EVIDEN

Docker Basics

19-02-2021

an atos business

EVIDEN

Introduction to Containerization

Docker Architecture

Docker Objects

Docker interaction commands

Docker Storage

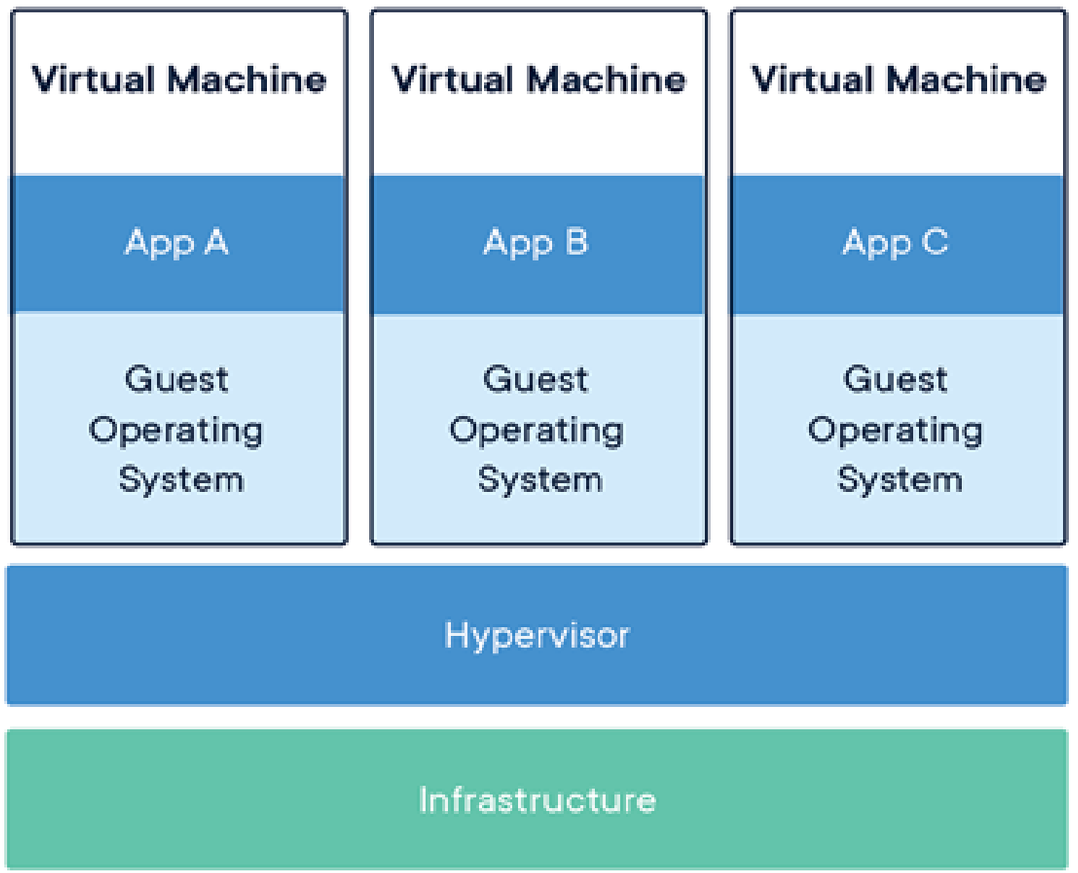
Docker Network

EVIDEN

01 Introduction to Containerization

What is Containerization

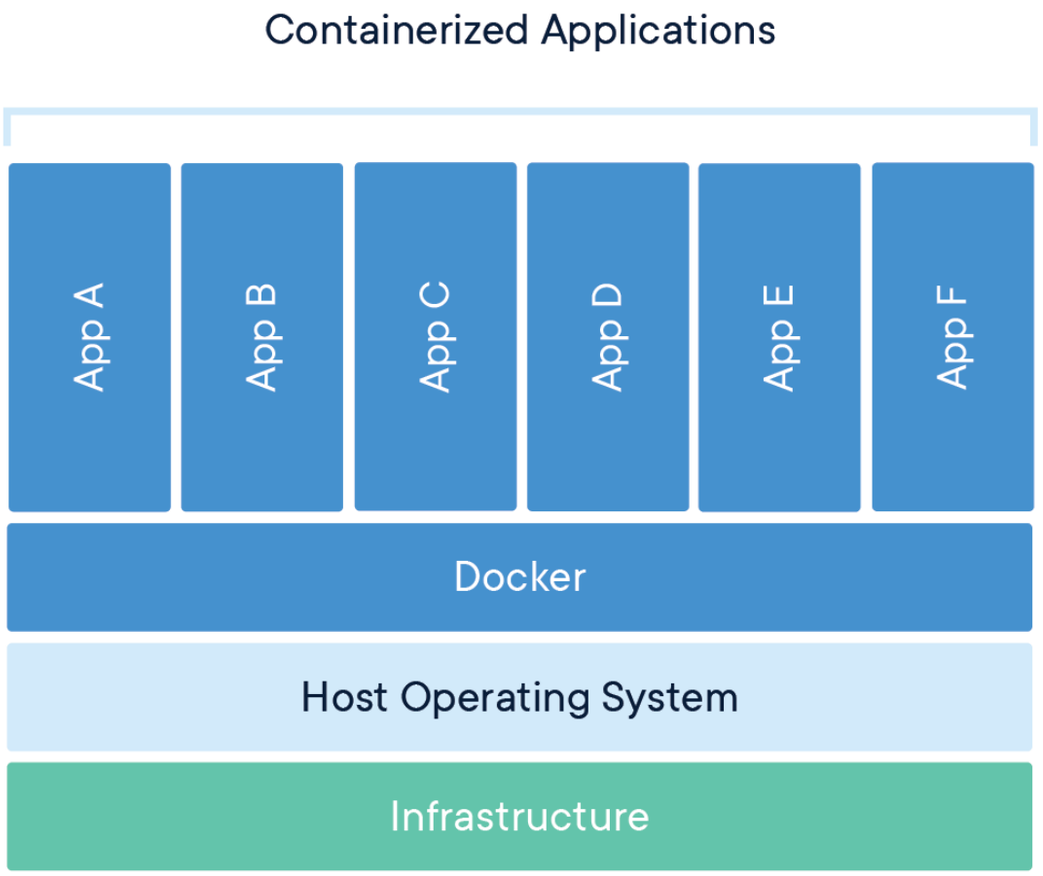
- Virtualization is a technique of importing a Guest OS on the top of host OS.
- This technique is revolution in the beginning because Developers run the different application in different VMs all running on same host.
- This eliminate the need of extra h/w resource and enable the backup allowing the recovery in failure conditions
- Lowering the total cost of foundation.



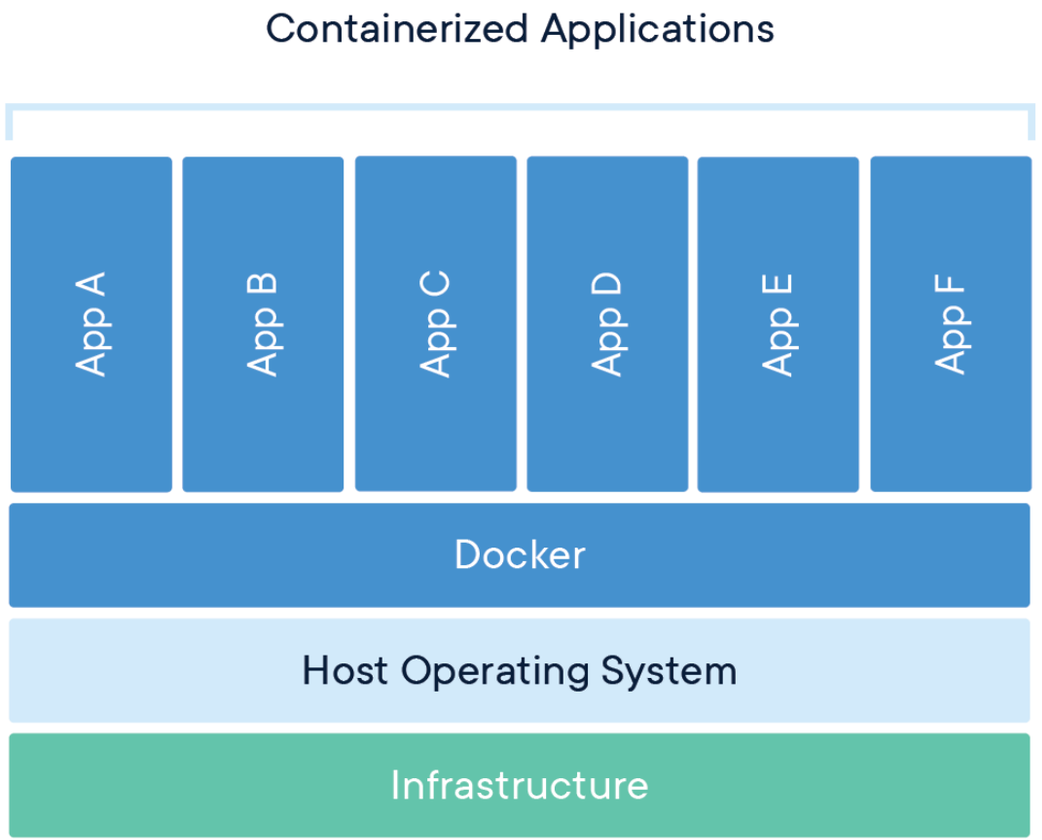
Containerization...

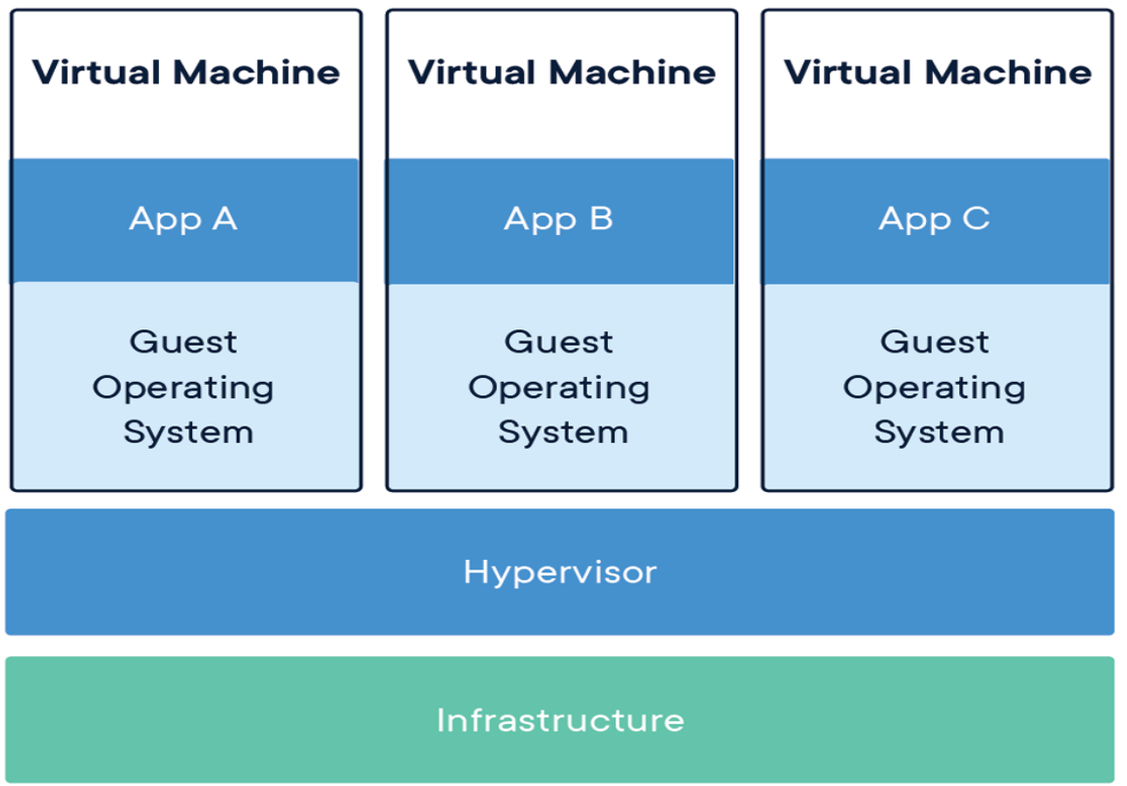
- Packaging Software into Standardized Units for Development, Shipment and Deployment is known as Containerization

- Docker is a software containerization platform, meaning you can build your application, package them along with their dependencies into a container and then these containers can be easily shipped to run on other machines.



Comparing VM and Containers





Comparing VM and Container

Criteria	Virtual Machine	Docker
Memory space	Occupies a lot of memory space	Docker Containers occupy less space
Boot-up time	Long boot-up time	Short boot-up time
Performance	Running multiple virtual machines leads to unstable performance	Containers have a better performance as they are hosted in a single Docker engine
Scaling	Difficult to scale up	Easy to scale up
Efficiency	Low efficiency	High efficiency
Portability	Compatibility issues while porting across different platforms	Easily portable across different platforms
Space allocation	Data volumes cannot be shared	Data volumes can be shared and reused among multiple containers

EVIDEN

Docker

Docker

What is Docker

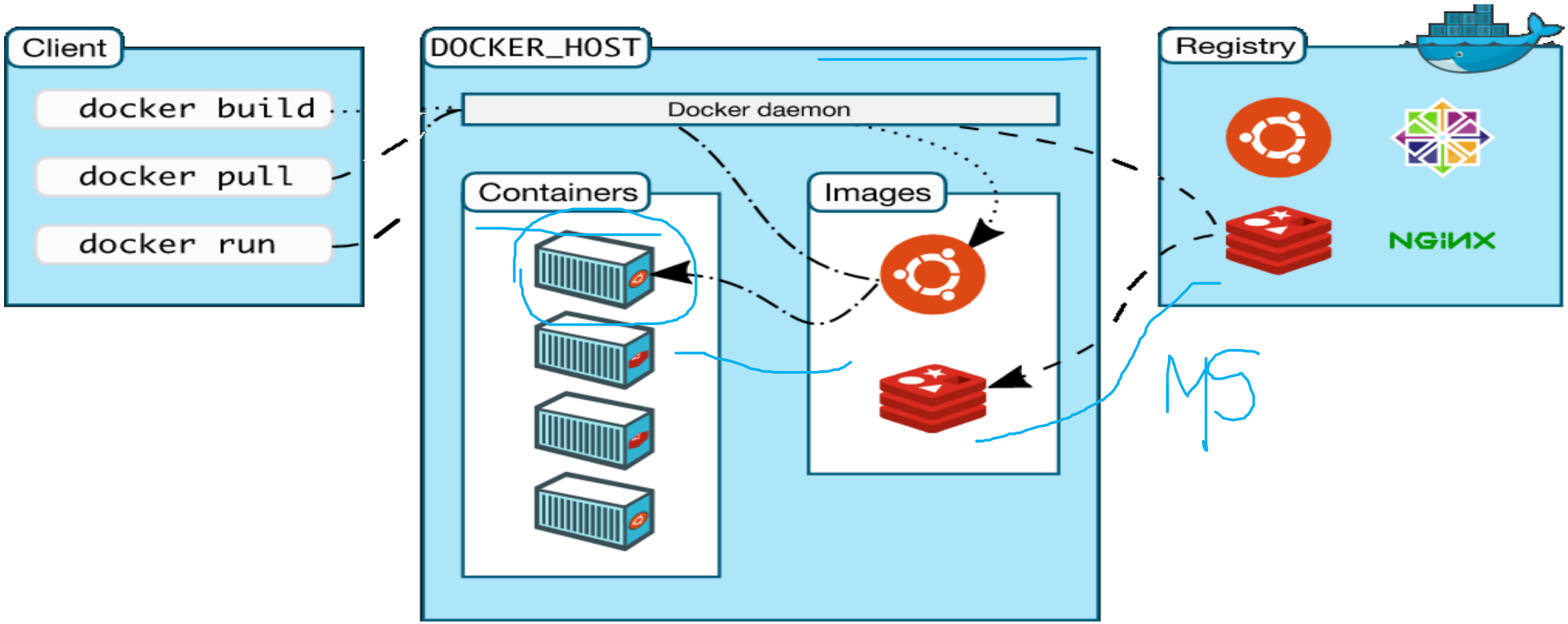
- Docker is an open-source platform that automates the deployment, scaling, and management of applications by packaging them into lightweight, portable containers.
- These containers include all the necessary components (like code, libraries, dependencies, and configurations) for an application to run consistently across different environments.

Application + Dependencies = Docker Container

EVIDEN

02 Docker Architecture

Docker Architecture



Docker Engine(dockerd)

- Docker Engine is a client-server application
  - Docker Engine: The core part of Docker, which is responsible for running and managing containers on a host machine. It's made up of:
  - Docker Daemon: The background service running on the host that manages Docker images, containers, networks, and storage.
  - Docker CLI(Docker client): The command-line interface that allows you to interact with Docker Daemon.

Docker Registries

- It is the location where the Docker images are stored. It can be a public docker registry or a private docker registry.
- Docker Hub is the default place of docker images, its stores' public registry.
- When you execute docker pull or docker run commands, the required docker image is pulled from the configured registry.
- When you execute docker push command, the docker image is stored on the configured registry.

EVIDEN

03 Objects in Docker

Objects in Docker

- When you use Docker, you are creating and using images, containers, networks, volumes, plugins, and other objects. This section is a brief overview of some of those objects.

- IMAGES
  - An image is a read-only template with instructions for creating a Docker container. Often, an image is based on another image, with some additional customization. For example, you may build an image which is based on the ubuntu image, but installs the Apache web server and your application, as well as the configuration details needed to make your application run.

  - You might create your own images or you might only use those created by others and published in a registry. To build your own image, you create a Dockerfile with a simple syntax for defining the steps needed to create the image and run it. Each instruction in a Dockerfile creates a layer in the image. When you change the Dockerfile and rebuild the image, only those layers which have changed are rebuilt. This is part of what makes images so lightweight, small, and fast, when compared to other virtualization technologies.

Containers

- Containers
  - A container is a runnable instance of an image. You can create, start, stop, move, or delete a container using the Docker API or CLI. You can connect a container to one or more networks, attach storage to it, or even create a new image based on its current state.
  - By default, a container is relatively well isolated from other containers and its host machine. You can control how isolated a container's network, storage, or other underlying subsystems are from other containers or from the host machine.
  - A container is defined by its image as well as any configuration options you provide to it when you create or start it. When a container is removed, any changes to its state that are not stored in persistent storage disappear.
  - Example docker run command

Docker Objects

- Services
  - Services allow you to scale containers across multiple Docker daemons, which all work together as a swarm with multiple managers and workers.

  - Each member of a swarm is a Docker daemon, and all the daemons communicate using the Docker API. A service allows you to define the desired state, such as the number of replicas of the service that must be available at any given time.

  - By default, the service is load-balanced across all worker nodes. To the consumer, the Docker service appears to be a single application. Docker Engine supports swarm mode in Docker 1.12 and higher.

EVIDEN

04 Docker interaction commands

Commands-starting and stopping containers

- docker start [CONTAINER]
- docker stop [CONTAINER]
- docker restart [CONTAINER]
- docker pause [CONTAINER]
- docker unpause [CONTAINER]
- docker kill [CONTAINER]
- docker ps
- docker logs [CONTAINER]
- docker inspect [OBJECT_NAME/ID]
- docke image ls
- docker history [IMAGE]

Docker Image commands

- docker build [URL]
- docker pull [IMAGE]
- docker push [IMAGE]
- docker commit [CONTAINER] [NEW_IMAGE_NAME]
- docker rmi [IMAGE]
- docker save [IMAGE] > [TAR_FILE]

EVIDEN

06 Docker Network

Docker Networking

- Docker takes care of the networking aspects so that the containers can communicate with other containers and also with the Docker Host.

- If you do an ifconfig on the Docker Host, you will see the Docker Ethernet adapter. This adapter is created when Docker is installed on the Docker Host.

- Docker's networking subsystem is pluggable, using drivers.

- Several drivers exist by default, and provide core networking functionality:
  - bridge
  - host
  - none
  - overlay
  - macvlan

Types of Networks

- **bridge**
  - The default network driver. If you don’t specify a driver, this is the type of network you are creating.
  - Bridge networks are usually used when your applications run in standalone containers that need to communicate.

- **host**
  - For standalone containers, remove network isolation between the container and the Docker host, and use the host’s networking directly.
  - host is only available for swarm services on Docker 17.06 and higher. See use the host network.

Types of Networks

- **overlay**
  - Overlay networks connect multiple Docker daemons together and enable swarm services to communicate with each other.
  - You can also use overlay networks to facilitate communication between a swarm service and a standalone container, or between two standalone containers on different Docker daemons.
  - This strategy removes the need to do OS-level routing between these containers

- **macvlan**
  - Macvlan networks allow you to assign a MAC address to a container, making it appear as a physical device on your network.
  - The Docker daemon routes traffic to containers by their MAC addresses.
  - Using the macvlan driver is sometimes the best choice when dealing with legacy applications that expect to be directly connected to the physical network, rather than routed through the Docker host's network stack.

Network Drivers

- **User-defined bridge networks** are best when you need multiple containers to communicate on the same Docker host.
- **Host networks** are best when the network stack should not be isolated from the Docker host, but you want other aspects of the container to be isolated.
- **Overlay networks** are best when you need containers running on different Docker hosts to communicate, or when multiple applications work together using swarm services.
- **Macvlan networks** are best when you are migrating from a VM setup or need your containers to look like physical hosts on your network, each with a unique MAC address.
- **Third-party network plugins** allow you to integrate Docker with specialized network stacks.

Networking Commands

- docker network ls
- docker network rm [NETWORK]
- docker network inspect [NETWORK]
- docker network connect [NETWORK] [CONTAINER]
- docker network disconnect [NETWORK] [CONTAINER]

Hands on

Docker Hands on

- Docker Installation
- Docker Image Commands
- Dockerfile
- Docker Container Commands
- Docker Volume Commands
- Docker Network Commands
- Docker-Compose
- Docker Swarm commands
- Docker Node commands
- Docker Service commands
- Docker Stack commands

EVIDEN

05 Docker Storage

Storage in Docker

- 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*. If you're running Docker on Windows you can also use a *named pipe*.
- Keep reading for more information about these two ways of persisting data.

Storage in Docker

- No matter which type of mount you choose to use, the data looks the same from within the container. It is exposed as either a directory or an individual file in the container's filesystem.
- An easy way to visualize the difference among volumes, bind mounts, and tmpfs mounts is to think about where the data lives on the Docker host.
- **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.

EVIDEN

02 Docker Compose

Docker Compose

- Docker Compose is a tool for defining and running multi-container Docker applications.
- It allows you to define a multi-container application using a YAML file (docker-compose.yml), making it easier to configure, manage, and run complex systems that require multiple services.
- Key Concepts:
- Services: These are individual containers that make up your application (e.g., a web server, database, or cache). Each service is defined in the docker-compose.yml file.

- Networks: Docker Compose automatically creates a network for your services to communicate with each other, so they can interact seamlessly.

- Volumes: Docker Compose can also manage shared storage between containers (such as a database file or user data), making persistent storage easier.

- Configurations: With Compose, you can define environment variables, ports, networks, volumes, and more for each container in the application.

EVIDEN

02 Docker File

Images from exiting Containers

- One of Docker's powerful features is the ability to create new images from an existing, running container.
- This is useful when you want to save the current state of a container (including its filesystem, configurations, and installed software) as a reusable image.
- You can then use this image to create new containers that start with the exact same state.
- Creating images from existing Docker containers can be accomplished in several ways,

- **Using docker commit**
- The most direct way to create an image from a running or stopped container is by using the docker commit command.

- **Using a Dockerfile with docker build**

Images from exiting Containers

- **Docker Push**
- The docker push command is used to upload (or push) a local Docker image to a remote container registry. This allows you to share your images with other users or deploy them on different systems.

- docker push [OPTIONS] NAME[:TAG]

- NAME: The name of the image, which typically includes the registry URL, repository name, and image name.
- TAG: An optional tag to specify a version of the image. If no tag is provided, Docker uses latest by default.
- OPTIONS: Various optional flags that can be used with the command.

Image from exiting container

- **Docker Pull**
- The docker pull command is used to download (or pull) Docker images from a remote container registry to your local system. This command retrieves images that you can then run as containers.

- docker pull [OPTIONS] NAME[:TAG|@DIGEST]

- NAME: The name of the image to pull, including the registry name if applicable.
- TAG: An optional tag to specify a particular version of the image. If no tag is provided, Docker defaults to latest.
- DIGEST: A unique identifier for a specific image version, usually represented as a SHA256 hash.

Hands On

- Docker commit
- Dockerfile
- Docker push
- Docker pull

EVIDEN

05 Docker Stack

Docker Stack

- Docker Stack is a feature of Docker Swarm that allows you to deploy and manage multi-container applications as a single unit. It is designed for use in a Swarm environment, where you can scale services and manage them across multiple nodes.
- **Key Features of Docker Stack**

- **Multi-Service Deployment**: Deploy multiple services with a single command.
- **Load Balancing**: Automatically balances traffic across multiple replicas of a service.
- **Service Discovery**: Services can communicate easily within a defined network.
- **Rolling Updates**: Supports updates with minimal downtime.

EVIDEN