Docker

What is Docker?

- Open platform for **developing, shipping, and running applications**.
- Docker is a tool that allows you to package your application inside a **container**, and this container includes everything the application needs to run (e.g. Operating system , libraries , the code itself).
- Allows you to separate apps from infrastructure for faster delivery.
- Helps manage infrastructure like applications.
- Docker makes CI/CD (Continuous Integration and Continuous Deployment) easier. Consistently test and deploy your code to different environments e.g. Stage, UAT(User Acceptance Testing), Production.
- Docker uses a **build-cache** to accelerate the docker build process significantly. This is indicated by the CACHED message in the console output.
- Docker offers **two ways** to store data **outside the container:**
  - **Volumes**: make special location outside of container UFS (Union File System)
  - **Bind Mounts**: link container path to host path
- Differences between volume mounts and bind mounts

	Named volumes	Bind mounts
Host location	Docker chooses	You decide
Populates new volume with container contents	Yes	No
Supports Volume Drivers	Yes	No

1. Host location (موقع البيانات على الجهاز المضيف):

- **Named volumes**: يختار Docker مكان تخزين البيانات تلقائيًا داخل مجلد خاص به (عادة في `var/lib/docker/volumes/`).
- **Bind mounts**: أنت من يحدد المسار على جهازك الذي تريد ربطه بالحاوية (مثلاً: `home/user/data/`).

2. Populates new volume with container contents (هل يملأ الحجم الجديد بمحتوى الحاوية؟):

- **Named volumes**: نعم، عند إنشاء volume جديد وربطه بمسار داخل الحاوية يحتوي على بيانات، فإن Docker ينسخ هذه البيانات إلى الـ volume.
- **Bind mounts**: لا، لأنه يربط مجلدًا حقيقيًا من النظام مباشرة، ولا ينسخ أي شيء من الحاوية إليه.

🔹 3. Supports Volume Drivers – يدعم برامج تشغيل الـ Volume

- **Named volumes**: نعم، يمكنك استخدام برامج تشغيل مختلفة (volume drivers) لتخزين البيانات في أماكن متعددة (مثل التخزين السحابي).
- **Bind mounts**: لا، لا يدعم برامج التشغيل هذه.

What Can Docker Be Used For?

Fast & Consistent Delivery

- Standard environments for all stages (dev, test, production).
- Great for CI/CD pipelines.
- Example:
    - Devs write code and share via containers.
    - Test environments run automated/manual tests.
    - Fixes are redeployed easily.
    - Final image is pushed to production.

Responsive Deployment & Scaling

- Run containers on laptops, servers, clouds, etc.
- Scale apps up or down based on needs.

Better Hardware Utilization

- Lightweight alternative to virtual machines.
- Ideal for high-density or small/medium deployments.

Docker Architecture → Docker follows a client-server architecture.

- **Client-Server model**:
    - The Docker client communicates with the Docker daemon.
    - **Docker daemon** validation of the **Dockerfile** and returns an error if the syntax is incorrect and then runs the instructions in the Dockerfile.
    - The **daemon** handles tasks like building, running, and distributing containers.
    - The client and daemon can:
        - Run on the same machine, or Be separated (client on one machine, daemon on another).
    - They communicate over REST API (via socket or network).
- **Docker Compose** is another type of client → helps manage multi-container apps.
    - **Compose** works in all environments: production, staging, development, testing.
    - With Docker Compose, you can define all of your containers and their configurations in a single YAML file. (docker-compose.yml).

- **Steps to use it:**

1. **Define the app environment** in a Dockerfile
(so the image can be built anywhere).

2. **Define the services** in docker-compose.yml
   (like: web, database... each one runs in isolation).

3. **Run the app**: you can use two ways to **run Compose**
   docker compose up → built directly into Docker.
   docker-compose up → using a separate tool called the docker-compose **binary**.

This will start all the services together in a single, connected environment.

file compose.yaml → It defines all the services that make up your application, along with their configurations. Each service specifies its **image, ports, volumes, networks**, and any other settings necessary for its functionality.

Dockerfile versus Compose file

A **Dockerfile** provides instructions to build an **image** while a **Compose file** defines your running **containers**. Quite often, a Compose file references a Dockerfile to build an image to use for a particular service.

Docker Components

Docker Daemon (dockerd)

- Manages images, containers, networks, volumes.
- Can talk to other daemons.

Docker Client (docker)

- Main interface for users.
- Runs commands like docker run, docker build, etc.

Docker Desktop

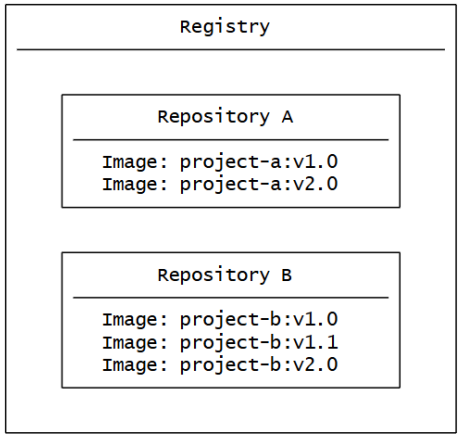
- Easy installation for Windows, Mac, Linux.
- Includes Docker Daemon, Client, Compose, Kubernetes, etc.

Docker Registry

- Stores Docker images.
- **Docker Hub** is the **default public** registry → Docker Hub is an online platform that provides ready images. You can pull these images directly and use them in your projects without rebuilding everything from scratch.
- Can use or create **private registries**. (Image Registry can be either public or private)
- Use docker pull, docker push to interact with registries.

Registry vs. repository

A **registry** is a centralized location that stores and manages images, whereas a **repository** is a collection of related images within a registry. Think of it as a folder where you organize your images based on projects. Each repository contains one or more images.



Docker Objects:

Images

- An image is a read-only template with instructions for creating a Docker container.
- An **image** is a standardized package that **includes all** of the files, binaries, libraries, and configurations to run a container.
- Official definition: "An Image is an ordered collection of root filesystem changes and the corresponding execution parameters for use within a container runtime."
- Can be based on other images (e.g., Ubuntu + Apache).
- Built using Dockerfile (each instruction = image layer).
- Only changed layers are rebuilt → lightweight and fast.
- There are Multi Images in the same server.
- Images are **immutable**. Once an image is created, it can't be modified. You can only make a new image or add changes on top of it.
- Images are composed of layers. And each of these layers, once created, are immutable.
- Each layer in an image contains a set of filesystem changes - additions, deletions, or modifications.

- Layers can be reused across different images, saving time, space, and bandwidth**.**

➢ **Layer Stacking (Union Filesystem)**

- Each layer is extracted into its own directory.
- When you run a container from an image, a union filesystem is created where layers are stacked on top of each other, creating a new and unified view.
- A separate directory is created for the container's changes, so the original layers stay untouched.

➢ **Images and their Layers (Explanation)**

- Each layer is uniquely identified and only stored once on a host
- This saves storage space on host and transfer time on push/pull
- A container is just a single read/write layer on top of image

---

Containers

- It contains everything that is in the Image.
- Runnable instances of images.
- Can start, stop, move, or delete them.
- Can connect to networks, use storage, etc.
- You can run multiple containers on the same host securely.
- Containers are **portable** (can be run on any OS) and can be shared easily.
- Isolated from host and other containers.
- Containers are **isolated** processes for each of your app's **components**, increasing the security of your applications.
- **Self-contained** → Each container has everything it needs to function with no reliance on any pre-installed dependencies on the host machine.
- **Independent** → Each container is independently managed. Deleting one container won't affect any others
- Containers are usually **immutable** and **ephemeral**
  - **ephemeral** → If any changes made inside a container aren't saved to persistent storage, such as a volume, they will be lost when the container is deleted or stopped.
  - **Immutable infrastructure** → only re-deploy containers, never change

- Isolation is achieved using two key technologies in Linux:
  - Namespaces:
    - Isolate things like process names, networks, files, etc.
    - Each container has its own copy of the environment.
  - Cgroups (Control Groups):
    - Control the resources the container can use:
    - CPU, RAM, network, etc.

Containers versus virtual machines (VMs)

a **VM** is an entire operating system with its own kernel, hardware drivers, programs, and applications. Spinning up a VM (Running a VM) only to isolate a single application is a lot of overhead.

A **container** is simply an isolated process with all of the files it needs to run. If you run multiple containers, they all share the same kernel, allowing you to run more applications on less infrastructure.

Using VMs and Containers Together (Using containers on VMs)

In cloud environments, virtual machines (VMs) are often used as the base infrastructure. **Instead of dedicating one VM to run one application**, a container runtime (like Docker) is installed on the VM to run multiple containers.

This setup combines the isolation and flexibility of VMs with the lightweight, fast, and efficient nature of containers.

Benefits → increasing resource utilization and Lower costs

- **Benefits of using Docker Images and Containers**

  - **Build Image**

Package everything your app needs (code + configs + tools) into one image.

  - **Ship Image**

Easily deliver the image to any environment (cloud or developer machine).

  - **Run Image**

Run the app the same way everywhere.

  - **CI/CD**

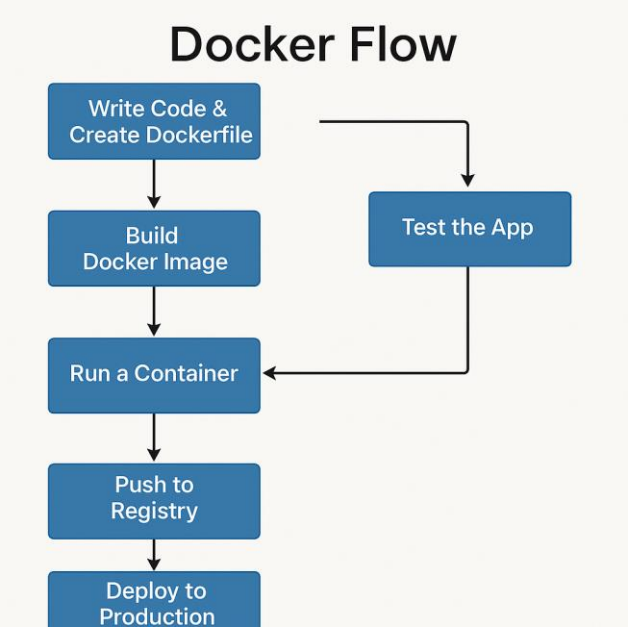
Automatically test and deploy code across environments (Staging, UAT, Production).

  - **Different Versions**

Run different versions of software without manual installation.

🔁 **Flow Summary in One Line:**

Code → Dockerfile → Image → Container → Test → Push → Deploy



Dockerfile

- Docker can build images automatically by reading the instructions from a **Dockerfile**.
- A **Dockerfile** is a text document that contains all the commands a user could call on the command line to assemble an image.
- Dockerfile located in the root of the context.

Dockerfile Format and Rules:

🔹 **General Format**

This is a comment

INSTRUCTION arguments

- Instructions are **case-insensitive**, but by convention, they are written in **UPPERCASE** to distinguish them from arguments.

🔹 **Instruction Order**

- Docker executes the instructions in a Dockerfile **in the order they appear**.
- A Dockerfile **must begin with a FROM instruction**.
- Only **one or more ARG** instructions are allowed **before** FROM.

🔹 **FROM Instruction**

- The FROM instruction sets the **base image** for the image being built.
- Each FROM starts a **new build stage**.

- A **valid Dockerfile** must contain **at least one FROM.**

Example:

ARG VERSION=20.04

FROM ubuntu:${VERSION}

- **Comments**
  - Any line that starts with # is treated as a **comment**.
  - A # appearing **mid-line** is treated as part of the instruction's **arguments**

Example:

This is a comment

RUN echo 'we are running some # of cool things' → **'#' here is not a comment**

- **ENV and Variables**
  - Environment variables (declared with the ENV statement) can also be used in certain instructions as variables to be interpreted by the Dockerfile.

Example:

FROM busybox

ENV FOO=/bar

WORKDIR ${FOO} # Becomes: WORKDIR /bar

ADD . $FOO # Becomes: ADD . /bar

Docker Networks

- Attach containers to more than one virtual network (or none)
- Skip virtual networks and use host IP :

  Instead of using a virtual network, you can make the container use the same network as the host machine by running:

  docker run --net=host ...

  This makes the container behave as if it is part of the host system itself, **without network isolation**

- **Default Security:**
  - Create your apps so frontend/backend sit on **same Docker network**
  - "Their inter-communication never leaves host" means that **communication** between containers on the same Docker network happens **internally** within the host machine, without passing through the internet .
  - **All externally** exposed **ports closed** by default
  - You must manually expose via -p, which is better default security!

- **DNS:**

  - Docker daemon has a built-in DNS server that containers use by default.
  - Docker defaults the hostname to the container's name, but you can also set aliases.
  - Containers shouldn't rely on IP's for inter-communication.
  - Docker has a built-in DNS system that works automatically when using custom networks, allowing containers to communicate using container names (friendly names) instead of IP addresses.
  - This gets way easier with Docker Compose.

Docker Volumes

- A **volume** in Docker is a way to store data outside the container, so it won't be deleted if the container stops or gets removed.
- Volumes are used to store **data persistently**, even when containers are stopped or deleted.
- **Bypasses Union File System** → Volumes do not use the Union File System used by Docker to group layers, which provides better performance and greater data stability.

- **Connect to none, one, or multiple containers at once**
  A volume can:
  - Be disconnected (not attached to any container)
  - Be attached to a single container
  - Or be shared and connected to multiple containers at the same time (allowing data sharing between them)
- Not subject to commit, save, or export commands
- By default, they only have a unique ID, but you can assign name

---

Command

- **Image Commands**

  docker build -f /path/to/a/Dockerfile .

  docker build -t my-image .

Command	Description
`-f`	Specifies the location of the Dockerfile
`-t`	Assigns a name (and optional tag) to the image
`.`	Sets the current directory as the build context

  docker rmi <Image ID> → Remove Image

  docker image ls → A command to list the images

  docker images → to list the newly created Docker image

docker search <Image name>

docker image history <Image name> →Show the history of an image

```
$ docker history nginx:latest
IMAGE          CREATED         CREATED BY                                      SIZE      COMMENT
abc123xyz456   2 weeks ago     /bin/sh -c #(nop)  CMD ["nginx" "-g" "daemon...   0B
def789uvw123   2 weeks ago     /bin/sh -c #(nop)  EXPOSE 80                    0B
ghi456rst789   2 weeks ago     /bin/sh -c #(nop)  COPY file:xyz123 in /etc/ngi...   5.6MB
...
```

docker image inspect → Display **detailed** information on one or more images, including:

- Config
- Entrypoint
- Env variables
- And most importantly: the defined Volumes

(e.g. docker image inspect my-image)

- **docker tag**
  - When we need to push into Docker Hub

    docker tag my-image YourUsername/my-image
  - When we need to push into AWS

    docker tag web-app
    <aws_account_id>.dkr.ecr.<region>.amazonaws.com/<repository_name>:tag

    (e.g. docker tag web-app 123456789012.dkr.ecr.us-east-1.amazonaws.com/my-web-app:latest)

docker push: Push an image or a repository **to a registry**

- When we need to push into Docker Hub

  docker push YOUR_DOCKER_USERNAME/my-image
- When we need to push into AWS

  docker push 123456789012.dkr.ecr.us-east-1.amazonaws.com/my-web-app:latest

---

- **Container Commands**

docker run -d -p 8080:80 nginx → **Creates and starts** a new container from an image
This maps port **80 inside the container** to port **8080 on your host machine**.

docker start nginx → Starts an **existing** (stopped) container

docker logs --tail 100 <Container Name or ID > → Print the last 100 lines of a container's logs

To run a MySQL database inside a container → docker container run -d --name mysql -e MYSQL_ALLOW_EMPTY_PASSWORD=True mysql

- -d: Run in the background.
- --name mysql: Assigns a name to the container.
- -e MYSQL_ALLOW_EMPTY_PASSWORD=True: Allows the root user to have no password (only safe for local/testing).
- mysql: The image name.

docker container port <container> →List port mappings for the container
docker container ls → This displays all currently running containers
docker container inspect mysql →This shows a full JSON description of the container, including:

- Mounts
- Network settings
- Ports
- Configuration

docker ps → The docker ps command will show you *only* running containers.
docker ps -a → to list all containers (Stopped and running)

docker stop <the-container-id>
docker rm <the-container-id>
docker rm -f <the-container-id> → stop and remove a container in a single command

- **Docker Hub Commands**

docker login →Log in to a Docker registry (Docker Hub)
docker logout → Log out from a Docker registry (Docker Hub)
docker pull mysql →This downloads the official MySQL image from Docker Hub to your local machine.

- **Network Commands**

docker network create <Network Name> → *Create a network*

docker network ls → Show networks
- --network bridge
- --network host
- -- network none

docker network inspect → Inspect a network

- **Volume Commands**

docker volume create <name> → Manually creates a volume

docker volume ls → This lists all the volumes currently created on your system.

docker run -dp 127.0.0.1:3000:3000 --mount type=volume,src=todo-db,target=/etc/todos getting-started → Runs Docker container with persistent data stored in a volume named todo-db, mounted to /etc/todos inside the container.

The following are examples of a named volume and a bind mount using **--mount:**

- Named volume: type=**volume**,src=my-volume,target=/usr/local/data
- Bind mount: type=**bind**,src=/path/to/data,target=/usr/local/data

---

- **Docker Compose Commands**

docker compose watch → Runs all required containers

docker compose up -d --build → Runs all services defined in docker-compose.yml

docker compose down → remove everything except volume (The idea is that you might want the data back if you start the stack again.)

docker compose down --volumes → remove the volumes (with everything)

- **Extra Commands**

<Command> --no-trunc → shows the full, unshortened output (e.g. docker ps --no-trunc)

docker exec -it <mysql-container-id> mysql -u root -p → Executes a command inside a container

الجزء	معناه
`docker exec`	يشغل أمر داخل **حاوية شغّالة** (يعني الحاوية لازم تكون already running).
`it-`	يسمح لك بالتفاعل مع التيرمنال (interactive terminal).
`<mysql-container-id>`	هذا هو **معرّف الحاوية** (container ID) **أو اسمها** اللي فيها MySQL.
`mysql`	الأمر اللي راح ينفذه داخل الحاوية، وهو هنا عميل MySQL (MySQL client).
`u root-`	يعني الدخول باستخدام المستخدم `root`.
`p-`	يعني "اطلب منّي كلمة المرور بعدين" (راح يطلبها بعد تشغيل الأمر مباشرة).

docker run -it --network todo-app nicolaka/netshoot

الجزء	المعنى
`docker run`	يشغل حاوية جديدة
`it-`	يشغل الحاوية بوضع تفاعلي (interact with terminal)
`network todo-app--`	يربط الحاوية بشبكة Docker اسمها `todo-app`
`nicolaka/netshoot`	اسم الصورة المستخدمة، وهي أداة مخصصة للتشخيص والتحقق من الشبكة (network debugging)

docker run -i -t ubuntu /bin/bash

- Pulls ubuntu image if not available.
- Creates a container and filesystem.
- Sets up networking.
- Starts an interactive terminal with Bash.
- Container stops when you type exit.

Instead of manually creating Docker files, you can create them automatically using the command:

docker init

- Automatically analyzes your project.
- Quickly generates:

  A Dockerfile

  A compose.yaml

  A .dockerignore

How to Use the NGINX Docker Official Image

- **What is NGINX?**
  - NGINX is a popular, fast, and reliable web server.
  - It's commonly used as a **reverse proxy** in front of APIs.
- **Running a Basic Web Server with NGINX:**
  - Run a container → *docker run -it --rm -d -p 8080:80 --name web nginx*

    --rm → Automatically delete the container after stopping it (Temporary container)

    Web → *Container name* / *Nginx* → *Image Name*

- Visit http://localhost:8080 in your browser — you'll see the default NGINX welcome page.

🔹 **Serving Custom HTML Files:**

- Create a folder (e.g., site-content) with an index.html file inside.
- Start a container with a bind mount:

  docker run -it --rm -d -p 8080:80 --name web -v ~/site-content:/usr/share/nginx/html nginx

  OR :

  *docker run -it --rm -d -p 8080:80 --name web *

  *--mount type=bind,**source**=$HOME/site-content,**target**=/usr/share/nginx/html *

  nginx

  source → The folder on your device

  target → The folder inside the container (the location of the NGINX files)

🔹 **Building a Custom NGINX Image:**

- Create a Dockerfile:

  FROM nginx:latest

  COPY ./index.html /usr/share/nginx/html/index.html

- هذا السطر **ينسخ ملف index.html من جهازك المحلي** ويتم نسخه إلى داخل ال image /usr/share/nginx/html/index.html هذا هو المكان الذي يبحث فيه NGINX عن صفحة البداية، وبالتالي أنت تقوم **باستبدال صفحة الترحيب الافتراضية** بصفحتك الخاصة.
- Build the image → docker build -t webserver .
- Run the custom image:

  docker run -it --rm -d -p 8080:80 --name web webserver

  --name is used to give the container a specific name instead of Docker choosing a random name.

🔹 **Pushing Your Custom Image to Docker Hub:**

- Login to Docker → docker login
- Tag and push the image:

  docker tag nginx-frontend yourusername/nginx-frontend

  docker push yourusername/nginx-frontend