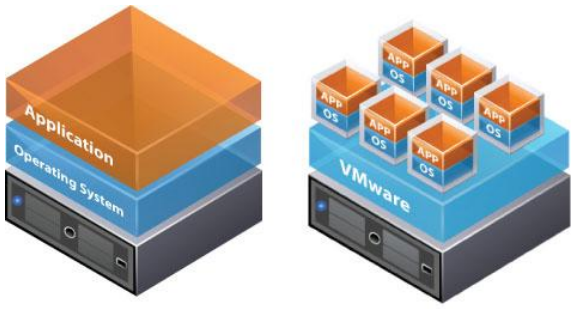
DOCKER

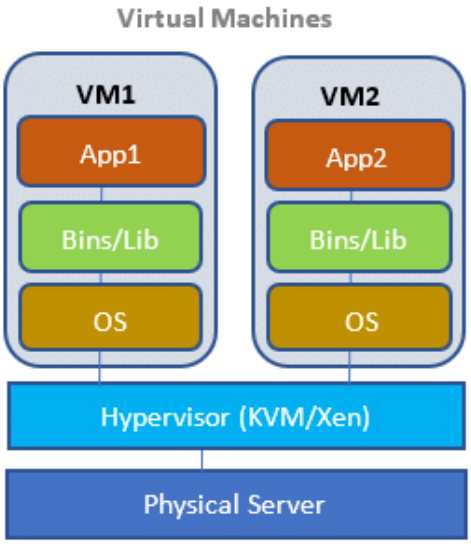
Introduction

What is Virtualization ?

- Virtualization is technology that lets you create useful IT services using resources that are traditionally bound to hardware.
- It allows you to use a physical machine's full capacity by distributing its capabilities among many users or environments.

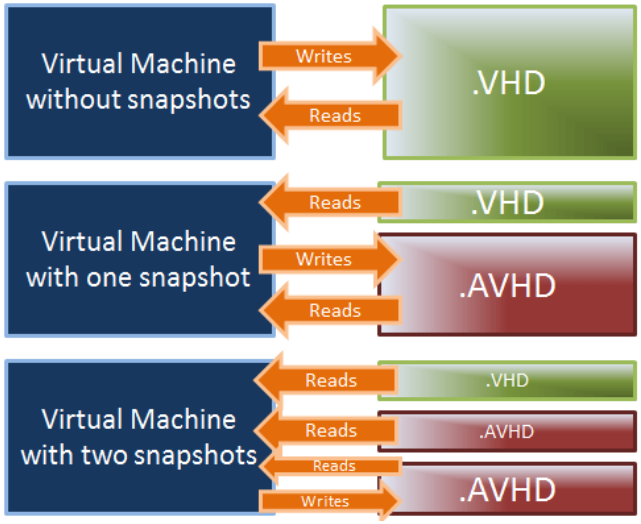


- كان في الطبيعى عندى في تكنولوجى الvirtualization انى بكون محتاج حاجتين ، الأولى هي Hyper-Visor الLayer نفسها اللى هتتعامل مع physical hardware الـ ، والتانية هي الSoftware بتاع الManagement نفسه زى VMWare Workstation مثلا.
- طيب انا في فكرة الVirtualization كان عندى wasting في الresources بطريقة كبيرة لأن كل VM بيكون نازل عليها OS كامل وحاجات كتير مساعدة ليه عشان بس انه يخدم في الاخر على الApp اللى عليه ، خصوصا كمان انه من ناحية الsecurity والperformance الأفضل انه يكون لكل App موجود VM خاصة بيه لوحده.
- حاجة كمان انى مثلا لو شغال على scale كبير ومثلا الOS دة مش فرى زى Windows او RHEL كدة انا هبقى محتاج ادفع license على كل VM وكدة التكلفة بتزيد برضو.
- في كمان نقطة انى لو جيت بصيت من حتة الdevops مثلا وقولت انا عاوز اعمل deployment لكذا VM مثلا عشان هنزل عليهم Apps ، انا هنا عندى مشكلة انى هبقى محتاج اعمل configurations كتيرة اوى لكل VM عشان تناسب كل App ، حتى لو عملت الموضوع بطريقة احسن شوية عن طريق template او كدة فبرضو انا هكون محتاج حد فاهم اوى ومتخصص يعمل الموضوع دة ، وكدة كدة الprocess دى هتكون ليها ليفل من الصعوبة عالى شوية.



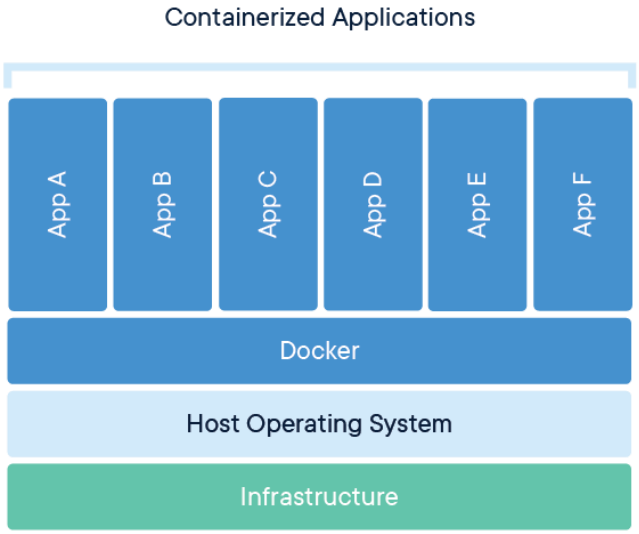
- فاللى انا محتاجه دلوقتى انى لو قدرت افصل الApp بتاعى وشوية الdependencies بتوعه واخدهم لوحدهم كدة اشتغل عليهم هيوفر عليا كتير ، ودى اول نقطة محورية في فايدة انى انتقل من الvirtualization للcontainerization.

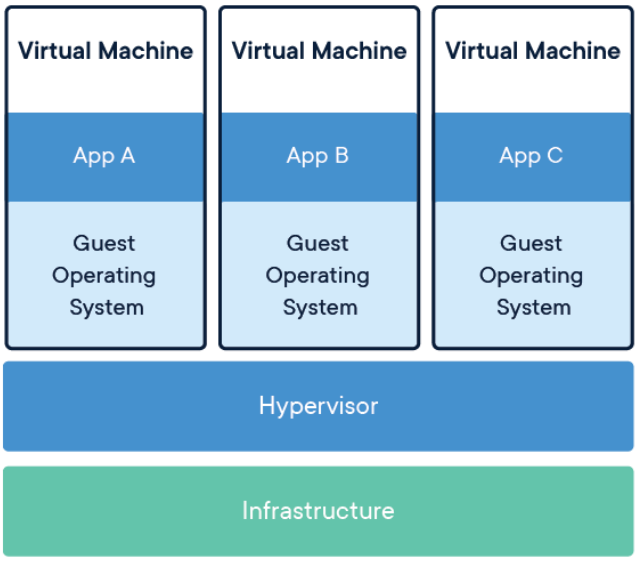
- تانى نقطة محورية في الVirtualization هي انى عندى مفهوم لو مثلا عاوز اعمل اكتر من VM لكذا App ، ممكن اعمل VM واحدة واظبط فيها كل الConfigurations بتاعتى اللى عاوزها وبعدين افضل اعمل منها كوبى بيست ع حسب مانا عاوز وكل كوبى جديدة بدخل عليها وبعدل في الConfigurations بتاعتها ع حسب كل App هيشتغل عليها ، لكن كان في طريقة تانية احسن زى مثلا في الHyper-V كدة كنت ممكن اعمل Base Disk اللى هو بيكون VHD. وبعد كدة بعمل Snapshot فيه مثلا التعديلات اللى عاوز اعملها اللى هو AVHD. او ساعات بيتقال عليه الDifferential disk وساعتها الSnapshot دة بيكون child للdisk parent ، يعنى من الاخر ممكن اعمل base disk واحد وكل VM بقى تاخد منه وبعدين تاخد الSnapshot بتاعها المخصوص ليها وتبدء تشتغل.



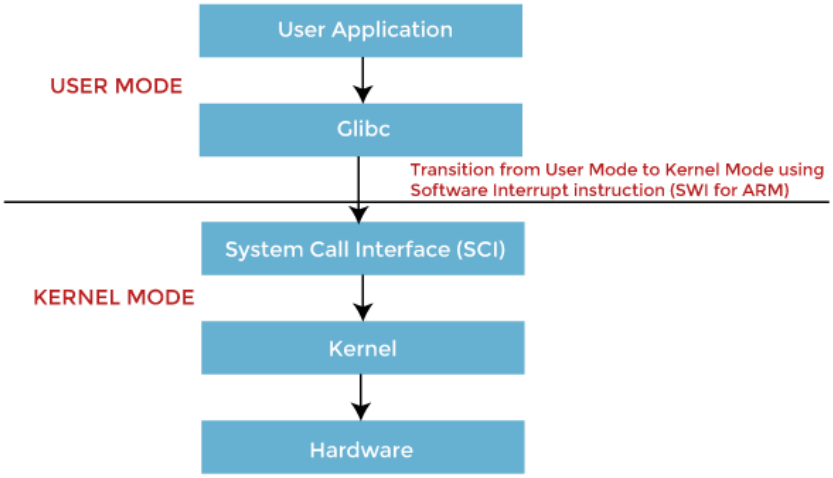
What is Containerization ?

- With containerization, one physical server can run many applications, and applications in containers do not need to know about the operating system (OS).
- Containers are smaller than VMs and require fewer hardware resources. You can run more containers on one server and many different versions of applications on one operating system.
- Each container includes the application code and other files that are required to run the application.
- These other files are called the application libraries. Another piece of software called the container engine manages the containers.

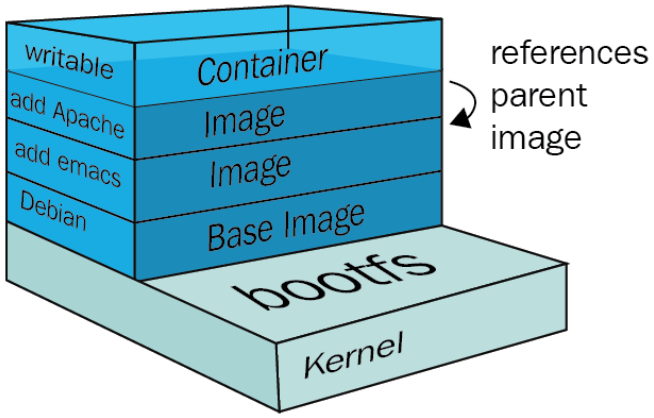




- لو جيت بصيت على الOS نفسه هلاقى انى ممكن اقسمه لجزئين ، الأول هو الKernel mode ودة اللى بيكون مسئول عن تشغيل الOS على الHardware اللى موجود عليها ، والتانى هو User mode ودة بقى اللى بيكون مسئول عن تجهيز مثلا الGUI وملفات كل يوزر المختلفة عن التانى بقى وهكذا.



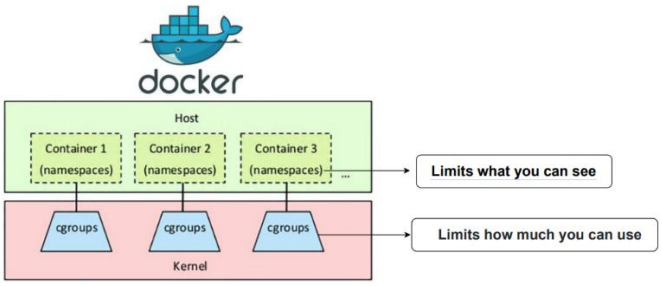
- اللى انا محتاجه عشان الContainers هو جزء الKernel mode بس.
- فكدة اول نقطة محورية في فكرة الContainer هي انه بيشتغل على الOS Kernel علطول مش محتاج باقى مكونات الOS ، وبالتالي انا كدة وفرت انى بس محتاج الOS الواحد اللى موجود عندى سواء كان الVirtual بتاع الVM او الHost بتاع الPhysical server نفسه.
- تانى نقطة محورية في الContainer وهو دة اللى بيحصل بالظبط ، ان الcontainer بيكون بادئ يشتغل على الKernel بس اللى موجود وبعد كدة يبدء يضيف layers من التعديلات اللى هو محتاجها ع حسب الApp اللى هيشغله محتاج ايه ، مثلا Layer تعمل OS Specifications ، Layer تانية تنزل Dependencies معينة ، Layer تالتة تعدل حتى في الLayers اللى قبلها ، وهكذا .



- الموضوع دة هيمشى معايا كويس لأن كدة كدة الKernel بيكون موحد ، يعنى كل أنواع الLinux Distributions هما في الاخر بيستخدموا نفس الLinux Kernel ، وكل أنواع الWindows برضو بيستعملوا الWindows Kernel ، فانا كدة قللت اوى الاعتمادية على الOS لأن كل اللى محتاجه منه هو الKernel بس.
- الفكرة دى مش جديدة هي أصلا موجودة من السبعينات في قلب الKernel بتاع Unix نفسه ، يعنى الKernel بيستعمل نفس الفكرة دى internally جواه في شغله هو عن طريق cgroups و namespaces.
- بناء على الكلام دة فانا عندى نوعين بس من الContainers وهما Linux Containers و Windows Containers ، مفيش mac Containers.
- لكن بالنسبة للContainer Software نفسه اللى هو مثال عليه Docker ممكن ينزل على اى OS من التلاتة ، **ولو انا منزله على Windows او Mac فهو هيعمل Linux VM عشان يشغل عليها الLinux Containers لو انا كنت عاوز اشغلها ، زى مثلا في الويندوز بيعمل الLinux VM دى عن طريق Hyper-V أو WSL.**

cgroups and Namespaces

- Containers achieve isolation, resource control, and security through two key Linux kernel features:
  - Namespaces – Provide process isolation.
  - cgroups (Control Groups) – Limit and manage resource usage.
- These features allow containers to run independently on the same host while sharing the same kernel.

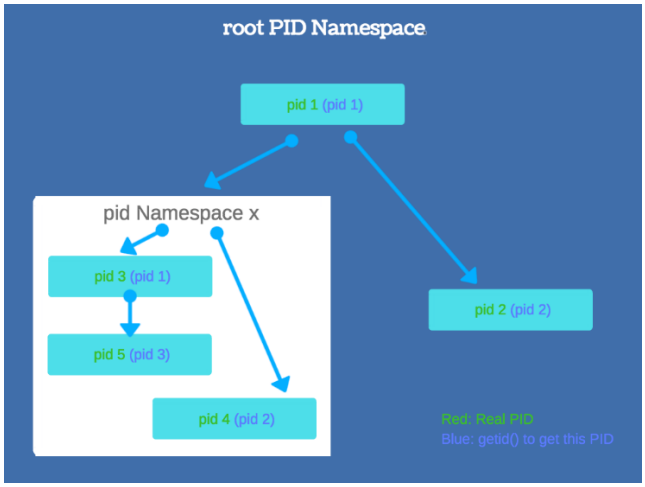


Namespaces: Process Isolation

- Namespaces provide isolation between containers and the host system by creating separate instances of global system resources.
- Each namespace ensures that processes within a container have their own isolated view of the system, preventing interference with other containers or the host.

Namespace	Purpose	Example in Containers
PID (Process ID)	Isolates process trees	Containers have their own process tree, preventing them from seeing host processes.
NET (Networking)	Isolates network stack	Each container can have its own IP, interfaces, and routing rules.
MNT (Mounts)	Isolates file system views	Containers see only their own mounted directories.
UTS (Hostname & Domain)	Provides separate hostnames	Each container can have a different hostname than the host.
IPC (Interprocess Communication)	Isolates shared memory	Prevents processes in different containers from interfering with each other.
USER (User IDs)	Isolates user and group IDs	Allows containers to have separate user privileges from the host.

- **How Containers Use Namespaces:**
  - When a container starts, the container runtime (e.g., Docker, containerd) creates a new set of namespaces for the container.
  - Processes inside the container are isolated from the host and other containers, ensuring they only see their own resources.
  - For example, a container's process might think it is running as root (UID 0), but on the host, it is mapped to a non-privileged user.

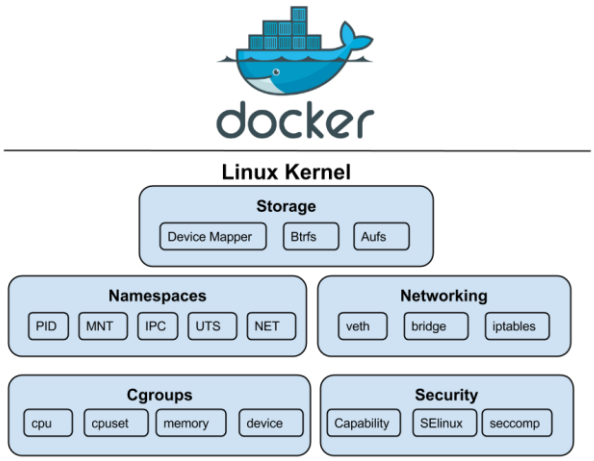


cgroups: Resource Management

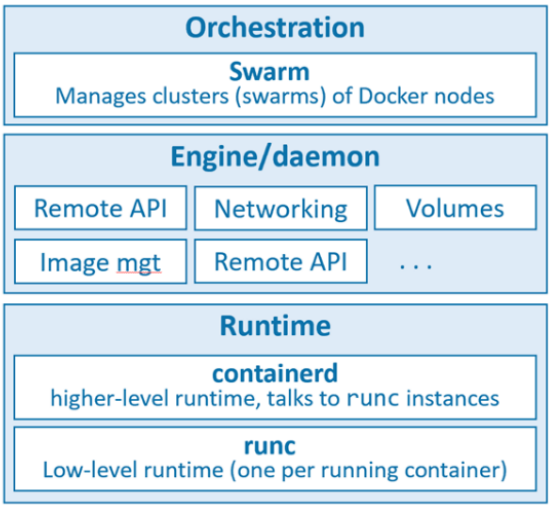
- cgroups control how much CPU, memory, disk I/O, and network bandwidth a container can use.
- **Key Features of cgroups:**
  - Resource Limiting: Set limits on CPU, memory, disk I/O, and network bandwidth for containers.
  - Prioritization: Allocate more resources to high-priority containers.
  - Accounting: Monitor resource usage by containers.
  - Control: Freeze, checkpoint, and restart groups of processes.
- **How Containers Use cgroups:**
  - When a container starts, the container runtime creates a cgroup for the container.
  - Resource limits (e.g., CPU shares, memory limits) are applied to the cgroup.
  - Processes inside the container are assigned to the cgroup, ensuring they adhere to the specified resource constraints.
  - For example, if a container is limited to 512MB of memory, the kernel will enforce this limit and kill processes if they exceed it.
- **Example: Limiting CPU & Memory with Docker (which uses cgroups)**

```
docker run --memory=500m --cpus=1.5 mycontainer
```

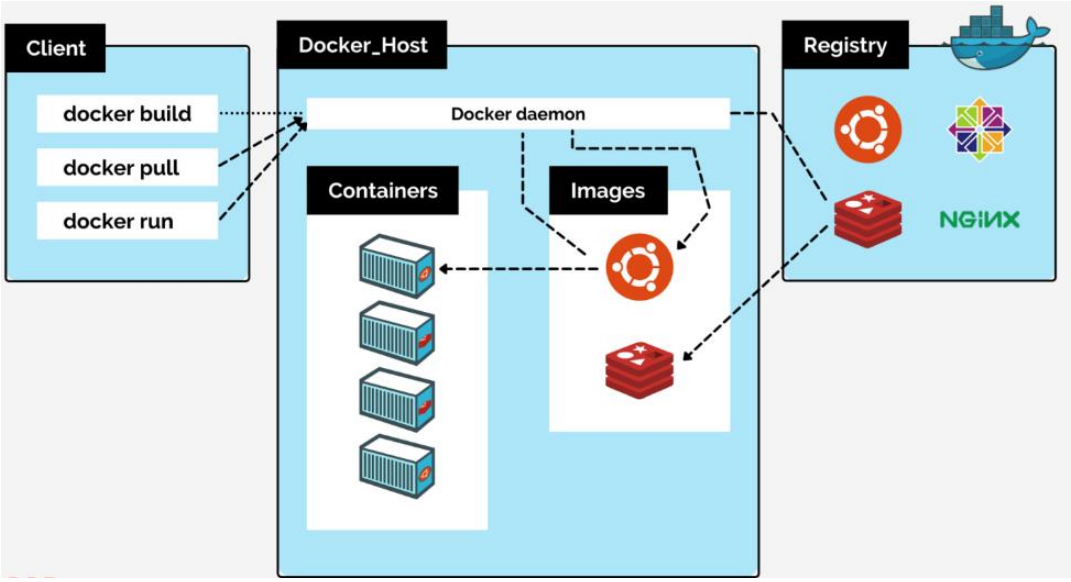
Docker Technology



- Docker is a platform and set of tools that use containerization technology to enable developers to build, package, and deploy applications in a consistent and portable manner.
- It allows applications and their dependencies to be bundled into lightweight, standalone containers that can run consistently across different computing environments, such as development, testing, and production.



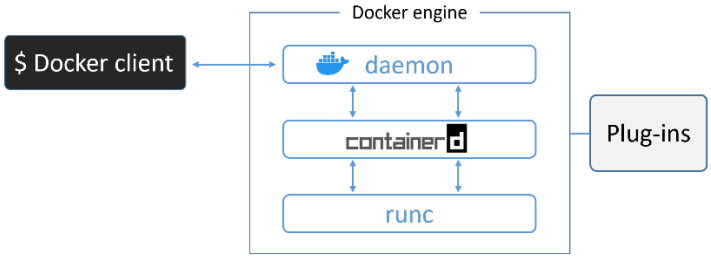
- Docker Engine is the core component of Docker, consisting of the Docker Daemon, Docker Client, and other related tools. It manages the lifecycle of Docker containers.



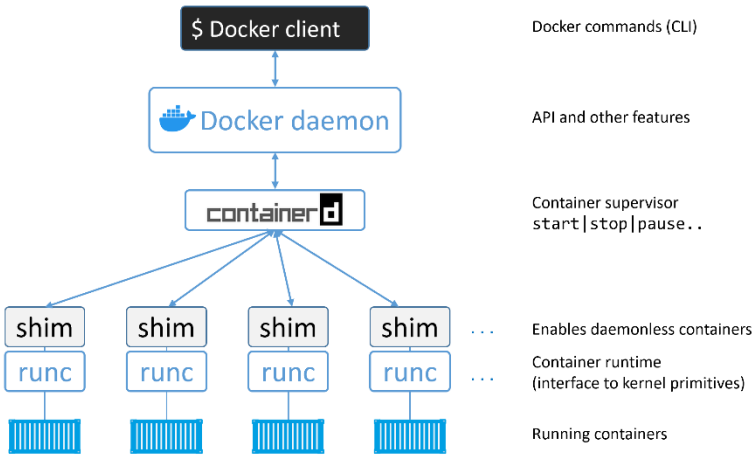
- Orchestration tools are used to manage and scale Docker containers across multiple hosts.
- They help in deploying, managing, and scaling containerized applications in a cluster.
- Docker Swarm is Docker's native orchestration tool, while Kubernetes is a widely used orchestration platform that can work with Docker containers.

Docker Engine Architecture

- Docker Engine is the core software that enables containerization on a host system.
- It consists of several components that work together to manage containers.



- One huge benefit of this model is that container runtime is decoupled from the Docker daemon (daemonless containers).
- You can perform maintenance and upgrades on the Docker daemon without impacting running containers



Docker Client

- ***Function:***
  - The primary user interface (CLI) to Docker.
  - It accepts commands from the user and communicates them to the Docker Daemon.
  - The binary file path is `/usr/bin/docker`
- ***Interaction:***
  - Uses REST APIs to interact with the Docker Daemon.

Docker Daemon (dockerd)

- ***Function:***
  - Listens for Docker API requests and manages Docker objects like images, containers, networks, and volumes.
- ***Role:***
  - Core service responsible for running and managing containers.

Containerd

- ***Function:***
  - An industry-standard core container runtime that manages the complete container lifecycle of its host system, including image transfer, container execution, and supervision.
- ***Features:***
  - Provides APIs for managing containers, storage, and network resources.

Shim

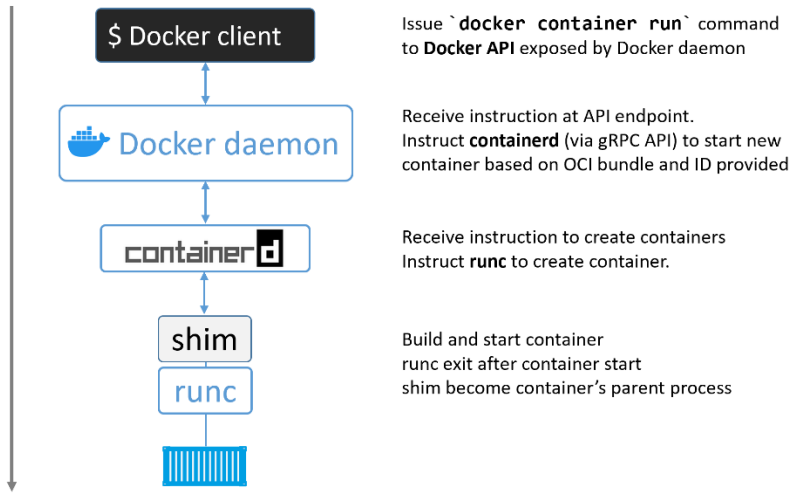
- ***Function:***
  - A lightweight process that sits between *containerd* and *runc*.
  - It enables "daemonless" containers, meaning containers can continue to run independently even if the Docker daemon stops.
  - The shim allows containers to run as child processes of the *containerd* daemon
  - It allows *runc* to exit after starting the container.
- ***Purpose:***
  - Ensures that the container's lifecycle is independent of *runc*, enabling better resource management and container management.

runc

- ***Function:***
  - A lightweight, portable container runtime.
  - It is responsible for spawning and running containers according to the Open Container Initiative (OCI) specifications.
  - It interfaces directly with the kernel to execute containers
- ***Role:***
  - Executes the containers using Linux kernel features like namespaces and *cgroups*.

Workflow Example

- **User Interaction:** The user interacts with the Docker Client, sending commands like docker run.
- **Command Processing:** The Docker Client sends API requests to the endpoint exposed by Docker daemon (*/var/run/docker.sock* on Linux).
- **Daemon Coordination:** The Docker Daemon processes the requests, interacting with *containerd* to manage the container's lifecycle.
- **Container Execution:** *containerd* uses *runc* to create and start the container.
- **Lifecycle Management:** A *shim* process is created to manage the container's lifecycle, keeping it running independently from *runc*.



Docker Concepts

Image

- A unit of packaging that contains everything required for an application to run.
- It contains Application code, Application dependencies and OS Constructs.
- A Docker image is composed of a series of read-only layers stacked on top of each other.
- Each layer represents a set of file system changes, and every Dockerfile instruction adds a new layer to the image.
- These layers are cached by Docker, enabling quicker image builds and efficient use of resources.
- Images are considered build-time constructs.
    - أي command انا بكتبه وبيتنفذ اثناء مرحلة بناء الimage هي تعتبر build-time commands.

Image Registry

- An image registry is a storage and distribution system for Docker images.
- It is a central place where Docker images are stored, managed, and distributed to different environments.
- Docker images are the building blocks for containers, and an image registry provides a way to share these images across different teams or organizations.
- ***Registry***
    - **Definition:** The actual service that stores Docker images and makes them available for download (pull) or upload (push).
    - **Example:** Docker Hub, Amazon Elastic Container Registry (ECR), Google Container Registry (GCR), and private registries.
        - Organizations can host their own registries using solutions like Harbor, JFrog Artifactory, or the open-source Docker Registry.
- ***Repository***
    - **Definition:** A collection of related Docker images, often representing different versions of the same application. Each image in a repository is identified by a tag (e.g., v1.0, latest).
    - **Usage:** Repositories help organize and manage different versions of an application. For example, you might have a *myapp* repository with tags like v1.0, v1.1, and v2.0.
- ***Image Tags***
    - **Definition:** Tags are labels attached to images within a repository that identify specific versions of the image. They allow users to specify which version of an image they want to use.
    - **Example:** The latest tag is commonly used to indicate the most recent stable version of an image.
    - يبقى كدة ممكن نقول ان اسم الimage هو عبارة عن <repo_name>:<tag> ، الكلام دة مع الOfficial Repos المعروفة مثلا زى python.
    - لكن باقى الimages اللى موجودة في Unofficial Repos المفروض بيكون اسمها <account_name>/<repo_name>:<tag> ، يعنى الdocker بيروح الأول للاكونت وبعدين يشوف الrepos اللى جواه وبعدين يدخل جوة repo معين ويجيب image معينة عن طريق الtag.

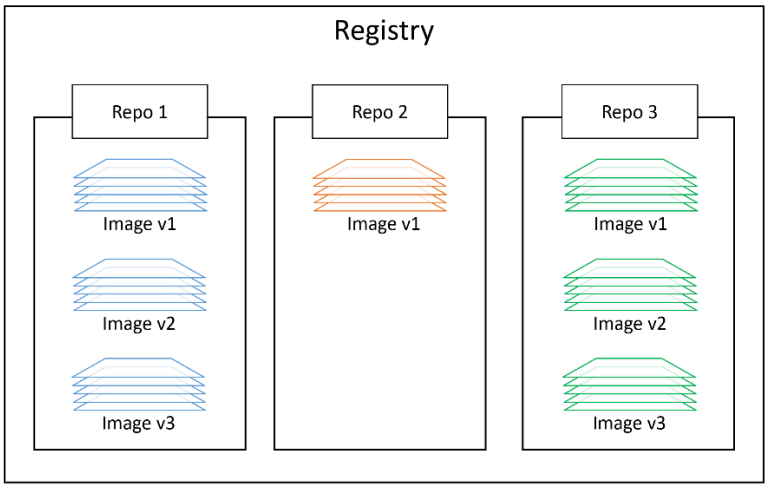
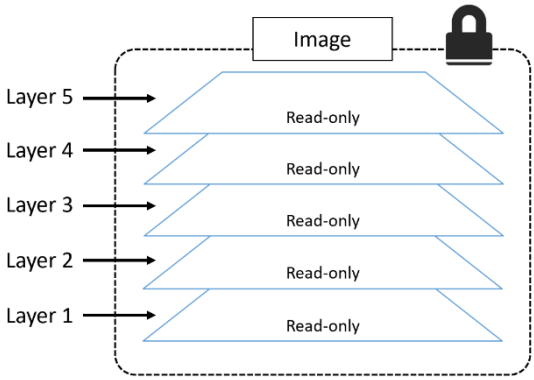
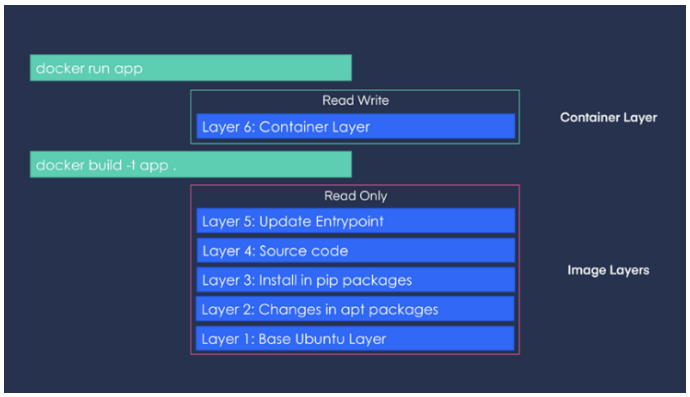


Image Layers

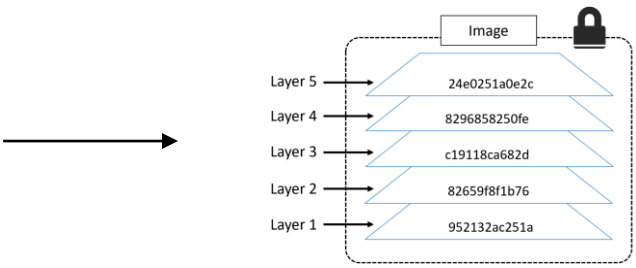
- A Docker image is a bunch of loosely-connected read-only layers.
- Each layer is a read-only file system that contains changes such as adding or modifying files, installing packages, or setting environment variables.
- When a container is run from an image, it uses these layers to create the container's filesystem.
- Layers are created during the image build process, typically based on instructions in a Dockerfile.
- When a container is created from an image, a new writable layer (container layer) is added on top of these read-only layers.





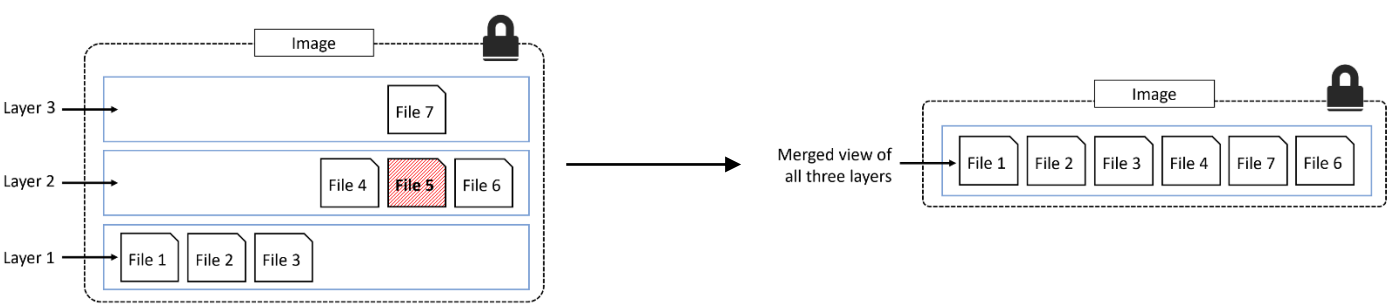
- Any changes made while the container is running (e.g., adding files, installing software) are made in this writable layer.
- Since layers are immutable and reusable, they can be shared across multiple images.

```
$ docker image pull ubuntu:latest
latest: Pulling from library/ubuntu
952132ac251a: Pull complete
82659f8f1b76: Pull complete
c19118ca682d: Pull complete
8296858250fe: Pull complete
24e0251a0e2c: Pull complete
Digest: sha256:f4691c96e6bbaa99d...28ae95a60369c506dd6e6f6ab
Status: Downloaded newer image for ubuntu:latest
docker.io/ubuntu:latest
```



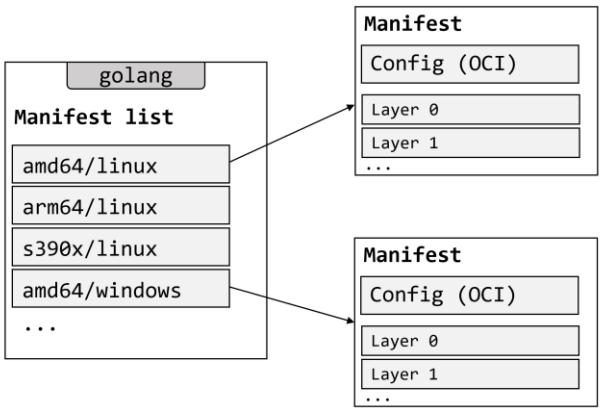
- في نقطة مهمة بخصوص الlayers بتاعة الimage ، انا ممكن اضغط الlayers بتوعى او اعمل ليهم squash كدة او ممكن اخليهم منفصلين ، لو خليتهم منفصلين كدة انا هستفيد من موضوع الcaching اللى جوة الdocker لأن هو جيت انزل image فيها Layers موجودة عندى أصلا كدة هي مش هتنزل تانى وهاخد اللى موجودة عندى.

- ***Layer Caching:***
  - Docker caches layers to improve build efficiency.
  - If a layer doesn't change between builds, Docker can reuse it rather than rebuild it.
  - This caching mechanism makes subsequent builds faster by only rebuilding the layers that have changed.
- Docker uses a storage driver in order to stack and merge all layers and present them as a single image.



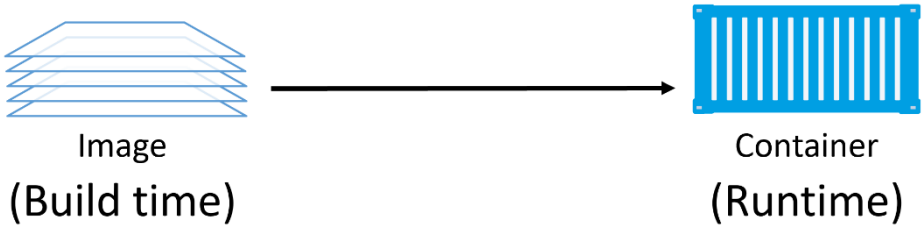
Manifest & Manifest List

- Manifests and manifest lists are metadata files that describe the contents and properties of Docker images.
- They play a crucial role in how Docker handles images, especially when dealing with multi-platform support.
- ***Manifest***
  - A manifest is a JSON file that provides detailed information about a specific Docker image, including its layers, configuration, and other metadata.
  - It is essentially the blueprint of an image and is used by the Docker engine to understand how to construct the image.
- ***Manifest List (or Multi-Architecture Image)***
  - A manifest list (also known as a multi-architecture image or fat manifest) is a higher-level manifest that references multiple image manifests, each tailored for a different platform or architecture.
  - This allows Docker to pull the correct image version based on the platform it is running on, enabling seamless multi-platform support.
  - When building multi-platform images, you can use Docker's build tools (like *buildx*) to create and push manifest lists to an image registry.



Container

- A container is runnable instance of an image.
- Containers are lightweight, portable, and self-sufficient units that package an application and its dependencies (libraries, configuration files, binaries) together.
- Containers share the host system's kernel but run in isolated user spaces.
- Containers are considered runtime constructs.
  - أي command انا بكتبه وبيتنفذ اثناء مرحلة بناء container هي تعتبر runtime commands.



- نقطة مهمة في موضوع الcontainers هو ان الcontainer أصلا بيكون معمول عشان خاطر process واحدة يتعملها run بس ، لو الprocess دى اتقفلت كدة الcontainer معدش ليه لازمة وهيحصله termination.
- الكلام دة طبعا عكس الVM لأن في الVM بيكون عندى OS عليه كل الpackages والApplications اللى انا محتاجها ، الVM دى بقى هتفضل شغالة حتى لو انا أصلا مش مشغل اى application ولا بعمل اى حاجة.
- يبقى لو انا عملت container حتى لو كان بسيط خالص مجرد OS بس زى مثلا انى انزل fedora image لازم يكون في process شغالة عليه وليكن مثلا bash.
- ممكن اعمل run لprocess تانية جوة الcontainer غير الprocess اللى هو أصلا معمول عشانها ، يعنى كمثال:
  - هنا انا بعمل container من python image وبفتح interactive terminal معاها ، فطبعا علشان هي python interactive terminal هيفتح على الcommand بتاع python3:

```
[ec2-user@ip-10-10-20-86 ~]$ docker container run -it --name my-python-cont python
Python 3.12.4 (main, Jul 10 2024, 19:07:58) [GCC 12.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello from container")
Hello from container
>>> 5+20
25
>>> exit()
[ec2-user@ip-10-10-20-86 ~]$ docker container ls -a
CONTAINER ID   IMAGE     COMMAND     CREATED          STATUS                      PORTS     NAMES
050f306def7a   python    "python3"   37 seconds ago   Exited (0) 7 seconds ago              my-python-cont
```

- وساعتها لما عملت exit للprocess اللى كان مفتوح عشانها اللى هي python3 الcontainer اتعمله termination.
- لكن ممكن مثلا اعمل container بprocess تانية زى مثلا انى افتح bash ، هنا بقى الprocess الأساسية اللى قايم عليها الcontainer هي الbash:

```
[ec2-user@ip-10-10-20-86 ~]$ docker container run -it --name my-python-cont-2 python /bin/bash
root@06e9ba504833:/# ps
    PID TTY          TIME CMD
      1 pts/0    00:00:00 bash
      7 pts/0    00:00:00 ps
```

- ساعتها ممكن افتح بقى اى process تانية براحتى واعمل للprocess اللى فتحتها دى termination عادى وساعتها الcontainer مش هيقفل لأن الprocess بتاعته الرئيسية لسه مفتوحة:

```
root@06e9ba504833:/# python3
Python 3.12.4 (main, Jul 10 2024, 19:07:58) [GCC 12.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 5+20
25
>>> exit()
root@06e9ba504833:/# ps
    PID TTY          TIME CMD
      1 pts/0    00:00:00 bash
      9 pts/0    00:00:00 ps
root@06e9ba504833:/#
```

```
root@06e9ba504833:/# exit
exit
[ec2-user@ip-10-10-20-86 ~]$ docker container ls -a
CONTAINER ID   IMAGE     COMMAND       CREATED          STATUS                      PORTS     NAMES
06e9ba504833   python    "/bin/bash"   2 minutes ago    Exited (0) 3 seconds ago              my-python-cont-2
050f306def7a   python    "python3"     11 minutes ago   Exited (0) 10 minutes ago             my-python-cont
```

Self-healing containers with restart policies

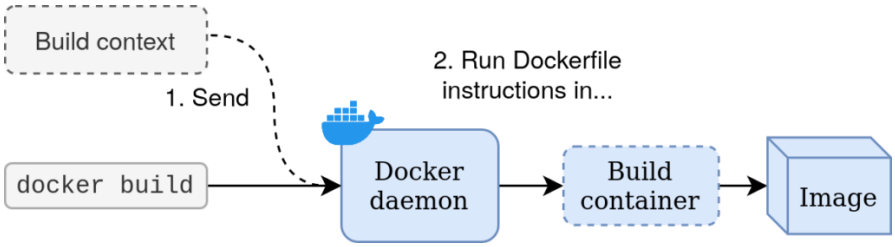
- Self-healing containers are containers configured to automatically recover from failures or unexpected terminations.
- Docker facilitates self-healing by using **Restart Policies**, which define the conditions under which Docker should automatically restart a container.
- Docker's restart policies allow you to specify under what circumstances a container should be restarted.
- Docker provides several restart policies that you can apply to a container:
- ***no (default):***
  - The container will not be restarted automatically, regardless of its exit status.
- **always:**
  - The container will always be restarted, regardless of the exit status.
  - If you manually stop the container, it will also be restarted unless Docker is stopped, or the restart policy is changed.
- **unless-stopped:**
  - The container will be restarted unless it was explicitly stopped by the user.
  - If the Docker daemon restarts, the container will be restarted only if it was running before the daemon stopped.
- **on-failure[:max-retries]:**
  - The container will be restarted only if it exits with a non-zero status code, indicating an error.
  - You can optionally specify a maximum number of retries.

```
docker container run --name <container-name> --restart always <image-name> <process>
```

Build Context vs Runtime Context

- Containers have two main phases:
  - Build Context: The environment and files available when building the image.
  - Runtime Context: The environment and resources available when running a container from the image.

Build Context



- The build context is the set of files and directories that are sent to the Docker daemon when building a Docker image using docker build.
- It provides the necessary files, dependencies, and configurations required to create the Docker image.
- The build context is specified when running the docker build command (e.g., `docker build .` where `.` is the build context).
- It typically includes the Dockerfile, application code, libraries, and any other files needed during the build process.
- The entire build context is sent to the Docker daemon, so it's important to keep it small and avoid including unnecessary files (e.g., use a `.dockerignore` file to exclude files).
- The build context is only relevant during the image creation phase and is not part of the final image unless explicitly copied using `COPY` or `ADD` in the Dockerfile.

Runtime Context

- The runtime context refers to the environment and resources available to a container when it is running.
- It defines how the container interacts with the host system, other containers, and external resources during execution.
- The runtime context includes:
  - Environment variables (ENV in Dockerfile or -e in docker run).
  - Volumes and bind mounts (-v in docker run).
  - Network settings (--network in docker run).
  - Resource limits (e.g., CPU, memory using `--cpus`, `--memory` in `docker run`).
  - Port mappings (-p in docker run).
- The runtime context is dynamic and can be modified when starting the container.
- It is independent of the build context and focuses on how the container operates after the image is built.

Feature	Build Context	Runtime Context
Phase	Image creation (build time)	Container execution (runtime)
Purpose	Provide files for building the image	Configure the container's runtime environment
Files	Dockerfile, application code, dependencies	Not directly related to files (except volumes)
Configuration	Defined in Dockerfile	Defined in docker run or Docker Compose
Scope	Limited to the build process	Applies to the running container

- **نقطة مهمة** هي أن الBuild Context بيكون عبارة عن الDirectory اللى فيه الdockerfile بس ومينفعش اطلع براه ، يعنى السطر اللى جاى دة يعمل Fail في الdockerfile.

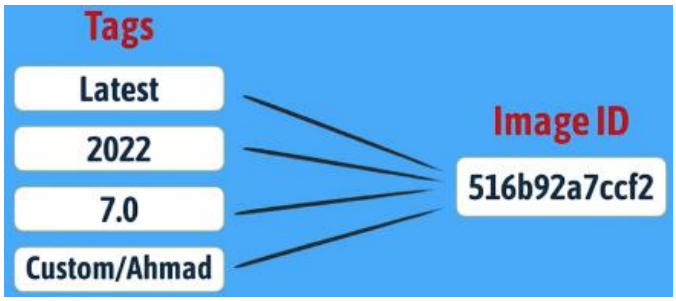
```
COPY ../secret-file /app/  # fail because "../secret-file" is outside the build context.
```

Docker Tags

- Meaningful and consistent image tags not only help users easily identify and select the appropriate image versions for their needs but also enhance clarity and streamlines workflows within the team.
- A tag is essentially a label assigned to a Docker image to help identify it. It typically consists of two components:
    - **Image name** (also known as repository name): This is the name of the image.
    - **Tag**: This is an optional identifier, commonly used to represent a specific version or variant of the image. If no tag is specified, Docker automatically assigns the latest tag to the image by default. It's important to remember that "latest" is just a tag, like any other tag. It doesn't carry any meaning.
- The Docker tag helps maintain the build version to push the image to the Docker Hub.
- The Docker Hub allows us to group images together based on name and tag.
- Multiple Docker tags can point to a particular image.
- Basically, As in Git, Docker tags are like a specific commit. Docker tags are just an alias for an image ID.

How to Tag a Docker Image?

- **Tag a Docker image during the build process:**
    - `docker build . -t my-image:1.0.0`
    - Above, the dot (.) after the build command indicates that the current directory (containing the Dockerfile) is the build context. We're using the -t flag to tag the Docker image with the name my-image and version 1.0.0.
- **Tag a Docker image after the build process:**
    - `docker image tag SOURCE_IMAGE[:TAG] TARGET_IMAGE[:TAG]`
        - docker image tag: This is the base command that tells Docker you want to tag an image.
        - SOURCE_IMAGE[:tag]: This is the name of the existing Docker image that you want to tag.
        - TARGET_IMAGE[:tag]: This is the name you want to apply to the image.



- **Tagging a Docker image for Docker Hub:**
    - `docker push <DOCKER_HUB_USERNAME/IMAGE_NAME[:tag]>`
- **Build an image with multiple Docker tags:**
    - `docker build . -t my-image:1.0.0-alpha -t my-image:1.0.0-dev`

Image Layers

- For instructions that change infrequently (e.g., installing system packages or dependencies), it's beneficial to place them near the top of the Dockerfile. This allows Docker to cache these layers, and subsequent builds can reuse them, saving time.
- As a rule, any Dockerfile instruction that modifies the file system creates a new layer.
- To view the commands that create the image layers and the sizes they contribute to the Docker image, execute the following command:
    - `docker history demo-app:v1`
    - Some commands or instructions create Intermediate layers which are 0B in size and don't add to the image size.

Squashing

- When Docker builds an image, it utilizes a layered filesystem (AUFS).
- Each command in the dockerfile adds a new layer that contains only the changes from the previous layer.
- This makes builds very fast since only changed files have to be copied.
- However, the layered filesystem also results in larger image sizes since each layer contains duplicate files.
- Squashing combines these layers into a single layer, reducing storage space and often improving runtime performance by decreasing mount points.

- يبقى الفكرة اللى هتساعد في حتة الimage layers انى يكون عندى أصلا cached layers موجودة وبالتالي تساعد في عملية الbuild لأى image جديدة لأن كدة بقى في جزء منها موجود عندى أصلا.
- لكن في نفس الوقت انا باخد الcached layer اللى عندى بافتراض ان الlayer اللى كانت قبلها محصلش فيها تغييريعنى تفضل unchanged.
- بمجرد ما يحصل تغيير في اى layer قبلها دة بـ invalidates the cache and results in recreation of subsequent layers from scratch.

- Docker image squashing is a process of combining multiple layers of a Docker image into a single layer or reducing the number of layers in the image.
- This is done to optimize the image by making it smaller and more efficient, as well as improving its performance during deployment.
- Here's why it matters:
    - **Layer reduction:** Each Docker image is built from multiple layers, and each layer represents a change or addition to the image (e.g., adding files, installing software, setting environment variables). Squashing reduces the total number of layers, which simplifies the image.
    - **Performance improvements:** Fewer layers can lead to faster image transfer and deployment since there are fewer parts to download and extract.
    - **Reduced redundancy:** Squashing helps eliminate redundant files or configurations across layers, making the image size smaller.
- When building docker image, we have some options available to us:
    - Leave it as-is.
    - Squash all the layers.
    - Squash layers down to the selected layer using the layer's ID.
    - Squash by specifying how many layers we want to squash.

How Squashing Works

- Docker uses a union file system to combine different layers of an image.
- Squashing merges these layers into a single unified layer, typically combining intermediate build steps, so the final image contains only what's needed for the application to run.
- You can use the `--squash` flag with `docker build` to squash layers. For example:

```
docker build --squash -t my-image:latest .
```

When to squash?

1. Temporary Files

- Sometimes you need to download some temporary files in one Docker layer just to remove them in a subsequent one.
- In such a case, they'll still contribute to your Docker image size, as the fact that you deleted it in a specific layer doesn't equal removing them from the previous ones.
- Once you merge these layers, only the diff from merged underlying instructions are preserved, and you can optimize your image size.
- This also refers to multiple layers modifying the same files - squashing will result in extracting only the delta of all merged layers.

2. Removing Sensitive Files

- During image creation, you may add temporary files or sensitive data (e.g., SSH keys, credentials, or temporary build artifacts) that are used in intermediate layers but are deleted in later steps.
- Even if they are deleted, they can still exist in previous layers.
- Squashing ensures that any sensitive or temporary data is fully removed from the image by merging the relevant layers.

3. Partial Squashing

- Imagine that you have a 190MiB container image for your app, but actually what changes from one release to another is mostly a few megabytes of JavaScript code.
- Squashing everything would mean downloading the whole 190MiB every time.
- Squashing nothing might mean a much bigger download, for example 990 MiB, not 190 MiB.
- The sweet spot is a partial squash where, usually, you get an update after downloading a few MiB, and everything feels fast.

4. CI/CD pipelines

- Preferably your images are built by CI/CD pipeline, where the Docker image cache starts from scratch.
- In such a case, you don't have to worry much about the caching behavior, and just optimize an image for the size.

5. Readable Dockerfiles

- As mentioned before, it's recommended to use as few layers as possible.
- A common technique is to concatenate RUN commands in a Dockerfile, cutting image size but also decreasing the readability.
- This may increase the barrier of entry for new hires significantly, and also negatively affect the developers' experience.
- Instead, you may still separate your Dockerfile instructions for the sake of simplicity, and squash layers after the image build to reduce the time taken for image pulls and container launches.

When not to squash?

1. Development Environments

- In development, having multiple layers can actually be an advantage because Docker can cache these layers and reuse them during incremental builds.
- Squashing eliminates some of Docker's ability to efficiently cache intermediate steps.
- For example, if you frequently rebuild only the last few layers of an image, you'll lose the caching benefits when squashing is applied.

2. Complexity and Loss of Transparency

- Squashing combines all changes into a single layer, which can make it harder to see which steps contribute to what part of the image.
- This transparency can be useful when debugging or optimizing specific layers.

3. Limited Support:

- While squashing is supported in Docker (using the --squash flag), it is not enabled by default and may not be available in all versions of Docker.
- This means it might not always be the most convenient option for every workflow.

Applying Docker Squashing in a Real-World DevOps Pipeline

- In DevOps, the goal is often to automate as much of the build, test, and deployment process as possible.
- Docker squashing can be integrated into your CI/CD pipeline to ensure optimized, production-ready images are built and deployed efficiently.
- Imagine you have a Jenkins pipeline that builds and deploys a Node.js application. Here's how you can integrate Docker image squashing into the build process:

Dockerfile (before squashing):

```
FROM node:16

# Install dependencies
RUN apt-get update && apt-get install -y build-essential

# Set working directory
WORKDIR /app

# Install app dependencies
COPY package.json .
RUN npm install

# Copy application source code
COPY . .

# Expose application port
EXPOSE 3000

# Start the application
CMD ["npm", "start"]
```

Jenkins Pipeline:

```
pipeline {
    agent any

    stages {
        stage('Build') {
            steps {
                script {
                    // Build the Docker image with squashing
                    sh 'docker build --squash -t my-node-app:latest .'
                }
            }
        }
        stage('Test') {
            steps {
                // Run unit tests (example)
                sh 'docker run my-node-app:latest npm test'
            }
        }
        stage('Push') {
            steps {
                // Push the squashed image to a Docker registry
                withCredentials([string(credentialsId: 'dockerhub-credentials', variable:
'DOCKER_PASSWORD')]) {
                    sh 'echo $DOCKER_PASSWORD | docker login -u "myusername" --password-stdin'
```

```
                    sh 'docker push myusername/my-node-app:latest'
                }
            }
        }
    }
}
```

Benefits of Squashing in DevOps Pipelines:

- Faster Deployment: Smaller images are quicker to push, pull, and deploy, making the CI/CD pipeline more efficient.
- Reduced Storage Costs: Smaller images reduce storage requirements in Docker registries and cloud environments.
- Security: By squashing intermediate layers, you ensure that sensitive data (e.g., secrets, SSH keys) or unnecessary files are completely removed from the final image.

Multi-Stage Build

- A multi-stage build in Docker is a technique used to optimize Docker images by allowing you to use multiple intermediate stages in a single Dockerfile.
- The primary goal of multi-stage builds is to reduce the final image size by separating the build environment from the runtime environment.
- This is especially useful for applications where the build dependencies are large but not needed during runtime.
- This approach is particularly useful when building complex applications (e.g., compiled languages like Go, Java, or C++).

Here's how it works:

- **First stage:** This stage is responsible for building the application. It includes all the tools, libraries, and dependencies required to compile the source code (e.g., compilers, build tools).
- **Subsequent stages:** These stages use the output from the first stage but only include the essential runtime dependencies, reducing the overall image size. Each stage can copy artifacts (like binaries) from a previous stage, excluding unnecessary build tools and libraries.

Example Dockerfile with a multi-stage build:

```
# Stage 1: Build the application
FROM golang:1.18-alpine AS builder
WORKDIR /app
COPY . .
RUN go build -o myapp .

# Stage 2: Create a minimal runtime image
FROM alpine:latest
WORKDIR /app
COPY --from=builder /app/myapp .
CMD ["./myapp"]
```

Example: Multi-Stage Builds for a Java Spring Boot Application

Dockerfile with Multi-Stage Build:

```
# Stage 1: Build the application
FROM maven:3.8.6-openjdk-11 AS builder
WORKDIR /app

# Copy the source code
COPY pom.xml .
COPY src ./src

# Build the application
RUN mvn clean package

# Stage 2: Create the final image
FROM openjdk:11-jre-slim
WORKDIR /app

# Copy only the JAR file from the builder stage
COPY --from=builder /app/target/my-spring-boot-app.jar ./my-spring-boot-app.jar

# Expose the application port
EXPOSE 8080

# Start the application
ENTRYPOINT ["java", "-jar", "my-spring-boot-app.jar"]
```

In this Dockerfile:

- The builder stage (FROM maven:3.8.6-openjdk-11 AS builder) compiles the Java application using Maven and produces a JAR file.
- The final stage (FROM openjdk:11-jre-slim) creates a lightweight image based on the Java runtime environment, and only the compiled JAR file from the builder stage is copied over.
- This results in a much smaller final image because none of the build dependencies (Maven, source files, etc.) are included.

Jenkins Pipeline for Multi-Stage Build:

```
pipeline {
    agent any

    stages {
        stage('Build') {
            steps {
                script {
                    // Build the Docker image with multi-stage builds
                    sh 'docker build -t my-spring-boot-app:latest .'
                }
            }
        }
        stage('Test') {
            steps {
                // Run tests inside the container (optional)
                sh 'docker run my-spring-boot-app:latest java -jar /app/my-spring-boot-app.jar
--spring.profiles.active=test'
            }
        }
        stage('Push') {
```

```
            steps {
                // Push the final image to Docker registry
                withCredentials([string(credentialsId: 'dockerhub-credentials', variable:
'DOCKER_PASSWORD')]) {
                    sh 'echo $DOCKER_PASSWORD | docker login -u "myusername" --password-stdin'
                    sh 'docker push myusername/my-spring-boot-app:latest'
                }
            }
        }
    }
}
```

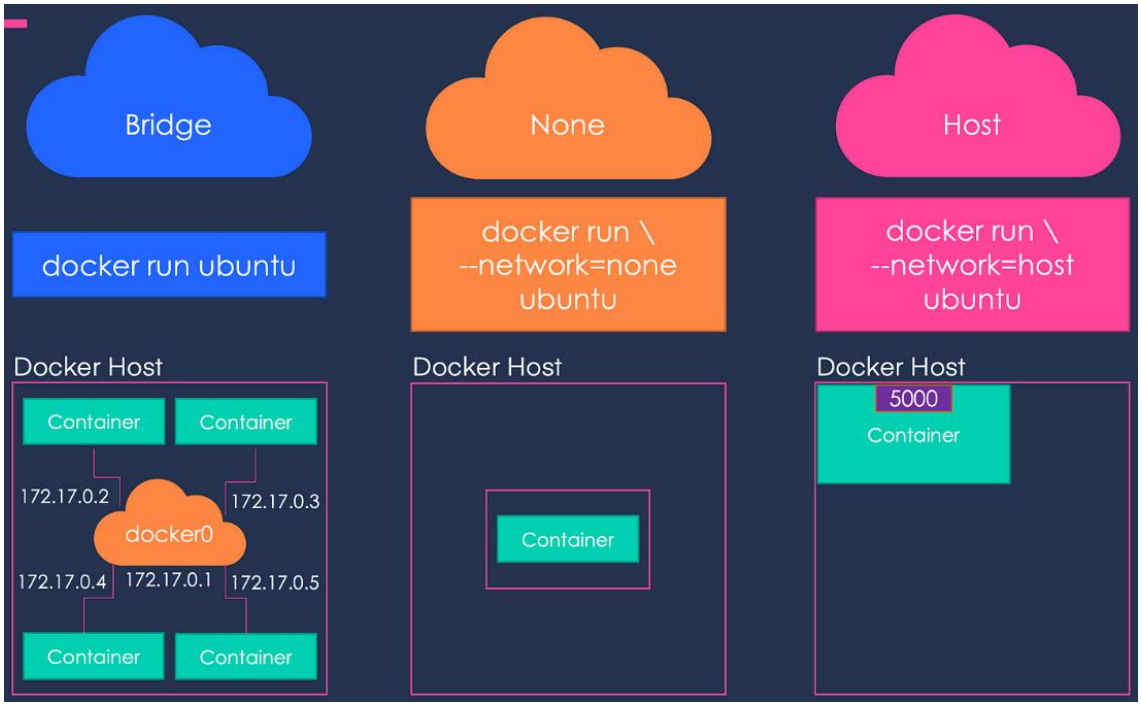
Aspect	Squashing	Multi-Stage Builds
Image Size	Reduces image size by merging layers	Reduces image size by copying only essential parts
Build Process	Simple, merges layers into one	Granular control over different build stages
Use Case	Suitable for final production images	Ideal for complex builds (compiled languages, large apps)
Caching	Loses some layer caching efficiency	Retains build caching across stages
Transparency	Makes the image harder to debug (hidden layers)	Easier to debug (separate stages, clear output)
Supported In	Docker CLI, but not widely adopted in CI/CD tools	Natively supported in Docker and widely used in CI/CD

When to Use Which?

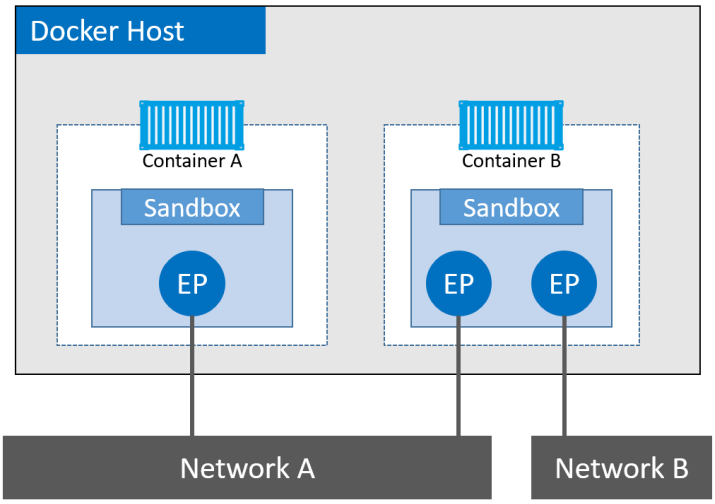
- **Use Squashing:** When you want to quickly reduce the number of layers in a relatively simple image and remove sensitive data or unnecessary files from intermediate layers.
- **Use Multi-Stage Builds:** When you have a complex build process, especially with compiled languages (e.g., Go, Java), and you want to ensure that only the necessary runtime and output files are included in the final image.

Docker Networking

- Docker networking is a fundamental part of Docker's functionality, enabling containers to communicate with each other, with the host machine, and with external networks.
- It provides isolation, security, and the ability to scale applications.
- Docker uses a **pluggable architecture** for networking, which means you can choose the best networking driver for your needs.
    - A pluggable architecture means that Docker's networking stack is modular and extensible.
    - Instead of being tied to a single networking model, Docker can use various networking drivers, each designed for different use cases.
    - These drivers can be built-in or added as plugins, allowing you to extend Docker's networking capabilities as needed.
- Docker provides five main types of networks:
    - Bridge Network (default)
    - Host Network
    - Overlay Network
    - Macvlan Network
    - None Network



- Docker container can be attached to multiple networks simultaneously.



Bridge Network (default)

- ***Description:***
  - The bridge network is Docker's default network.
  - When you create a container without specifying a network, Docker attaches it to a bridge network named bridge.
- ***Use Case:***
  - Ideal for containers running on a single Docker host that need to communicate with each other but remain isolated from external networks.
- ***Communication:***
  - Containers on the same bridge network can communicate with each other using their container names or IP addresses.
- ***Isolation:***
  - Containers on different bridge networks cannot communicate with each other by default.

```
docker run -d --name webapp --network bridge nginx
```

```
[ec2-user@ip-10-10-20-86 ~]$ ip link show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: enX0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 9001 qdisc fq_codel state UP mode DEFAULT group default qlen 1000
    link/ether 0a:1b:b6:e0:c8:3f brd ff:ff:ff:ff:ff:ff
    altname eni-0f7ae24e6702b6c8b
    altname device-number-0
3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN mode DEFAULT group default
    link/ether 02:42:60:a9:50:8b brd ff:ff:ff:ff:ff:ff
```

- لو جيت عملت كوماند ip link show على الmachine بتاعتى هلاقى ان docker عمل default interface كدة اللى هي bridge.

```
[ec2-user@ip-10-10-20-86 ~]$ ip addr show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host noprefixroute
       valid_lft forever preferred_lft forever
2: enX0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 9001 qdisc fq_codel state UP group default qlen 1000
    link/ether 0a:1b:b6:e0:c8:3f brd ff:ff:ff:ff:ff:ff
    altname eni-0f7ae24e6702b6c8b
    altname device-number-0
    inet 10.10.20.86/24 metric 512 brd 10.10.20.255 scope global dynamic enX0
       valid_lft 2941sec preferred_lft 2941sec
    inet6 fe80::81b:b6ff:fee0:c83f/64 scope link
       valid_lft forever preferred_lft forever
3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN group default
    link/ether 02:42:60:a9:50:8b brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
       valid_lft forever preferred_lft forever
```

- لو عملت ip addr show هلاقى تفاصيل اكتر عن كل interface فهلاقى مثلا ان الdefault bridge interface واخد 172.17.0.1/16 وبالتالي اى container هعمله من غير ما احدد له اى network هياخد علطول ip من الرينج دة.
- كل ما هاجى اعمل container جديد بقى واخليه يستعمل الbridge network هلاقى في virtual interface اتعمل له مخصوص على الlocal machine بتاعتى زى الصورة دى:

```
[ec2-user@ip-10-10-20-86 ~]$ docker container ls
CONTAINER ID   IMAGE     COMMAND     CREATED          STATUS          PORTS     NAMES
043cc22a29df   alpine    "/bin/sh"   5 seconds ago    Up 4 seconds              alpine1
82e3ebe68dc1   alpine    "/bin/sh"   9 seconds ago    Up 9 seconds              alpine2
deb28ccb15e0   alpine    "/bin/sh"   37 seconds ago   Up 36 seconds             alpine3
[ec2-user@ip-10-10-20-86 ~]$ ip link show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: enX0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 9001 qdisc fq_codel state UP mode DEFAULT group default qlen 1000
    link/ether 0a:1b:b6:e0:c8:3f brd ff:ff:ff:ff:ff:ff
    altname eni-0f7ae24e6702b6c8b
    altname device-number-0
3: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP mode DEFAULT group default
    link/ether 02:42:60:a9:50:8b brd ff:ff:ff:ff:ff:ff
11: vethb99cb69@if10: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker0 state UP mode DEFAULT group default
    link/ether 92:e6:92:60:4f:db brd ff:ff:ff:ff:ff:ff link-netnsid 0
13: vetha409624@if12: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker0 state UP mode DEFAULT group default
    link/ether 9a:09:6c:14:b3:19 brd ff:ff:ff:ff:ff:ff link-netnsid 1
15: veth4dd6cf9@if14: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker0 state UP mode DEFAULT group default
    link/ether 3a:50:ea:42:37:7b brd ff:ff:ff:ff:ff:ff link-netnsid 2
```

- ولو دخلت على الcontainer من جو هلاقيه واخد ip من الرينج:

```
/ # hostname
043cc22a29df
/ # ip addr show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
14: eth0@if15: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue state UP
    link/ether 02:42:ac:11:00:04 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.4/16 brd 172.17.255.255 scope global eth0
       valid lft forever preferred lft forever
```

Host Network

- ***Description:***
  - In the host network, the container shares the host's network stack, meaning it directly uses the host's IP address.
- ***Use Case:***
  - Useful when you need the container to have the same network performance as the host or when you need to avoid network overhead.
- ***Communication:***
  - Since the container uses the host's network stack, it can communicate with external networks using the host's IP address. However, it does not get its own IP address.
- ***Security:***
  - There is less isolation, as the container can affect the host's network configuration.

```
docker run -d --name webapp --network host nginx
```

- هنا بقى الcontainer بيبقى وكأنه جزء من الloacal machine وبيشوف كل الinterfaces اللى موجودة وكأنه local machine بالظبط:

```
[ec2-user@ip-10-10-20-86 ~]$ docker container run -it --network host --name alpine4 alpine
/ # hostname
ip-10-10-20-86.eu-west-2.compute.internal
/ # ip link show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: enX0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 9001 qdisc fq_codel state UP qlen 1000
    link/ether 0a:1b:b6:e0:c8:3f brd ff:ff:ff:ff:ff:ff
3: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP
    link/ether 02:42:60:a9:50:8b brd ff:ff:ff:ff:ff:ff
11: vethb99cb69@if10: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue master docker0 state UP
    link/ether 92:e6:92:60:4f:db brd ff:ff:ff:ff:ff:ff
13: vetha409624@if12: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue master docker0 state UP
    link/ether 9a:09:6c:14:b3:19 brd ff:ff:ff:ff:ff:ff
15: veth4dd6cf9@if14: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue master docker0 state UP
    link/ether 3a:50:ea:42:37:7b brd ff:ff:ff:ff:ff:ff
```

Overlay Network

- ***Description:***
  - The overlay network allows containers running on different Docker hosts to communicate as if they are on the same network.
  - This network is essential in a Docker Swarm or Kubernetes setup.
- ***Use Case:***
  - Ideal for multi-host Docker networks, enabling container communication across different hosts.
- ***Security:***
  - Provides secure communication between containers across hosts using an encrypted network.
- ***Scalability:***
  - Supports distributed, scalable applications by allowing communication between containers on different machines.

```
docker network create -d overlay my-overlay-network
```

Macvlan Network

- ***Description:***
  - The Macvlan network allows you to assign a MAC address to each container, making them appear as physical devices on your network.
  - Each container gets its own IP address on your physical network.
- ***Use Case:***
  - Useful for legacy applications that rely on MAC addresses or need to be directly accessible on the physical network.
- ***Isolation:***
  - Containers on a Macvlan network are isolated from the host unless the host is explicitly added to the network.
- ***Performance:***
  - Provides near-native network performance since it bypasses Docker's virtual network layer.

```
docker network create -d macvlan \
    --subnet=192.168.1.0/24 \
    --gateway=192.168.1.1 \
    -o parent=eth0 my-macvlan-network
```

This creates a Macvlan network with the specified subnet and gateway, attaching it to the eth0 interface.

None Network

- ***Description:***
  - The none network disables networking for the container.
  - The container has no access to any network interfaces other than a loopback interface.
- ***Use Case:***
  - Useful for highly isolated containers or when networking is managed externally.
- ***Isolation:***
  - Maximum network isolation, as the container cannot communicate with any other containers or external networks.

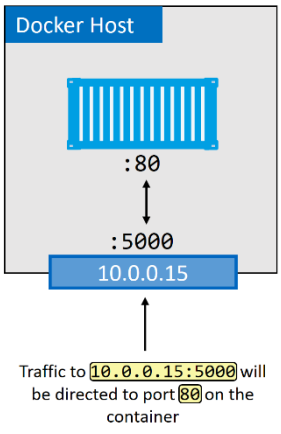
```
docker run -d --name isolated-container --network none nginx
```

```
[ec2-user@ip-10-10-20-86 ~]$ docker container run -it --network none --name alpine5 alpine
/ # hostname
fde5209c0032
/ # ip addr show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
```

Port Mapping

- Port mapping in Docker is the mechanism that allows a Docker container to be accessible from the host machine or other external networks by mapping a port on the host to a port on the container.
- This is particularly useful when you want to expose a service running inside a container (such as a web server or database) to the outside world or to other services running on the same host.

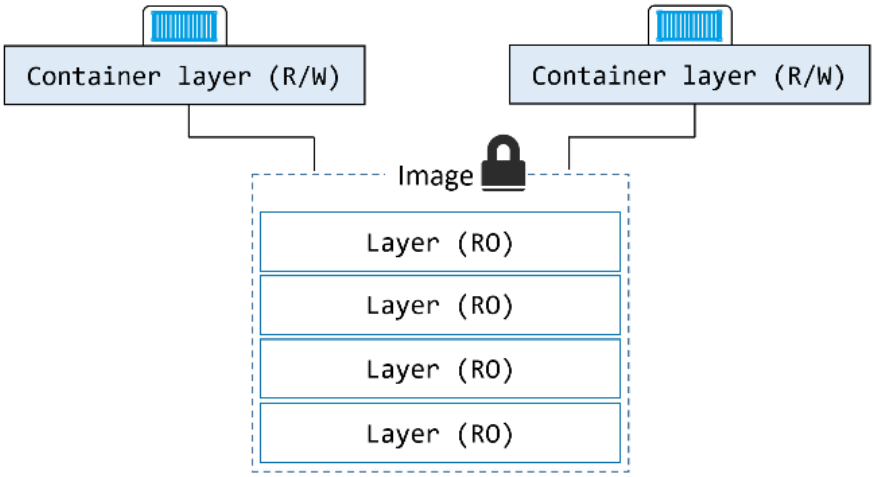
```
docker run -d -p <host_port>:<container_port> <image_name>

docker run -d -p 8080:80 -p 8443:443 nginx
```



Docker Storage

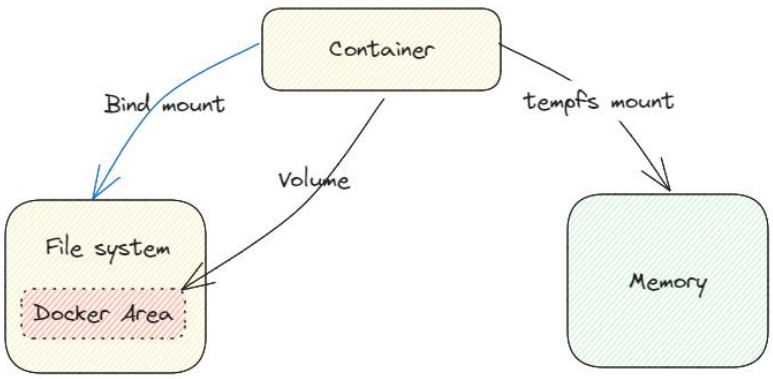
Containers and Non-Persistent Data

- Docker containers are designed to be ephemeral and immutable, meaning they can be created, destroyed, and recreated without affecting the base image or any persistent data.
- However, many applications require a read-write filesystem to operate. Docker addresses this by creating a thin read-write layer on top of the read-only filesystem provided by the container image.
- This read-write layer allows the container to perform write operations while keeping the underlying image immutable.
- The data in the writeable layer is temporary. It is stored on the Docker host and is tied to the lifecycle of the container. Once the container is deleted, the writeable layer and its data are also deleted.
- Each container has its own isolated writeable layer, meaning changes made in one container's writeable layer do not affect other containers or the underlying image.
- ***Location on Host:***
    - Linux Docker Hosts: `/var/lib/docker/<storage-driver>/....`
    - Windows Docker Hosts: `C:\ProgramData\Docker\windowsfilter\....`
- The storage driver is a critical component of Docker that manages how the writeable layer is created and maintained on the Docker host. The most common driver is *Overlay2*.



Containers and Persistent data

- Persistent data is data that remains available and unchanged even after the container is stopped or removed.
- Docker provides two primary ways to manage persistent data:
    - Bind Mounts
    - Volumes



Bind Mounts

- Bind mounts allow you to mount a specific directory from the host filesystem into a container.
- Bind mounts are directly tied to the host's filesystem and are not managed by Docker.
- ***Advantages:***
    - You can directly access and manipulate the files from the host, making bind mounts useful for development environments.
    - Any directory on the host can be used, offering more control over the location of the data.
- ***Disadvantages:***
    - Bind mounts are not as portable as volumes since they rely on the host's filesystem structure.
    - Bind mounts expose the host's directory structure to the container, which could lead to security issues if not carefully managed.
    - Non-Docker processes on the host or within Docker containers can modify these mounts at any time, which might lead to potential conflicts or inconsistencies.
- You specify a bind mount by providing the full path to the host directory when running a container:

```
Docker container run -d -v /host/path:/container/path nginx
```

Volumes

- Volumes are the most recommended and flexible way to persist data in Docker.
- Volumes are managed by Docker and are stored outside of the container's filesystem, typically in a location managed by Docker on the host machine.
- A single Docker volume can be attached to multiple containers simultaneously.
- Non-Docker processes cannot modify this part of the filesystem.
- ***Advantages:***
    - Volumes can be shared between containers, and they can be backed up, restored, and moved between Docker hosts.
    - Volumes are decoupled from the host filesystem, reducing the risk of accidental data exposure or conflicts.
    - Docker can use volume drivers to manage volumes across different storage backends, such as cloud storage or network file systems.
- ***Location on Host:***
    - On Linux: `/var/lib/docker/volumes/`
    - On Windows: `C:\ProgramData\Docker\volumes\`
- Creating a Volume:

```
docker volume create my-volume
```

- Using a Volume: When starting a container, you can mount a volume to a specific directory inside the container.

```
docker run -d -v my-volume:/var/lib/mysql mysql
```

tmpfs Mounts

- tmpfs mounts store data in the host's memory rather than on disk.
- This is useful for cases where you need fast, non-persistent storage that is cleared when the container stops.

```
docker run -d --tmpfs /app:rw,nodev,nosuid nginx
```

In this example, /app inside the container is mounted as a tmpfs filesystem, which exists in memory and will not persist after the container stops.

Containerizing Application

- You can create Docker images using two primary methods: from an existing container or with a Dockerfile.

Creating an Image from an Existing Container

This method involves creating a Docker image based on the current state of an existing container.

1- *Start a Container:*

First, start a container from a base image or an existing image.

```
docker run -it --name my-container ubuntu
```

2- *Make Changes:*

Make any changes to the container as needed. For example, you might install new software, modify configurations, or add files.

```
# Inside the container
apt-get update && apt-get install -y curl
# Exit the container
exit
```

3- *Commit the Container:*

Once you have made the necessary changes, commit the container to create a new image.

```
docker commit <container-id> <new-image-name>
```

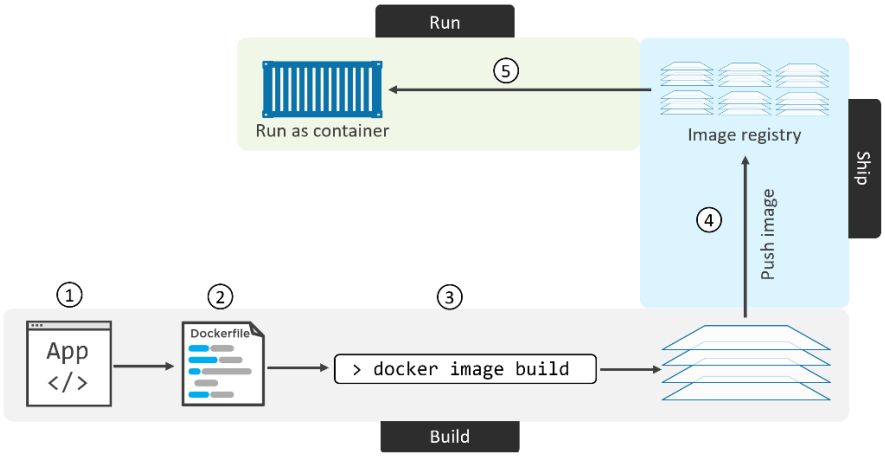
4- *Verify the New Image:*

You can check if the new image was created successfully by listing all images.

```
docker images
```

Creating an Image with a Dockerfile

- This method involves defining the instructions for creating an image in a Dockerfile, which is a text file with a series of commands.
- The process of containerizing an app looks like this:
    - Start with your application code and dependencies
    - Create a Dockerfile that describes your app, its dependencies, and how to run it
    - Feed the Dockerfile into the docker image build command
    - Push the new image to a registry (optional)
    - Run container from the image



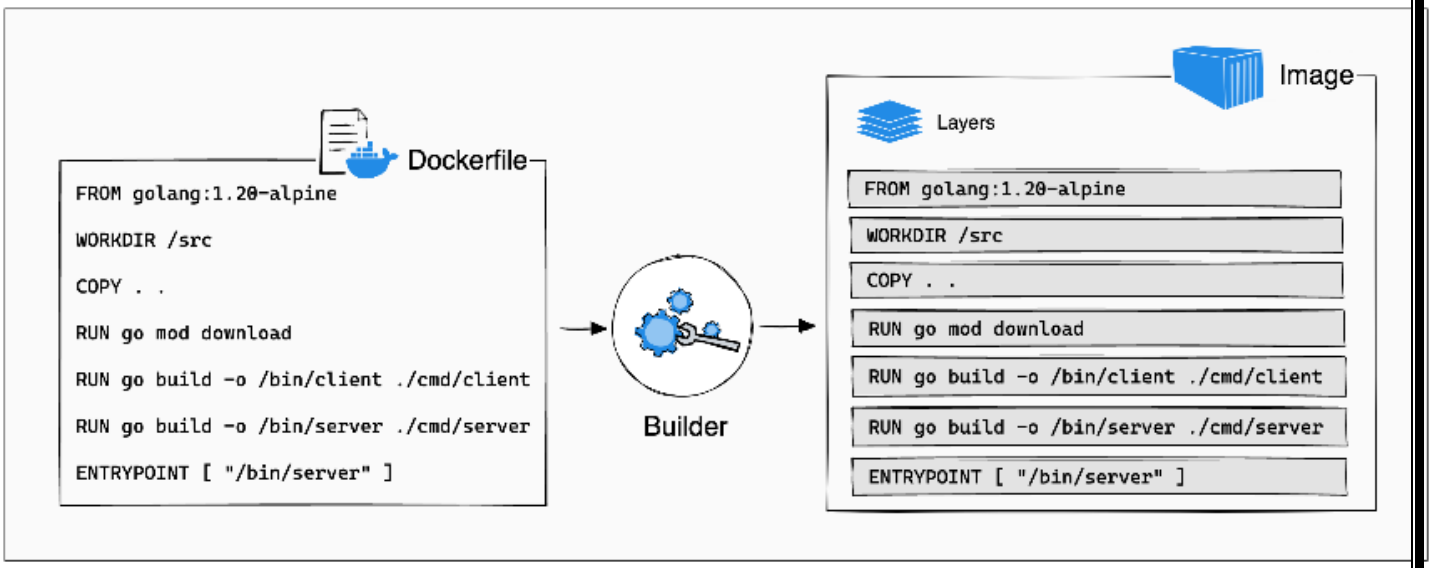
Dockerfile

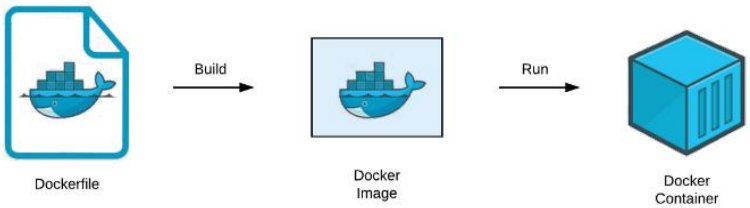
- A Dockerfile is a script composed of a series of instructions to build a Docker image.
- It defines how the image is created, including the base image, commands to run, files to include, and environment variables.
- Each Dockerfile instruction (FROM, RUN, COPY, ADD, etc.) generates a new layer. These layers are stacked on top of each other to build the final Docker image.
- Each layer is built on top of the previous one. This means that modifications in a later layer do not affect earlier layers, but they do build on them.
- Docker uses a caching mechanism to optimize the build process. If an instruction hasn't changed and its dependencies (including previous layers) haven't changed, Docker can use the cached version of that layer to speed up the build.
- Place instructions that are least likely to change earlier in the Dockerfile. This allows Docker to use cached layers for these instructions, speeding up builds.
- Combining commands where practical (e.g., using && in RUN commands) can help reduce the number of layers.
- The name of the Dockerfile can be changed from the default Dockerfile, but you'll need to specify the new name when building the image.

```
docker build -f MyDockerfile -t my-image:latest .
```

- **Example:**

```
FROM ubuntu:20.04             # Base image layer
RUN apt-get update            # New layer for updating package index
RUN apt-get install -y curl   # New layer for installing curl
COPY . /app                   # New layer for copying files into the image
CMD ["./app/start.sh"]        # New layer that defines the startup command
```





Shell Form vs Exec Form

- In a Dockerfile, there are two forms in which you can specify commands for the `RUN`, `CMD`, and `ENTRYPOINT` instructions: **Shell form** and **Exec form**.
- The key difference lies in how the command is executed inside the container.
- ***Shell Form***
    - Uses a single string.
    - Runs the command via `/bin/sh -c` (or `cmd.exe /S /C` on Windows).
    - The process running the command is the shell itself, not the command directly.
    - Environment variables and shell features (e.g., piping, redirection) can be used directly.
    - Syntax:

```
CMD command arg1 arg2
```

    - Example:

```
CMD echo "Hello, World!"
```

This is equivalent to:

```
/bin/sh -c "echo 'Hello, World!'"
```

    - Pros:
        - Easy to use and familiar for shell scripting.
        - Supports shell features like variable substitution and wildcards.
    - Cons:
        - Adds an extra layer (the shell process) to the container, which can increase overhead.
        - Signals (e.g., `SIGTERM`) are sent to the shell process, not the actual command, which can affect signal handling.

- ***Exec Form***
    - The command is written as a JSON array, where the first element is the executable and the subsequent elements are arguments. (`["executable", "arg1", "arg2"]`).
    - Executes the command directly without a shell.
    - The command replaces the container's PID 1 process.
    - Syntax:

```
ENTRYPOINT ["executable", "arg1", "arg2"]
```

    - Example:

```
CMD ["echo", "Hello, World!"]
```

This runs the echo command directly, without invoking `/bin/sh`.

- Pros:
    - More efficient because it avoids the overhead of a shell process.
    - Better signal handling, as signals are sent directly to the command.
- Cons:
    - Requires explicit handling of environment variables and shell features.
    - Less intuitive for complex commands that rely on shell features.

Feature	Shell Form CMD command arg1 arg2	Exec Form CMD ["command", "arg1", "arg2"]
Execution Method	Runs inside `/bin/sh -c` (shell)	Runs directly as an executable
Signal Handling	Poor (does not handle `SIGTERM` well)	Proper signal handling
Performance	Slightly slower due to shell overhead	Faster (no extra shell process)
Environment Variables	`$VAR` expands automatically	Requires explicit shell (`["sh", "-c", "echo $VAR"]`)
Complex Commands	Supports `&&`, `||`, `|`, and file globbing (`*`).	Does not support shell features directly. You must explicitly invoke a shell, e.g.
Override Behavior	Harder to override	Easier to override

Example Comparison

- Shell Form

```
CMD echo $HOME
```

This will print the value of the HOME environment variable using the shell.

- Exec Form

```
CMD ["sh", "-c", "echo $HOME"]
```

This achieves the same result as the shell form but explicitly uses the shell.

Popular Dockerfile Instructions

FROM:

- Represents the base image(OS), which is the command that is executed first before any other commands.
- **Syntax:**

```
FROM <ImageName>
FROM ubuntu:19.04
FROM ubuntu@sha256:2ca708c1c9d1e60373a000365bd78d8a0c13216927396fe99ee2e894d4a101d1
FROM myusername/myapp:1.0     # A custom application image stored on Docker Hub
FROM registry.example.com/mycompany/myimage:2.0 # Custom image from a private registry
```

WORKDIR:

- This instruction sets the working directory for any command that follows it in the Dockerfile.
- Any subsequent commands in the Dockerfile, such as `COPY`, `RUN`, or `CMD`, will be executed in this directory.
- WORKDIR instruction creates the directory if it does not exist.
- **Syntax:**

```
WORKDIR <DIRECTROY>
WORKDIR /app
```

COPY:

- The copy command is used to copy files or directories from the local file system (the Docker build context) into the image's filesystem.
- **Syntax:**

```
COPY <Source> <Destination>
COPY target/java-web-app.war  /usr/local/tomcat/webapps/java-web-app.war
COPY hello.py start.sh /app/
COPY ["first file.py", "Second File.sh", "/app"]
```

- If you want to copy all the files from the host's current directory to the container's current directory:

```
COPY . .
```

ADD:

- The ADD command does everything that `COPY` does, but with some additional features.
- If the `<source>` is a URL, `ADD` will download the file from the URL and place it in the specified <destination> directory in the image.
- If the `<source>` is a local archive file (e.g., .tar, .tar.gz, .zip), `ADD` will automatically extract the contents of the archive into the `<destination>`.
- **Syntax:**

```
ADD <source> <destination>
ADD https://example.com/file.tar.gz /app/
```

This will download file.tar.gz from the URL and place it in the /app/ directory.

```
ADD file.tar.gz /app/
```

This will extract the contents of file.tar.gz into the /app/ directory.

SHELL:

- Allows the default shell used for the shell form of commands to be overridden.
- The default shell used in a Docker container depends on the base image specified in the `FROM` command of your Dockerfile:
  - **Linux-based Images:** `/bin/sh`
  - **Windows-based Images:** `cmd.exe`
- **Syntax:**

```
SHELL ["executable", "parameters"]
SHELL ["/bin/bash", "-c"]
SHELL ["/usr/local/bin/python", "-c"]
```

RUN:

- Executes commands in a new layer on top of the current image and commits the results.
- Combine multiple RUN commands into a single one using `&&` to minimize the number of layers in your Docker image.
- When using apt-get or similar package managers, it’s good practice to clean up the package cache with apt-get clean to reduce the image size.

```
RUN ["echo", "This is exec form"]
RUN echo "This is shell form"
RUN apt-get update && apt-get install -y curl && apt-get clean
RUN echo "Hello, Docker!" > /app/welcome.txt
```

- ميزة مهمة هنا لل RUN command انه بيشتغل في الـbuild time ، فمثلا ممكن اعمل اى command يكون بيطلع output عشان يظهر في الـoutput بتاع docker build command ، دة ممكن يفيدنى في الـdebugging مثلا انى اتأكد خطوة اتنفذت صح او في مشكلة في الـbuilding الـimage في حتة معينة.

```
RUN java --version
```

LABEL:

- Adds metadata to an image in the form of key-value pairs.
- Labels are useful for adding descriptive metadata about the image, such as the author, version, or purpose.
- **Syntax:**

```
LABEL <key>=<value>
LABEL maintainer="mohaned@example.com"
```

EXPOSE:

- The EXPOSE instruction informs Docker that the container listens on the specified network ports at runtime.
- You can specify whether the port listens on TCP or UDP, and the default is TCP if you don't specify a protocol.
- EXPOSE does not actually publish the port; it only serves as documentation for users. Ports still need to be published using `docker run -p`.
- Syntax:

```
EXPOSE <PORT>
EXPOSE 80/udp
```

ARG (Build-time Variables):

- `ARG` is used to define variables that are available only during the build process of the Docker image.
- *Scope:*
  - Available only in the Dockerfile where they are defined.
  - Not accessible in the running container.
  - Can be overridden by passing a value at build time using the `--build-arg` flag with docker build.

```
docker build --build-arg APP_VERSION=2.0 -t myimage .
```

- *Syntax:*

```
ARG variable_name=default_value
```

- *Example:*

```
ARG APP_VERSION=1.0
RUN echo "Building version $APP_VERSION"
```

- *Usage:*
  - Pass build-time configuration (e.g., versions, paths, or flags).
  - Useful for customizing the build process without hardcoding values.

ENV (Environment Variables):

- `ENV` is used to define environment variables that are available both during the build process and in the running container.
- Scope:
  - Available in the Dockerfile during the build.
  - Persisted in the final image and accessible in the running container.
  - Can be overridden at runtime only using the `-e` flag with docker run.
- Syntax:

```
ENV variable_name=value
```

- Example:

```
ENV APP_HOME=/app
WORKDIR $APP_HOME
```

- Usage:
  - Set configuration for the application running in the container (e.g., database URLs, API keys).
  - Provide runtime configuration to the container.
- المفروض ان setting environment variables دة بيعادل export VAR=Value ، لكن انا مينفعش اعمل كدة لأن دة بيتعمل وقت الbuild time واللى بيحصل ساعتها ان docker بيعمل temp container ينفذ فيه الcommand بتاع RUN وبعدين ياخد منه الlayer وبعدين يمسحه وبالتالي الEnvironment variable هيتمسح اول ما الtemp container يتمسح ، لكن command الENV دة بيكون في الmetadata بتاعة الimage يعنى بيكون global على مستوى كل الlayers وكل الtemp containers.

Feature	ARG (Build-Time)	ENV (Runtime)
Scope	Available only during the build process	Available both during build and runtime
Persistence	Not available in the final image	Stored in the final image
Access in Running Container	No	Yes
Override During Build	`docker build --build-arg KEY=VALUE`	No
Override During Run	No	`docker run -e KEY=VALUE myimage`
Priority	Higher priority	Lower priority
Value Requirement	A value must be assigned in the Dockerfile	A value is optional in the Dockerfile

CMD:

- *Purpose:* Provides a default command and/or arguments for the container.
- *Behavior:*
  - The command specified by `CMD` can be overridden by providing arguments to `docker run`.
  - If `ENTRYPOINT` is also defined, `CMD` provides default arguments to `ENTRYPOINT`.
  - If no `ENTRYPOINT` is defined, `CMD` specifies the executable to run.
- *Syntax:*
  - Shell form: `CMD command param1 param2`
  - Exec form: `CMD ["executable", "param1", "param2"]`
- *Example:*

```
CMD ["echo", "Hello, World!"]
CMD echo "Hello, World!"
```

- *Use Case:*
  - Define default behavior for the container.
  - Provide default arguments that can be easily overridden at runtime.

ENTRYPOINT:

- *Purpose:* Defines the main command or executable that runs when the container starts.
- *Behavior:*
  - The command specified by `ENTRYPOINT` is not easily overridden (unless `--entrypoint` is used).
  - Arguments passed to `docker run` **are appended** to the `ENTRYPOINT` command.
  - If `CMD` is also defined, its arguments are passed to `ENTRYPOINT` as defaults.
  - If no `CMD` is defined, only `ENTRYPOINT` is executed.
- *Syntax:*
  - Shell form: `ENTRYPOINT command param1 param2`
  - Exec form: `ENTRYPOINT ["executable", "param1", "param2"]`
- *Example:*

```
ENTRYPOINT echo "Hello, World!"
ENTRYPOINT ["echo","Welcome to GFG"]
```

- *Use Case:*
  - Define the main executable for the container.
  - Ensure the container always runs a specific command, with optional arguments.

Feature	CMD	ENTRYPOINT
Purpose	Default command	Fixed command
Override Behavior	Can be overridden by `docker run <image> <command>`	Cannot be overridden (unless `--entrypoint` is used)
Flexibility	Meant for suggested commands	Meant for mandatory commands
Use Case	When you want users to have control over the command	When you want the container to behave like an executable

- You can use `ENTRYPOINT` and `CMD` together. In this case, `CMD` will provide default arguments to `ENTRYPOINT`.
- **Example:**

```
FROM ubuntu
ENTRYPOINT ["python3"]
CMD ["app.py"]
```

- This setup means that when you run the container without specifying a command, it will execute `python3 app.py`.
- You can override `app.py` by running `docker run myimage other_script.py`.
- If you want to override `ENTRYPOINT`, you must explicitly set it with `--entrypoint`:

```
docker run --entrypoint bash myimage
```

- This will run a Bash shell instead of python3.

- **Example:**

```
FROM ubuntu
CMD ["echo", "Hello, World!"]
```

- Running this container with `docker run myimage` will print Hello, World!

- **Example:**

```
FROM ubuntu
ENTRYPOINT ["echo", "Hello,"]
```

- Running this container with `docker run myimage "World!"` will print Hello, World!

- **Example:**

```
FROM ubuntu
ENTRYPOINT ["echo", "Hello,"]
CMD ["World!"]
```

- Running this container with `docker run myimage` will print Hello, World!
- Running this container with `docker run myimage "Docker!"` will print Hello, Docker!

.dockerignore

- The .dockerignore file is used to specify which files and directories should be excluded from the Docker build context.
- This is similar to a .gitignore file in Git, but for Docker.
- The .dockerignore file should be placed in the root of your **build context**, typically the same directory where your Dockerfile is located.
- When you build a Docker image, Docker sends the contents of your build context (the directory where the Dockerfile resides) to the Docker daemon.
- The .dockerignore file tells Docker which files and directories to exclude from this context, reducing the amount of data sent and potentially speeding up the build process.
- Each line in the .dockerignore file specifies a pattern that matches files or directories to be excluded.
- Exclude unnecessary files like documentation, .git directories, local development configuration files, test directories, and temporary files.
- **Example:**

```
# Ignore node_modules directory
node_modules/

# Ignore local environment files
.env
.env.local

# Ignore logs
logs/
*.log

# Ignore Docker-related files
Dockerfile
.dockerignore

# Ignore version control system directories
.git
.gitignore

# Ignore IDE/editor files
.vscode/
.idea/
*.sublime-project
*.sublime-workspace
```

Containerization Example

```
FROM alpine

LABEL maintainer="nigelpoulton@hotmail.com"

# Install Node and NPM
RUN apk add --update nodejs npm curl

# Copy app to /src
COPY . /src

WORKDIR /src

# Install dependencies
RUN  npm install

EXPOSE 8080

ENTRYPOINT ["node", "./app.js"]
```

Dockerfile Breakdown

```
FROM alpine
```

- This command creates the first layer of the Docker image, representing the entire Alpine Linux file system.
- This layer serves as the foundation for all subsequent layers.

```
LABEL maintainer="nigelpoulton@hotmail.com"
```

- Adds metadata to the image, specifying the maintainer's contact information.
- No layer is created for this command. Docker stores metadata separately, so no new layer is added.

```
RUN apk add --update nodejs npm curl
```

- The `RUN` instruction executes a command to install Node.js, NPM, and curl using the apk package manager (Alpine's package manager).
- The `--update` flag ensures the package list is refreshed before installation.
- This creates a new layer. The layer includes all the changes made by the command, such as downloaded packages and installed files.

```
COPY . /src
```

- The `COPY` instruction copies the application code from the current directory on the host machine to the `/src` directory in the container.
- This creates a new layer. The layer contains the application's source code, and it allows Docker to cache this layer independently, so if the source code changes, only this layer needs to be rebuilt.

```
WORKDIR /src
```

- Sets the working directory to `/src`, so any subsequent commands will be run from this directory.
- This command also creates a new layer. It doesn't add much data to the image, but Docker tracks it as a separate layer to maintain the image's history and caching mechanisms.

```
RUN  npm install
```

- The `RUN` command installs the Node.js application dependencies specified in the `package.json` file.
- A new layer has been created to include the installed Node.js modules and their dependencies.

- This layer can be cached by Docker, so if the package.json file hasn't changed, Docker won't need to reinstall the dependencies in future builds.

```
EXPOSE 8080
```

- Documents that the application will listen on port 8080 when the container runs.
- However, this does not actually publish the port; it just informs users of the image.
- No layer is created for this command. It’s a metadata instruction that doesn't alter the file system or create a new layer.

```
ENTRYPOINT ["node", "./app.js"]
```

- The ENTRYPOINT command specifies the default command to run when the container starts.
- In this case, it runs node ./app.js, which starts the Node.js application.
- No layer is created for this command. It defines the container's entry point but doesn't modify the file system.

- في المثال اللى فات دة لو جيت عملت inspect للimage بتاعة alpine والimage اللى انا عملتها من الDockerfile ، هلاقى ان الimage بتاعتى استخدمت مجموعة الlayers اللى موجودين في الbase image ، واللى في الحالة دى layer واحدة بس جوة alpine ، بعد كدة فضلت تضيف layers عليها بعدد التعديلات اللى حصلت بسبب commands الDockerfile.

```
"Layers": [
  "sha256:78561cef0761903dd2f7d09856150a6d4fb48967a8f113f3e33d79effbf59a07"
]
```

```
"Layers": [
  "sha256:78561cef0761903dd2f7d09856150a6d4fb48967a8f113f3e33d79effbf59a07",
  "sha256:7f6bf9dcf74b07b299d76b9296f95e634991e7fcd52a67242a2f5dbf9e34ca42",
  "sha256:0bb6be902733bb2221ef04cd8df911bdf94a4e3e1cd420affe0cd035e58f3e25",
  "sha256:5f70bf18a086007016e948b04aed3b82103a36bea41755b6cddfaf10ace3c6ef",
  "sha256:2b71281ef02da7b8874eeec4164f2cd62e47a867df9a614441032ebf5461d90d"
]
```

Commands

docker run	This command is composed of three subcommands: 1- docker image pull 2- docker container create 3- docker container start - To run a container in foreground: docker container run hello-world docker run -d redis:alpine هنا انا بعمل run لـcontainer بس في detached mode بدل من foreground docker container run --detach --publish 5555:80 --name n1 ngnix - الاوبشن p- بعمل expose او publish لبورت معين انا اختارته هنا 5555 وربطته بالبورت بتاع nginx اللى هو 80 - لو انا معملتش أصلا الخطوة دى كدة انا معملتش port binding وبالتالي الـcontainer بتاعى unreachable. - لو انا هعمل run لاكتر من container الـimages بتاعتهم بتستخدم نفس البورت ، لازم بقى البورت بتاع الـhost اللى هعمله binding مع الـcontainer يكون مختلف عشان يبقى كل container مميز عن التانى. docker container run -d --name mymariadb -v myvolume:/var/lib/sql -e MARIADB_ROOT_PASSWORD=1234 mariadb - الاوبشن v- عشان اعمل volume جديد اسمه myvolume واعمله attachment للـdestination بتاع الداتابيز اللى جوة الكونتينر اللى هو /var/lib/sql عشان الداتا بيز تقدر تكتب عليه واللى عرفت الـpath بتاعه عن طريق انى عملت inspect للـimage بتاعة الداتا بيز وعرفت هي بتكتب فين. - الاوبشن e- عشان أدى parameter أو environmental variable للـcontainer يديه للداتابيز وهى بتتعمل عشان يعمل باسورد للروت docker container run -d --name mynginx -p 5555:80 -v ${pwd}:/usr/share/nginx/html nginx هنا مثلا انا بعمل container من nginx وبستخدم طريقة Bind mounting ، فإستخدمت الاوبشن v- عشان اعمل bind mounting لفولدر انا اوريدى واقف فيه من الـCLI وجواه ملفات موقع وبربطه برضو بالـpath بتاع الويب سيرفر. وبما انى هنا عامل ربط بطريقة bind mounting فكدة الفولدر الحقيقى الموجود على الـhost ممكن مثلا اعدل في ملف الـindex باستخدام vscode عادى. docker container run -d --network=my-network ubuntu هنا انا بعمل attach لـ network driver معين غير الـdefault اللى هو الـbridge.
docker container	- To list all available sub commands: docker container - To list all running containers: docker container ls - To list all created containers (running and stopped): docker container ls -a - To create a new container: docker container create --name mycont redis:alpine - To start one or more stopped containers: docker container start mycont - To stop one or more running containers: docker container stop mycont - To remove one or more containers: docker container rm mycont -----> stopped container docker container rm -f mycont -----> running container

	- To list logs of container:
	`docker container logs mycont`
	
	- To display a live stream of container(s) resource usage statistics:
	`docker container stats mycont`
	
	- To execute a command in a running container:
	`docker container exec mycont ping -c 1 google.com`
	`docker container exec -it mycont bash`
	`-i, --interactive Keep STDIN open even if not attached`
	`-t, --tty Allocate a pseudo-TTY`
	
	- To copy files/folders between a container and the local filesystem:
	`docker container cp ./some_file CONTAINER:/work`
	`docker container cp CONTAINER:/var/logs/ /tmp/app_logs`

`docker image`	- To list all available sub commands:
	`docker image`
	
	- To list all available images:
	`docker images`
	
	- To download an image from a registry:
	`docker image pull mariadb:2.7`
	
	- To create a tag TARGET_IMAGE that refers to SOURCE_IMAGE:
	`docker tag SOURCE_IMAGE[:TAG] TARGET_IMAGE[:TAG]`
	
	- To upload an image to a registry:
	`docker image push mohanedahmed/myimage:v1`
	
	- To show the history of an image:
	`docker image history redis`
	
	- To build an image from a Dockerfile:
	`docker image build --tag myownimage /path/to/Dockerfile`
	
	- To remove an image:
	`docker image rm redis`
	لو جيت امسح اى image وكان معمول منها container حتى لو stopped مش هيرضى يمسحها وهيقولى:
	Error response from daemon: conflict: unable to delete d2c94e258dcb (must be forced) - image is being used by stopped container 0fce9fae13d2

`docker inspect`	- To list information about a docker object:
	`docker image inspect redis`
	`docker container inspect cont1`
	`docker network inspect newnetwork`

`docker network`	- To list available sub commands:
	`docker network`
	
	- To list all available networks:
	`docker network ls`
	
	- To create a custom docker network:
	`docker network create --driver bridge my-new-network`
	`docker network create --driver bridge --subnet "172.25.0.0/16 my-new-network`
	
	- To connect a running Docker Container to an existing Network:
	`docker network connect <network-name> <container-name or id>`
	
	- To remove a Container from the Network:
	`docker network disconnect <network-name> <container-name>`
	
	- To provide detailed information about a specific network:
	`docker network inspect <network_name>`

	- To delete all unused networks on a Docker host: `docker network prune`

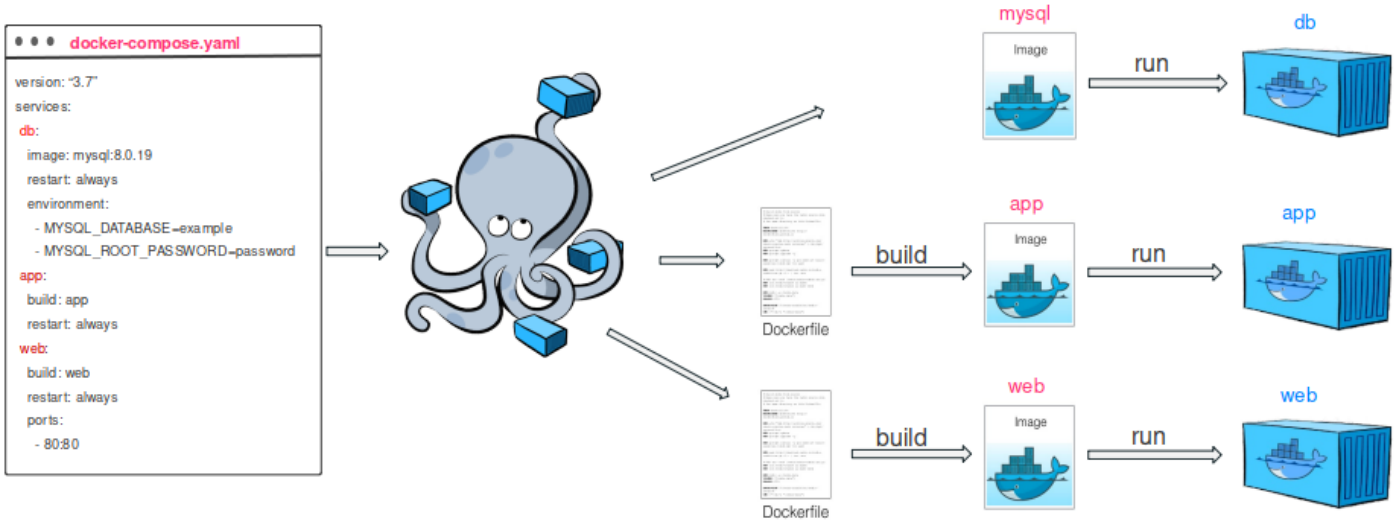
`docker volume`	- To list available sub commands:
	`docker volume`
	
	- To list all available volumes:
	`docker volume ls`
	
	- To create a new volume:
	`docker volume create <volume-name>`
	
	- To provide detailed information about a specific volume:
	`docker volume inspect <volume-name>`
	
	- To delete a specific volume that is not in use:
	`docker volume rm <volume-name>`
	
	- To delete all volumes that are not in use by a container or service replica:
	`docker volume prune`

`docker search`	Search for images in a Docker registry, such as Docker Hub:
	`docker search nigelpoulton`
	`docker search ubuntu --filter "is-official=true"`

`docker commit`	Create a new Docker image from the changes made to a container's filesystem: `docker commit mycontainer myimage:latest`
`docker export`	It is typically used for backup purposes or to migrate container states between environments. It exports the entire container's filesystem without any Docker-specific information, making it suitable for sharing or archiving. `docker export mycontainer > mycontainer.tar`
`docker import`	Create an image from a tarball file that contains a filesystem. This allows you to re-import a container's filesystem into Docker as an image. `docker import /path/to/mycontainer.tar myimage:latest`
`docker history`	Show the history of an image, detailing each layer that was added during the creation of the image. `docker history [OPTIONS] IMAGE`
`docker info`	To display information about Docker.
`docker login`	To login into docker hub account.
`docker logout`	To logout from docker hub account.

Docker Compose

- Docker Compose is a tool that simplifies the process of defining and running multi-container Docker applications.
- It allows you to configure application services in a single YAML file, called docker-compose.yml, and manage them as a group.
- With Docker Compose, you can start, stop, and configure multiple containers with just a few commands, making it easier to orchestrate complex applications.
- Docker Compose is part of the Docker ecosystem but is a separate tool from the Docker Engine itself.
- Docker Compose is written in Python which makes it platform-independent.



YAML Configuration File (docker-compose.yml):

- The docker-compose.yml file is where you define your application's services, networks, and volumes.
- The name of the Docker Compose file can be changed from the default docker-compose.yml, but you'll need to specify the custom name when running Docker Compose commands.

```
docker-compose -f MyComposeFile.yml up
```

- Each service is described with configuration options like the Docker image to use, ports to expose, environment variables, and dependencies on other services.
- **Example:**

```
version: "3.8"

services:
  web:
    image: nginx:latest
    ports:
      - "8080:80"
    depends_on:
      - db
    networks:
      - front-end

  db:
    image: mysql:5.7
    environment:
      MYSQL_ROOT_PASSWORD: example
    volumes:
```

```
      - db_data:/var/lib/mysql
    networks:
      - back-end

volumes:
  db_data:

networks:
  front-end:
  back-end:
```

Top Syntax Keys

(version – services – volumes – networks)

version

- The version key is used to specify the version of the Docker Compose file format that your configuration adheres to.
- This key is typically the first line in a docker-compose.yml file.
- **Syntax:**

```
version : "3.8"
```

Common Versions

- version: "3.8":
  - One of the most commonly used versions.
  - Introduced in Docker Compose 1.25.5.
  - Supports a wide range of features, including secrets, configs, and additional options for services.
- version: "3":
  - The 3.x series is the most widely used and is compatible with Docker Swarm.
  - This version introduced support for Docker Swarm mode, making it possible to define stacks for orchestration.
- version: "2.4":
  - Often used in legacy applications.
  - Supports features like depends_on, which is useful for defining service dependencies.
- version: "2":
  - An older format that predates the introduction of Docker Swarm mode.
  - Some features like networking are more basic compared to version 3.
- version: "1":
  - The original version, but not commonly used anymore.
  - Lacks support for many modern Docker Compose features.

services

- A service in Docker Compose represents a single container or a group of containers running the same image.
- Each service can be thought of as a microservice or a specific component of your application, like a web server, database, or caching layer.
- Services are isolated from each other, but they can communicate over a network, which Docker Compose sets up automatically.

Key Components of a Service

- `image`
  - Specifies the Docker image to use for the service.
  - **Example:**

```
web:
  image: nginx:latest
```

- `build`
  - Specifies the build context or Dockerfile for building the image.
  - **Example:**

```
web:
  build: .
```

- `command`
  - Overrides the default command defined in the Dockerfile.
  - **Example:**

```
web:
  command: ["npm", "start"]
```

- `ports`
  - Maps ports on the host to ports on the container.
  - **Example:**

```
web:
  ports:
    - "8080:80"
```

- `environment`
  - Defines environment variables for the service.
  - **Example:**

```
web:
  environment:
    - NODE_ENV=production
```

- `volumes`
  - Mounts host paths or named volumes into the container.
  - **Example:**

```
web:
  volumes:
    - ./data:/var/www/html
```

- `networks`
  - Connects the service to one or more Docker networks.
  - **Example:**

```
web:
  networks:
    - front-end
```

- `depends_on`
  - Specifies dependencies between services, ensuring one service starts before another.
  - **Example:**

```
web:
  depends_on:
    - db
```

Scaling the Service to Run Multiple Containers

- In Docker Compose, you can scale a service to run multiple containers by:
  - Specifying the scale option (in older versions).
  - Or using the `--scale` flag when you run the `docker-compose up` command.
- Suppose you have the following docker-compose.yml file:

```
version: "3.8"

services:
  web:
    image: nginx:latest
    ports:
      - "8080:80"
    networks:
      - front-end
```

- You can scale this service to run, for example, 3 instances (containers) using the following command:

```
$ docker-compose up --scale web=3 -d
```

- Docker Compose will start 3 containers for the web service, all running the `nginx:latest` image.
- These containers will be part of the same web service, but each will have a unique container ID.

networks

- Docker Compose allows you to define custom networks in which your services will operate.
- This feature enables communication between containers in the same network and isolates them from containers outside that network.
- Services connected to the same network can communicate with each other using their service names as hostnames.

Key Components of a Network

- `driver`
  - Specifies the network driver to use. The default network driver is bridge for single-host networking.
  - **Example:**

```
networks:
  front-end:
    driver: bridge
  back-end:
    driver: overlay
```

- `driver_opts`
  - Configure network-specific settings, such as bridge names or MTU sizes.
  - **Example:**

```
networks:
  front-end:
    driver: bridge
    driver_opts:
      com.docker.network.bridge.name: br0
      com.docker.network.driver.mtu: 1500
```

- `ipam`
  - Allows custom IP address management configurations.
  - **Example:**

```
web:
  command: ["npm", "start"]
```

- `ports`
  - Maps ports on the host to ports on the container.
  - *Components:*
    - **driver**: Specifies the IPAM driver.
    - **config**: Contains subnet, IP range, gateway, and auxiliary address.
  - **Example:**

```
networks:
  front-end:
    ipam:
      driver: default
      config:
        - subnet: 172.16.238.0/24
          gateway: 172.16.238.1
        - subnet: 172.16.239.0/24
```

- `external`
  - Specifies that a network should be created outside the scope of Docker Compose, often used for connecting to pre-existing networks.
  - **Example:**

```
networks:
  front-end:
    external: true
```

- `internal`
  - Configures the network to be internal, meaning it is isolated and containers cannot access external networks.
  - **Example:**

```
networks:
  front-end:
    internal: true
```

volumes

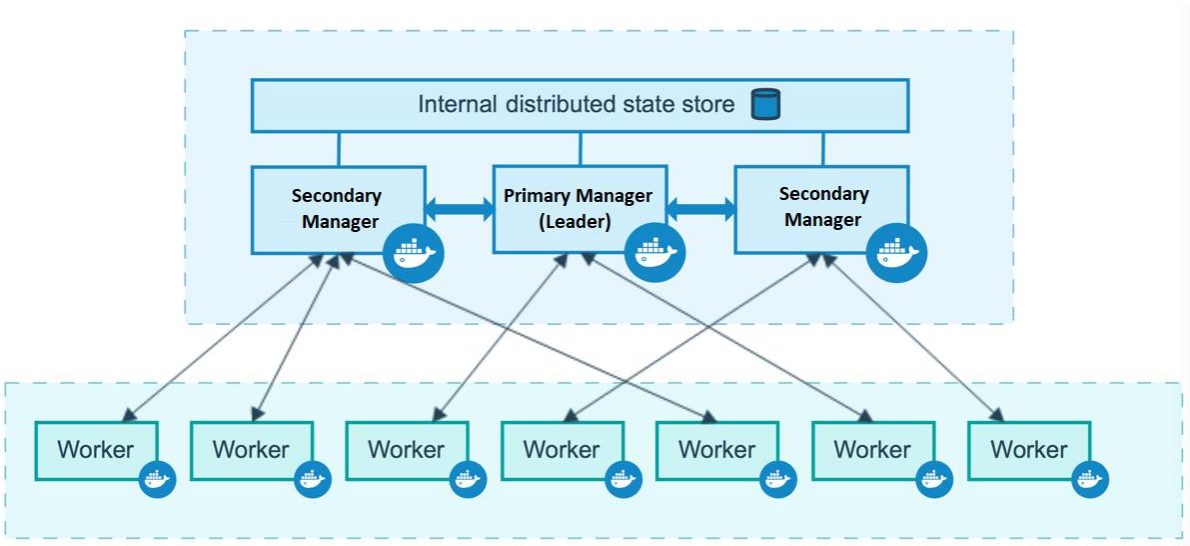
- You can define and manage data volumes in the `docker-compose.yml` file.
- Volumes allow you to persist data generated by and used by Docker containers, ensuring that data is not lost when the containers are stopped or removed.

Docker Compose Commands

`docker-compose`	`docker-compose up`
	This command creates and starts all the services defined in your docker-compose.yml file. If the required Docker images are not available locally, they will be pulled from a registry.
	• `-f` to specify the name and path of one or more Compose files.
	• `-p` to specify a project name.
	• `-d`, `--detach` Detached mode: Run containers in the background
	
	`docker-compose down`
	This command stops and removes all the containers, networks, and volumes created by docker-compose up. It's a clean-up command to bring down the environment.
	لما بعمل الكوماند دة هو بيمسح الcontainers والnetworks وبيسيب الimages والvolumes ، لأن الفكرة أصلا من الvolumes persistent انها متتمسحش لما الcontainer يقع او يتشال ، وبيسيب الimages عشان يبقى اسرع بعد كدة لما يجى يقوم نفس الـ environment تانى.
	
	`docker-compose ps`
	This command lists the status of the services defined in the docker-compose.yml file.
	
	`docker-compose exec web /bin/bash`
	This command allows you to run commands in a running container defined by the docker-compose.yml file.
	
	`docker-compose logs web`
	This command shows the logs for all services, or a specific service defined in the docker-compose.yml file.

Docker Swarm

- Docker Swarm is Docker's native clustering and orchestration tool.
- It turns a pool of Docker hosts into a single, virtual Docker host.
- With Docker Swarm, you can manage a cluster of Docker engines (referred to as a "swarm") and deploy services across multiple nodes for high availability, load balancing, and scalability.



Key Concepts in Docker Swarm

Node

- A node is an individual Docker Engine instance that participates in the Swarm.
    - **Manager Node:** Responsible for managing the Swarm, including maintaining the desired state, scheduling tasks, and managing the cluster.
    - **Worker Node:** Executes tasks that are assigned by the Manager Node.

Service

- A service in Docker Swarm is a higher-level abstraction for running and managing containers across a swarm of Docker nodes.
- It defines how the containers should be run, including the number of replicas, the image to use, the network settings, and other configurations.
- In Docker Swarm, services follow a declarative model.
    - You declare the desired state (e.g., "I want 3 instances of a NGINX container running"), and Docker Swarm works to ensure that state is achieved and maintained.
- **Global Services:** A global service runs exactly one task on every node in the swarm. This is useful for tasks that need to run on every machine, like monitoring agents.

Task

- A task is the smallest unit in Docker Swarm.
- It represents a single container running on a swarm node.
- The lifecycle of a task is managed by Docker Swarm.
- When you scale a service, Docker Swarm creates or removes tasks accordingly.
- If a task fails (e.g., due to a node failure), Docker Swarm automatically reschedules the task on another node to maintain the desired state.
- Once a task is created, it is immutable.
    - If a task needs to be updated (e.g., due to a change in the service definition), Docker Swarm creates a new task with the updated configuration and removes the old task.

Swarm Mode

- When Docker is in swarm mode, it can be managed using swarm-specific commands.
- You can initialize a swarm using `docker swarm init` and add nodes to it.

Overlay Network

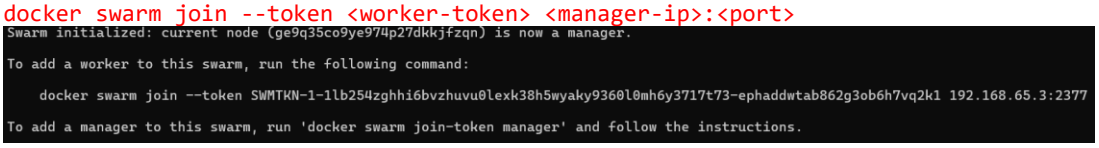
- A multi-host network that allows containers on different Docker hosts to communicate with each other securely.
- By default, Docker Swarm creates an overlay network for services to communicate across nodes.

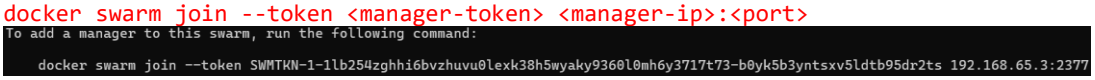
Ingress and Load Balancing

- Docker Swarm includes built-in load balancing.
- The swarm manager automatically assigns tasks to nodes based on available resources and balances incoming requests across available instances.

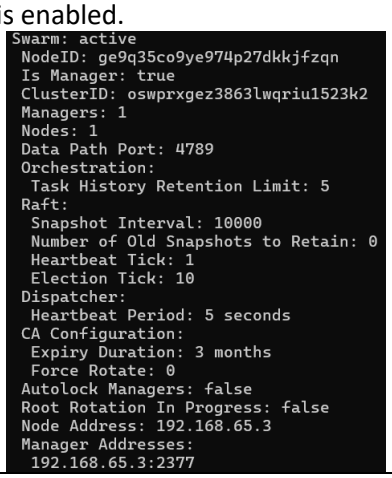
Docker Swarm Commands

<table>
<tr>
<td>docker swarm</td>
<td>Initialize the Swarm:
docker swarm init

After initialization, you'll get a command to add worker nodes:
docker swarm join --token <worker-token> <manager-ip>:<port>
 

Add manager node:
docker swarm join --token <manager-token> <manager-ip>:<port>
 

Retrieve the token that allows a new node to join the swarm as a worker node:
docker swarm join-token worker

Retrieve the token that allows a new node to join the swarm as a manager node:
docker swarm join-token manager</td>
</tr>
<tr>
<td>docker info</td>
<td>Detailed information about the Docker installation on your system, including details about the Docker Swarm mode if it is enabled.
 
</td>
</tr>
</table>

`docker service`	Deploy a service to the swarm:
	`docker service create --name web -p 80:80 --replicas 5 nginx:latest`
	
	List all services:
	`docker service ls`
	
	Inspect service tasks:
	`docker service ps web`
	
	Deploy a service to the swarm:
	`docker service rm web`
	
	Adjust the number of replicas:
	`docker service scale web=9`
	
	Update the service with new image:
	`docker service update --image nginx:latest --update-parallelism 2 --update-delay 5s web`

`docker node`	List nodes: `docker node ls`

ملاحظات:

- ممكن اروح اعمل بنفسى standalone container على node من الswarm عن طريق انى اروح عليها واعمله لكن هيبقى كدة مش متشاف من الswarm بشكل عام لأنه ساعتها مش service والswarm مش هتعمله management او تبدله مثلا لو وقع.

- لو انا مثلا نزلت nginx على nodes وجيت من node مش معمول عليها container وجيت بعت request هلاقى ان nginx بيرد عليا عادى لأن ساعتها network الoverlay عملت forward للrequest دة لnode عليها container ورجعتلى الresponse.

- بما ان الdocker swarm بيشتغل بمفهوم الservice ، فالservice دى حاجة stateful يعنى ليها مجموعة من الsettings تخليها توصل لdesired state ، بالتالى الهدف بتاع الdocker swarm طول الوقت انه يحافظ على الdesired state دى ، يعنى مثلا لو عملت docker service create --name myUbuntu --replicas 2 ubuntu:latest فكدة انا بقول لdocker swarm ان الdesired state بتاعتى ان يكون عندى 2 replicas ، لكن اللى بيحصل ان الcontainer الubunt بيعمل exit دايما لان مفيش service بتشتغل جواه عشان يفضل قايم عشانها ، بالتالى كل ما يتعمل container بيتقفل ويفضل docker swarm يعمل واحد مكانه وندخل في infinite loop بتنتهى لما docker swarm يدى failure.

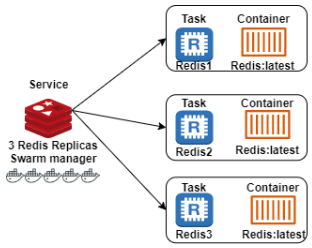
- طبعا الdesired state بيكون ليها settings تانية غير الreplicas زى مثلا الnetworks.

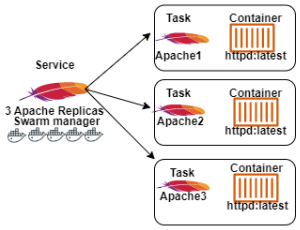
- عيب برضو من عيوب الdocker swarm انه لما يجى يبص للdesired state بتاع الcontainer هو بيشوف بس هو running ولا لا ، بمعنى مثلا لو الcontainer قايم بس من جواه حاجة ضاربة هو مش هيعمل حاجة وهيشوف الدنيا ماشية تمام.

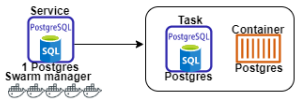
- مشاكل بقى تخص الdocker swarm انه مثلا الvolumes بتكون مخصوصة لكل node مش بينهم وبين بعض ، مثلا برضو الimage cache مش shared بين الnodes يعنى لو مثلا انا كنت مشغل container على node معينة وعاوز اشغله على node تانية فهو هيروح يسحب الimage تانى على الnode التانية دى الأول.

Docker Stack

- Docker Stack is a feature of Docker Swarm that allows you to deploy and manage a group of services that define an application.
- A stack is essentially a collection of services that are deployed together and can be managed as a single entity.
- Docker Stack uses a `docker-stack.yml` file to define the stack's services, networks, and volumes, making it easier to manage complex multi-service applications in a Swarm environment.







Docker Stack Commands

`docker stack`	Deploy a Stack:
	`docker stack deploy -c docker-stack.yml <stack_name>`
	
	List Stacks:
	`docker stack ls`
	
	List Services in a Stack:
	`docker stack services <stack_name>`
	
	Remove a Stack:
	`docker stack rm <stack_name>`

Practical Problems

Running docker commands with sudo privileges

- في بعض الـcommands في docker لازم تكون واخدة sudo privileges عشان تعرف تشتغل زى مثلا الـserver side info في الـcommand بتاع docker info.
- الـdocker لما بينزل اوتوماتيك بيعمل group اسمها docker وبيديها الـprivilegs دى فانا ممكن اضيف اليوزر بتاعى فيها وخلاص:

```
sudo usermod -aG docker <my-user>
```

About the Author

Connect with me on Linkedin: mohaned-ahmad

Explore more at: GitHub Repository