson

One application on one physical server



Historical limitations of application deployment

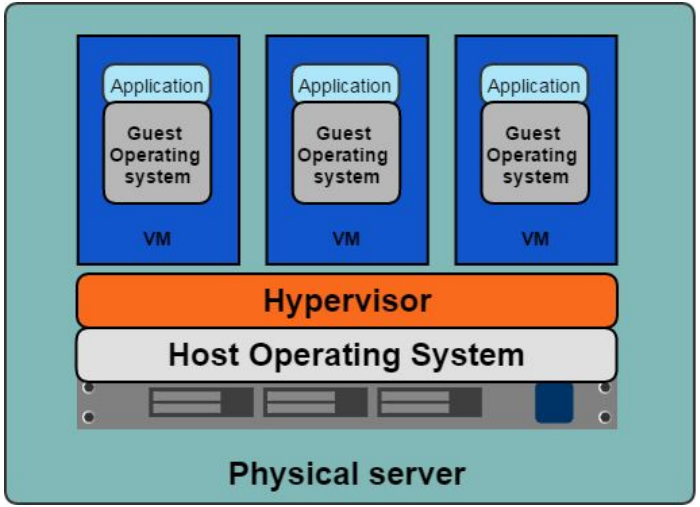
- Slow deployment times
- Huge costs
- Wasted resources
- Difficult to scale
- Difficult to migrate
- Vendor lock in



A History Lesson

Hypervisor-based Virtualization

- One physical server can contain multiple applications
- Each application runs in a virtual machine (VM)



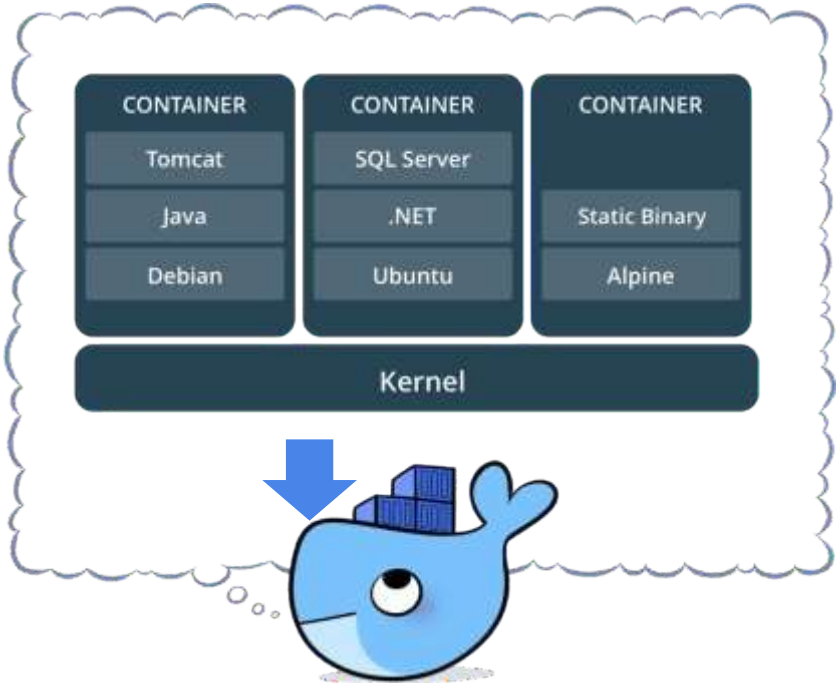
Benefits of VMs

- Better resource pooling
  - One physical machine divided into multiple virtual machines
- Easier to scale
- VMs in the cloud
  - Rapid elasticity
  - Pay as you go model

Limitations of VMs

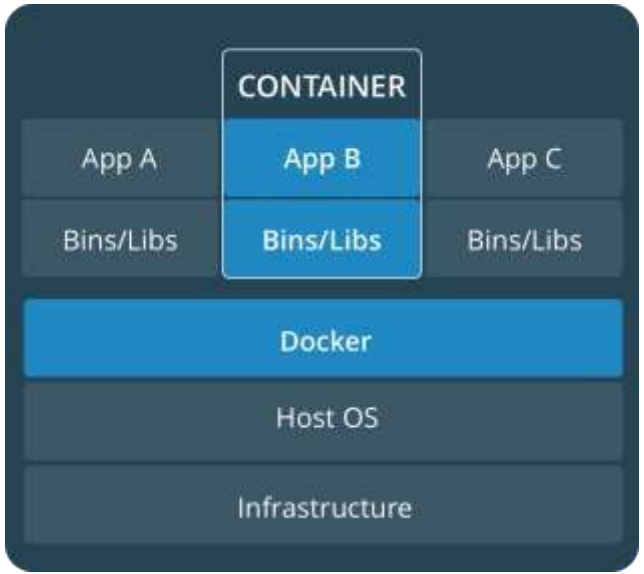
- Each VM stills requires
  - CPU allocation
  - Storage
  - RAM
  - An entire guest operating system
- The more VMs you run, the more resources you need
- Guest OS means wasted resources
- Application portability not guaranteed

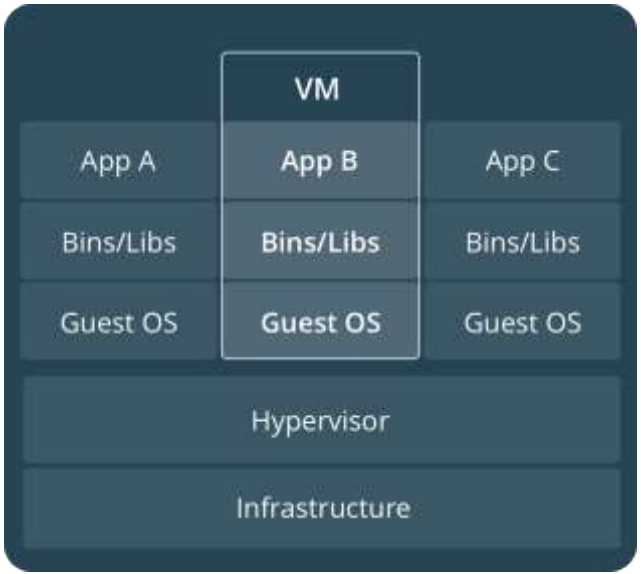
What is a container?



- Standardized packaging for software and dependencies
- Isolate apps from each other
- Share the same OS kernel
- Works with all major Linux and Windows Server

Comparing Containers and VMs

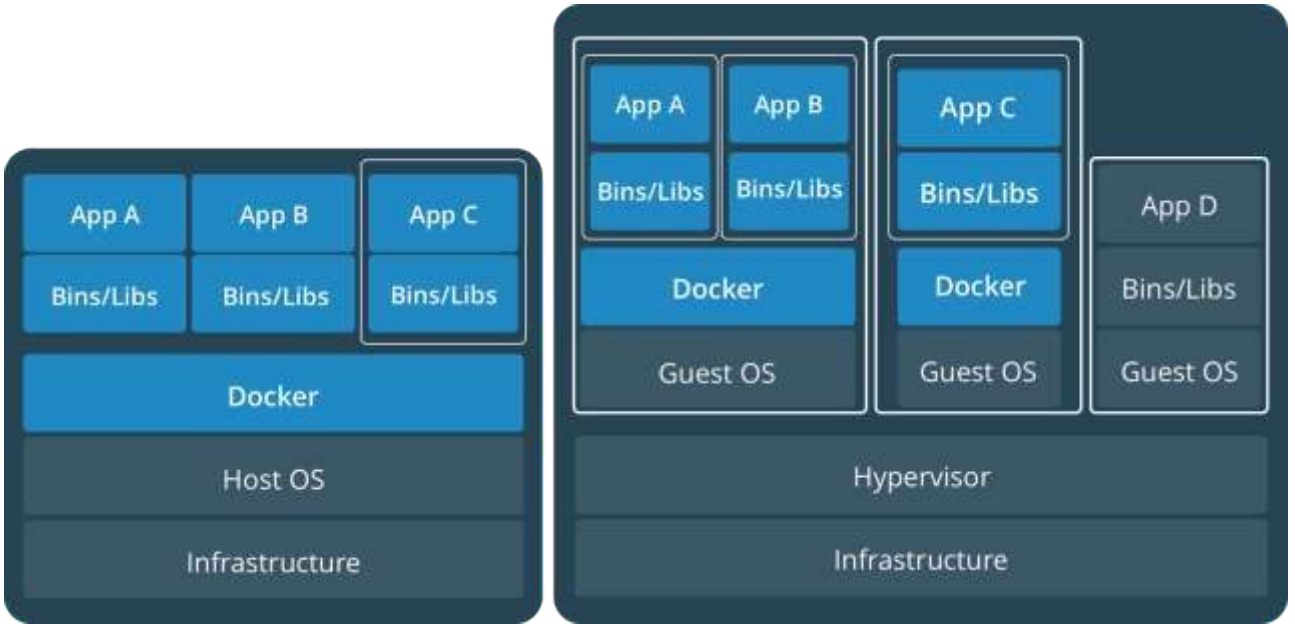




Containers are an app level construct

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

Containers and VMs together



Containers and VMs together provide a tremendous amount of flexibility for IT to optimally deploy and manage apps.

Key Benefits of Docker Containers

Speed

- No OS to boot = applications online in seconds

Portability

- Less dependencies between process layers = ability to move between infrastructure

Efficiency

- Less OS overhead
- Improved VM density

Docker Basics

- **Container**
  The image when it is ‘running.’ The standard unit for app service

- **Image**
  The basis of a Docker container. The content at rest.

- **Engine**
  The software that executes commands for containers. Networking and volumes are part of Engine. Can be clustered together.

- **Registry**
  Stores, distributes and manages Docker images

- **Control Plane**
  Management plane for container and cluster orchestration

Docker Commands

- docker run – Runs a command in a new container.
- docker start – Starts one or more stopped containers
- docker stop – Stops one or more running containers
- docker build – Builds an image from a Dockerfile
- docker pull – Pulls an image or a repository from a registry
- docker push – Pushes an image or a repository to a registry
- docker export – Exports a container’s filesystem as a tar archive
- docker exec – Runs a command in a run-time container
- docker search – Searches the Docker Hub for images
- docker attach – Attaches to a running container
- docker commit – Creates a new image from a container’s changes

Docker Image

- A Docker image is a read-only template that contains a set of instructions for creating a container that can run on the Docker platform.

- It provides a convenient way to package up applications and preconfigured server environments, which you can use for your own private use or share publicly with other Docker users.

- A Docker image is made up of a collection of files that bundle together all the essentials, such as installations, application code and dependencies, required to configure a fully operational container environment.

Docker Registry

- Docker Registry is where the Docker Images are stored.

- The Registry can be either a user's local repository or a public repository like a Docker Hub allowing multiple users to collaborate in building an application.

- Even with multiple teams within the same organization can exchange or share containers by uploading them to the Docker Hub, which is a cloud repository similar to GitHub.

Dockerfile

- Each Docker container starts with a *Dockerfile*.

- A Dockerfile is a text file written in an easy-to-understand syntax that includes the instructions to build a Docker *image*.

- A Dockerfile specifies the operating system that will underlie the container, along with the languages, environmental variables, file locations, network ports, and other components it needs—and, of course, what the container will actually be doing once we run it.

Dockerfile Example :-

```
FROM nginx

RUN apt update -y && apt upgrade -y

COPY . /usr/share/nginx/html

EXPOSE 80
```

Dockerfile Commands

- **FROM**
- **WORKDIR**
- **ENV**
- **COPY**
- **LABEL**
- **RUN**
- **ADD**
- **.dockerignore**
- **ARG**
- **EXPOSE**
- **USER**
- **VOLUME**
- **CMD**
- **ENTRYPOINT**

Docker Container Restart Policy

Docker provides restart policies to control whether your containers start automatically when they exit, or when Docker restarts. Restart policies ensure that linked containers are started in the correct order. Docker recommends that you use restart policies, and avoid using process managers to start containers.

- no

Do not automatically restart the container. (the default)

- on-failure

Restart the container if it exit due to an error, which manifests as a non-zero exit code.

- always

Always restart the container if it stops. If it is manually stopped, it is restarted only when Docker daemon restarts or the container itself is manually restarted. (See the second bullet listed in restart policy details)

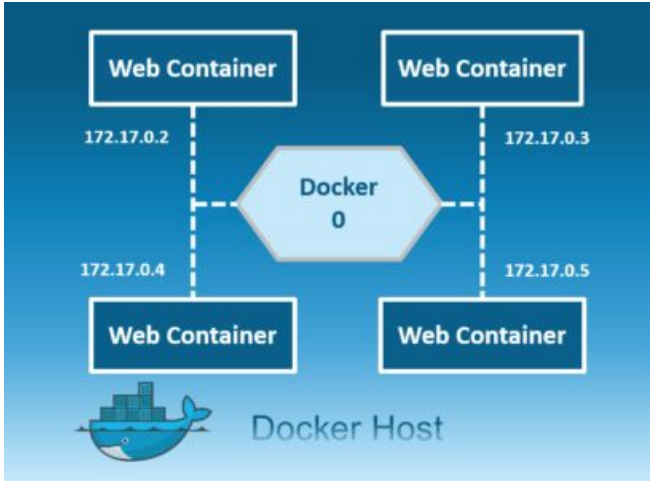
- unless-stopped

Similar to always, except that when the container is stopped (manually or otherwise), it is not restarted even after Docker daemon restarts.

Docker Network

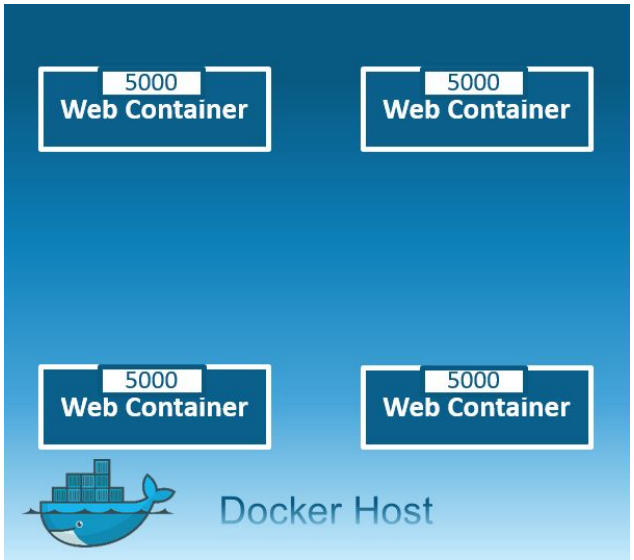
There are mainly 4 network drivers: Bridge, Host, None, Overlay.

Bridge: The bridge network is a private default internal network created by docker on the host. So, all containers get an internal IP address and these containers can access each other, using this internal IP. The Bridge networks are usually used when your applications run in standalone containers that need to communicate.



Docker Host Network

Host: This driver removes the network isolation between the docker host and the docker containers to use the host's networking directly. So with this, you will not be able to run multiple web containers on the same host, on the same port as the port is now common to all containers in the host network.

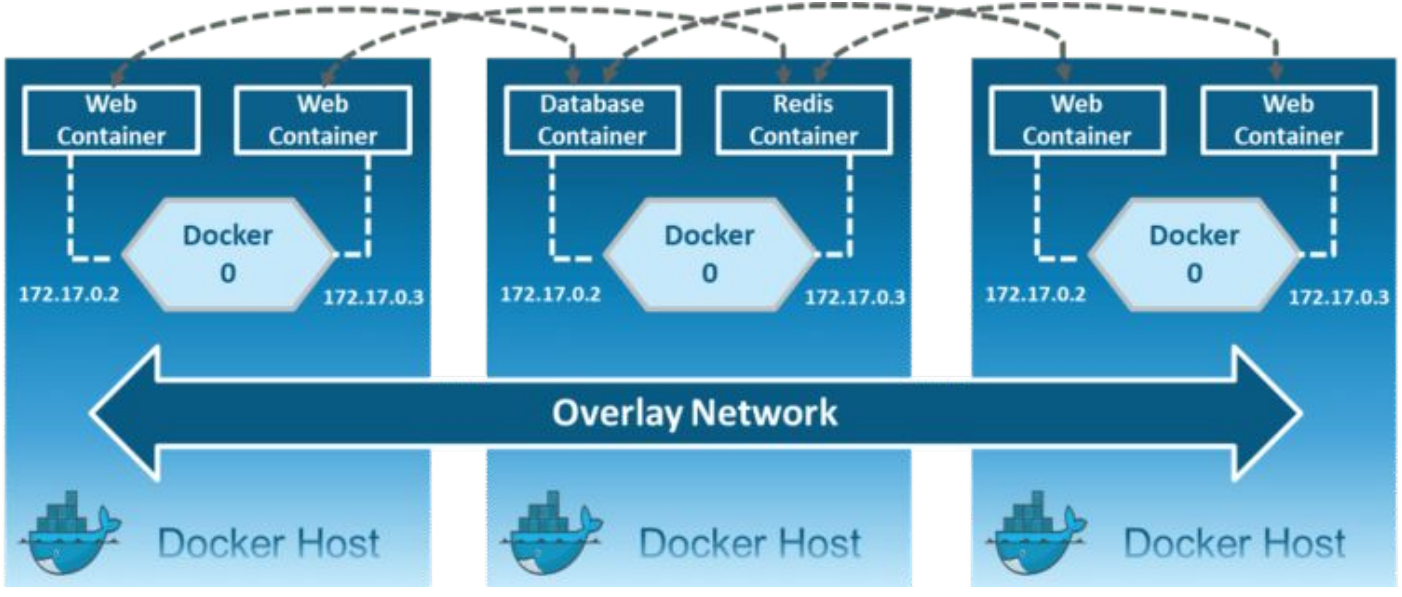


Docker ‘None’ Network

None: In this kind of network, containers are not attached to any network and do not have any access to the external network or other containers. So, this network is used when you want to completely disable the networking stack on a container and, only create a loopback device.

Docker Overlay Network

Overlay: Creates an internal private network that spans across all the nodes participating in the swarm cluster. So, Overlay networks facilitate communication between a swarm service and a standalone container, or between two standalone containers on different Docker Daemons.



Docker Storage

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 Storage Solution

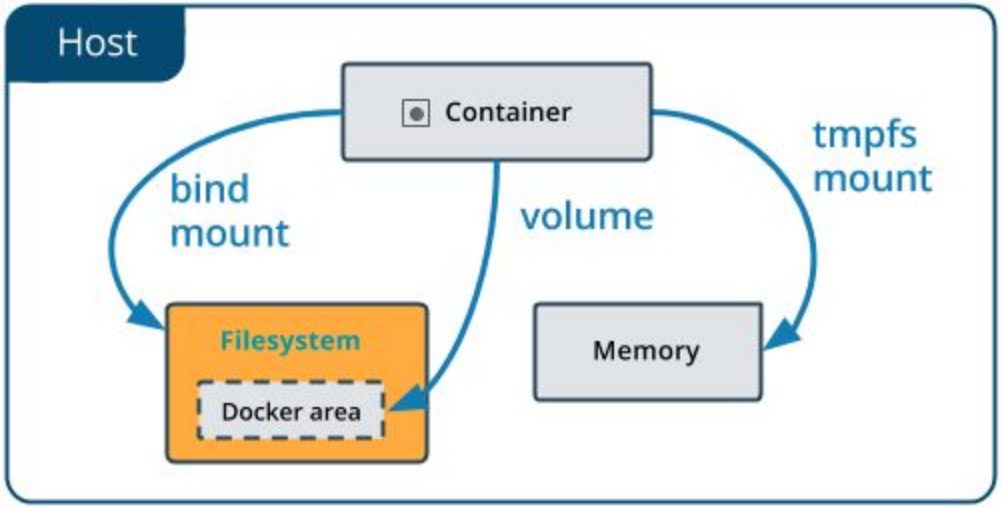
Docker has two options for containers to store files in the host machine, so that the files are persisted even after the container stops: *volumes*, and *bind mounts*.

- **Docker Bind Mounts**

- **Docker Volumes**

Docker Bind Mount

Bind mounts have been around 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 or relative path on the host machine. By contrast, when you use a volume, a new directory is created within Docker's storage directory on the host machine, and Docker manages that directory's contents.



Docker Volume

Volumes are the preferred mechanism for persisting data generated by and used by Docker containers. While bind mounts are dependent on the directory structure of the host machine, volumes are completely managed by Docker. Volumes have several advantages over bind mounts:

- Volumes are easier to back up or migrate than bind mounts.
- You can manage volumes using Docker CLI commands or the Docker API.
- Volumes work on both Linux and Windows containers.
- Volumes can be more safely shared among multiple containers.
- Volume drivers let you store volumes on remote hosts or cloud providers, to encrypt the contents of volumes, or to add other functionality.
- New volumes can have their content pre-populated by a container.

Docker Compose

Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a YAML file to configure your application's services. Then, with a single command, you create and start all the services from your configuration.

Using Compose is basically a three-step process:

1. Define your app's environment with a Dockerfile so it can be reproduced anywhere.
2. Define the services that make up your app in docker-compose.yml so they can be run together in an isolated environment.
3. Run docker-compose up and Compose starts and runs your entire app.

Container Orchestration

Container orchestration is all about managing the lifecycles of containers, especially in large, dynamic environments. Software teams use container orchestration to control and automate many tasks:

- Provisioning and deployment of containers
- Redundancy and availability 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
- Allocation of resources between containers
- External exposure of services running in a container with the outside world
- Load balancing of service discovery between containers
- Health monitoring of containers and hosts
- Configuration of an application in relation to the containers running it

Container Orchestration Platform

- **Docker Swarm**
- **Kubernetes**
- **Openshift**
- **Apache Mesos**

Docker Swarm

A Docker Swarm is a group of either physical or virtual machines that are running the Docker application and that have been configured to join together in a cluster.

Docker swarm is a container orchestration tool, meaning that it allows the user to manage multiple containers deployed across multiple host machines.

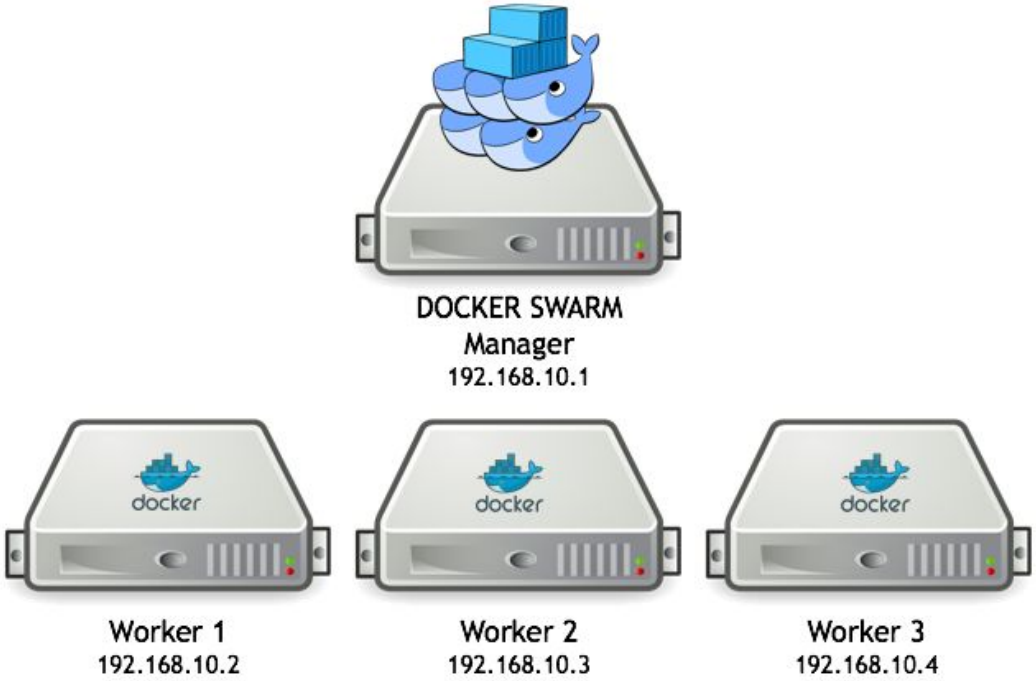
One of the key benefits associated with the operation of a docker swarm is the high level of availability offered for applications. In a docker swarm, there are typically several worker nodes and at least one manager node that is responsible for handling the worker nodes' resources efficiently and ensuring that the cluster operates efficiently.

Features of Docker Swarm

Some of the most essential features of Docker Swarm are:

- Decentralized access: Swarm makes it very easy for teams to access and manage the environment
- High security: Any communication between the manager and client nodes within the Swarm is highly secure
- Autoload balancing: There is autoload balancing within your environment, and you can script that into how you write out and structure the Swarm environment
- High scalability: Load balancing converts the Swarm environment into a highly scalable infrastructure
- Roll-back a task: Swarm allows you to roll back environments to previous safe environments

Docker swarm Architecture



Docker Swarm Nodes

Manager Node

The primary function of manager nodes is to assign tasks to worker nodes in the swarm. Manager nodes also help to carry out some of the managerial tasks needed to operate the swarm. Docker recommends a maximum of seven manager nodes for a swarm.

Worker Node

In a docker swarm with numerous hosts, each worker node functions by receiving and executing the tasks that are allocated to it by manager nodes. By default, all manager modes are also worker nodes and are capable of executing tasks when they have the resources available to do so.