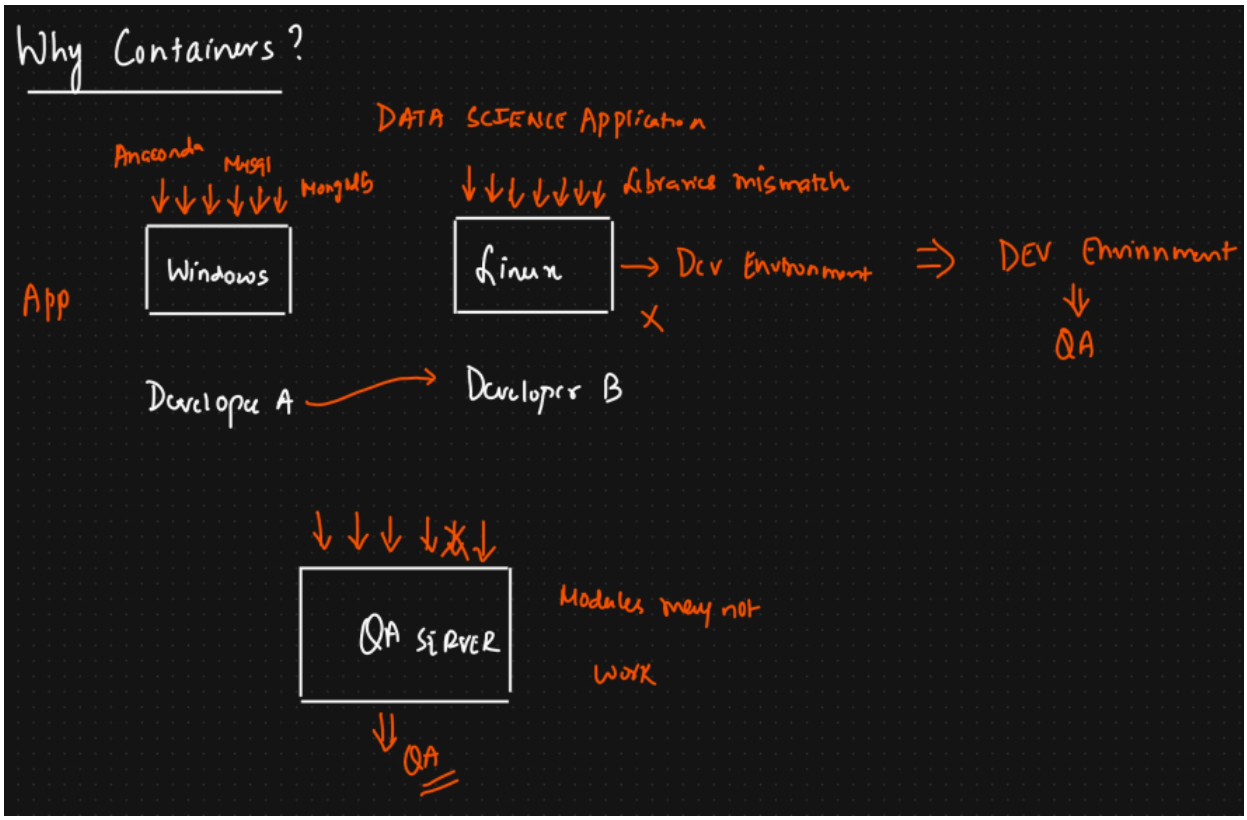
Docker

Docker is a **containerization platform** that allows developers to package applications with all their dependencies into standardized units called **containers**. It ensures the application works consistently across environments.

Problem Solved: It primarily addresses the "it works on my machine" problem by creating consistent environments from development through staging to production.

Why Use Docker? (Key Benefits)

- **Consistency:** Same environment everywhere (Dev, Test, Prod).
- **Portability:** Build once, run anywhere Docker is installed.
- **Efficiency:** Lightweight, fast startup, uses fewer resources than VMs.
- **Isolation:** Apps run separately, avoiding conflicts.
- **Scalability:** Easily create more containers to handle load.
- **Rapid Deployment:** Speeds up CI/CD pipelines (Build, Test, Deploy).
- **Resource Optimization:** Less CPU, RAM, and disk space needed compared to VMs.



Core Concepts

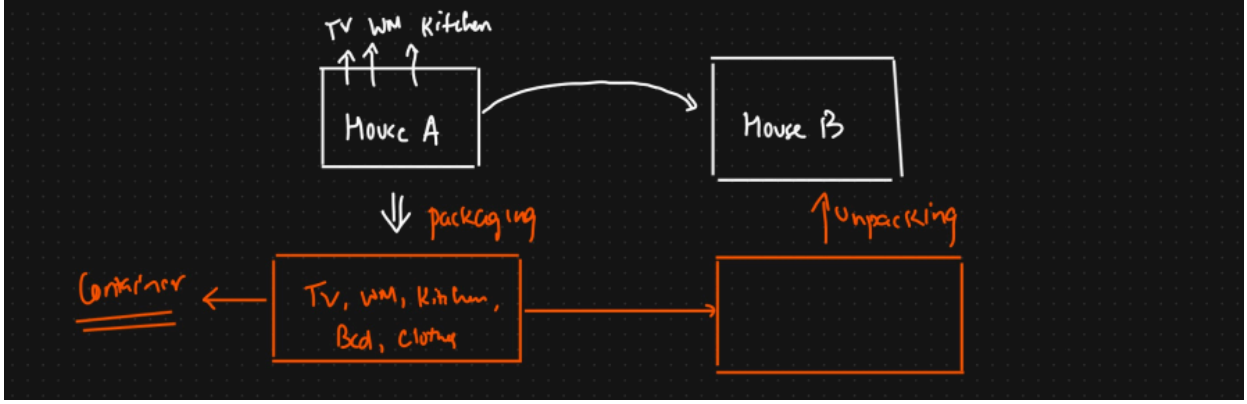
- **Containerization:** A lightweight form of virtualization where applications run in isolated user spaces called containers, sharing the host operating system's kernel.
- **Images:** Read-only templates used to create containers. Images contain the application code, libraries, dependencies, tools, and other files needed for an application to run. Images are often built in layers.
- **Containers:** Runnable instances of an image. You can create, start, stop, move, or delete containers using the Docker API or CLI. Each container is isolated but can communicate with others through well-defined networks.
- **Dockerfile:** A text file containing instructions (commands) for **building a Docker image** automatically. It specifies the base image, dependencies to install, files to copy, ports to expose, and the command to run when the container starts.
- **Registry:** A stateless, scalable server-side application that stores and distributes Docker images.
    - **Docker Hub:** The default public registry provided by Docker.
    - **Private Registries:** Can be hosted on-premises or using cloud provider services (e.g., AWS ECR, Google GCR, Azure ACR) for private image storage.
- **Volumes:** The preferred mechanism for persisting data generated by and used by Docker containers. Volumes are managed by Docker and are isolated from the host machine's core functionality. They are easier to back up or migrate than bind mounts.
- **Bind Mounts:** Allow mapping a directory or file on the host machine directly into a container. Useful for development when you want code changes on the host to reflect immediately inside the container. Less portable than volumes.
- **Networking:** Docker provides networking capabilities to connect containers together, connect them to the host machine, or connect them to external networks. Common network types include `bridge` (default), `host`, `overlay` (for swarm services), and `none`.

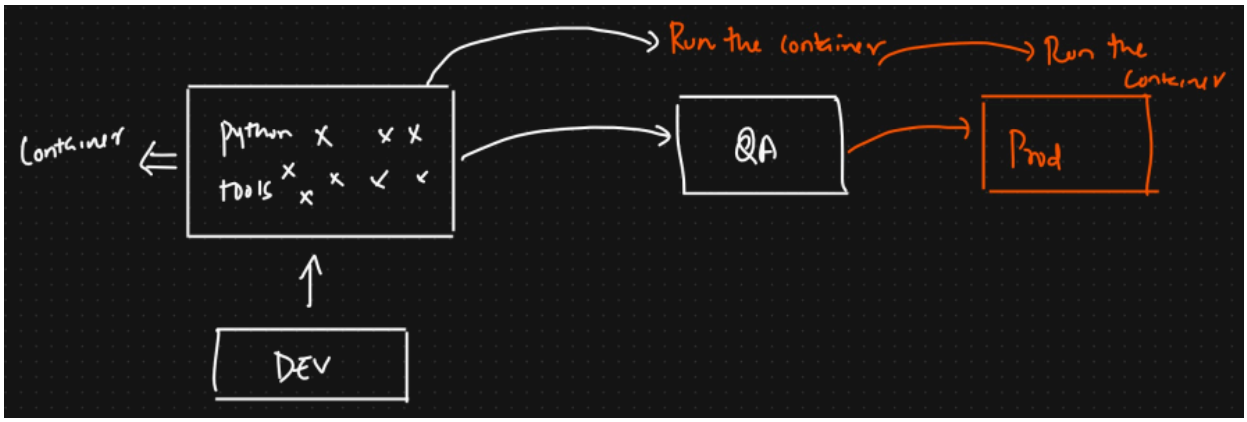
Containers

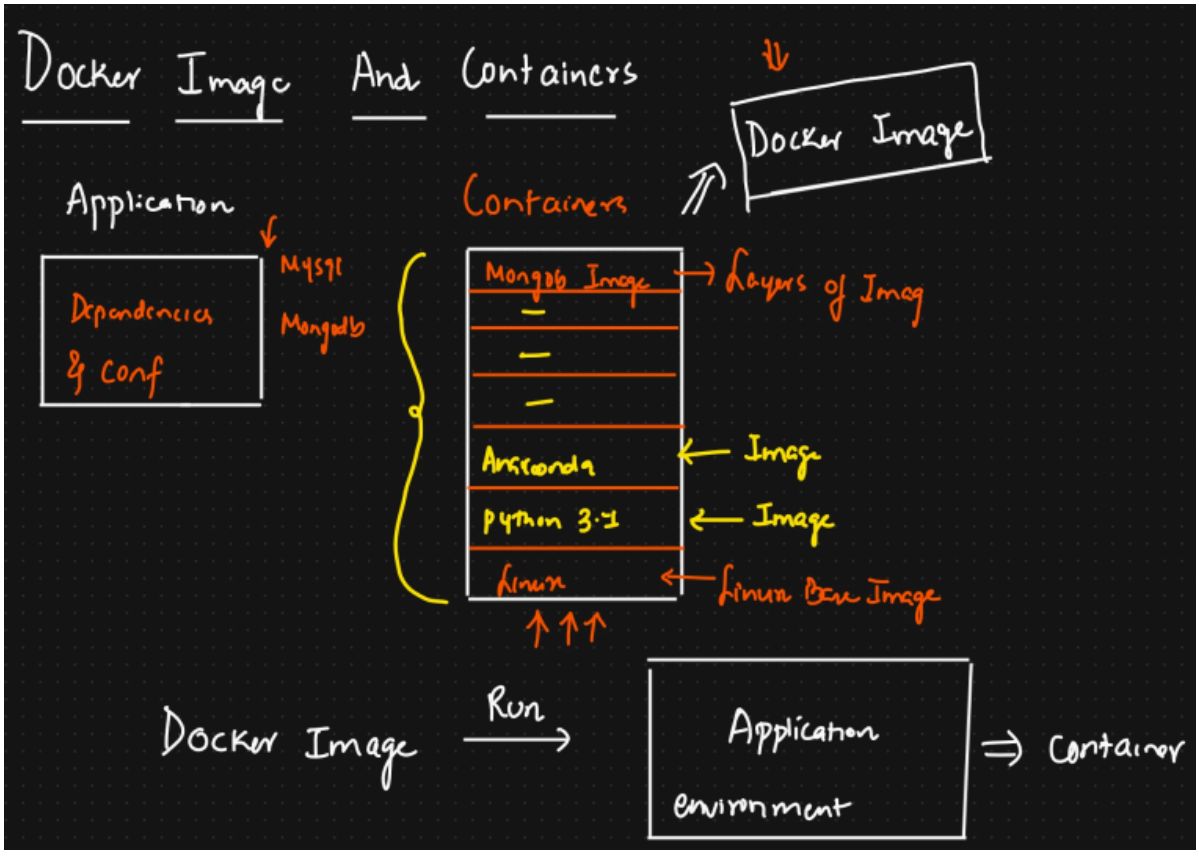
*) A way to package application with all the necessary dependencies And configuration

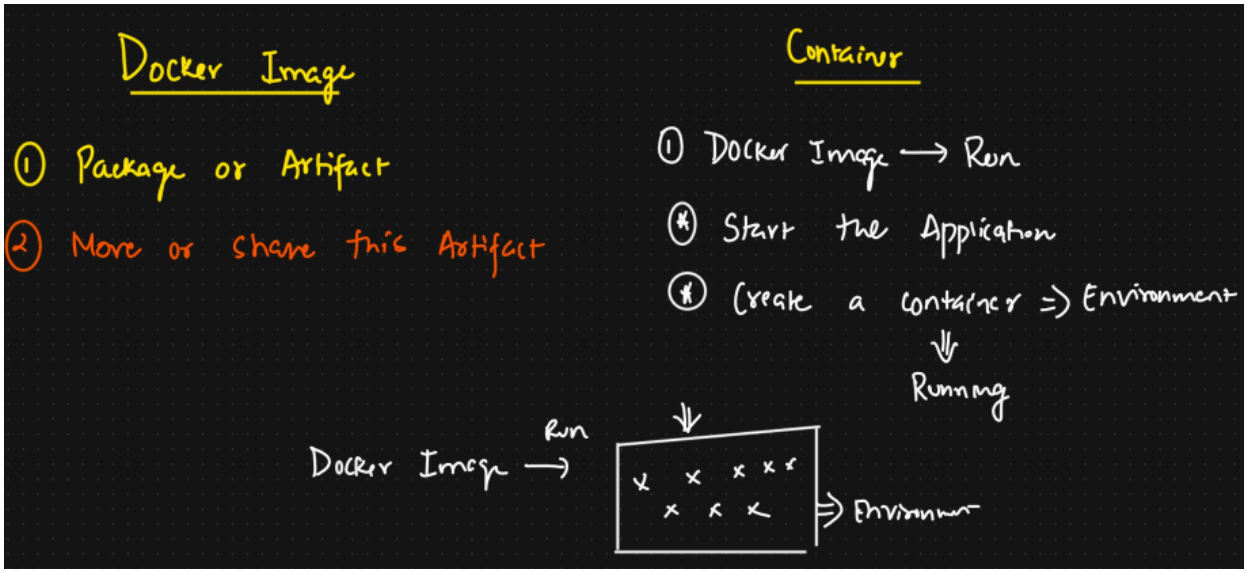
*) Portable artifact, easily share and move this package to any environment

*) Makes development and deployment more easy and efficient





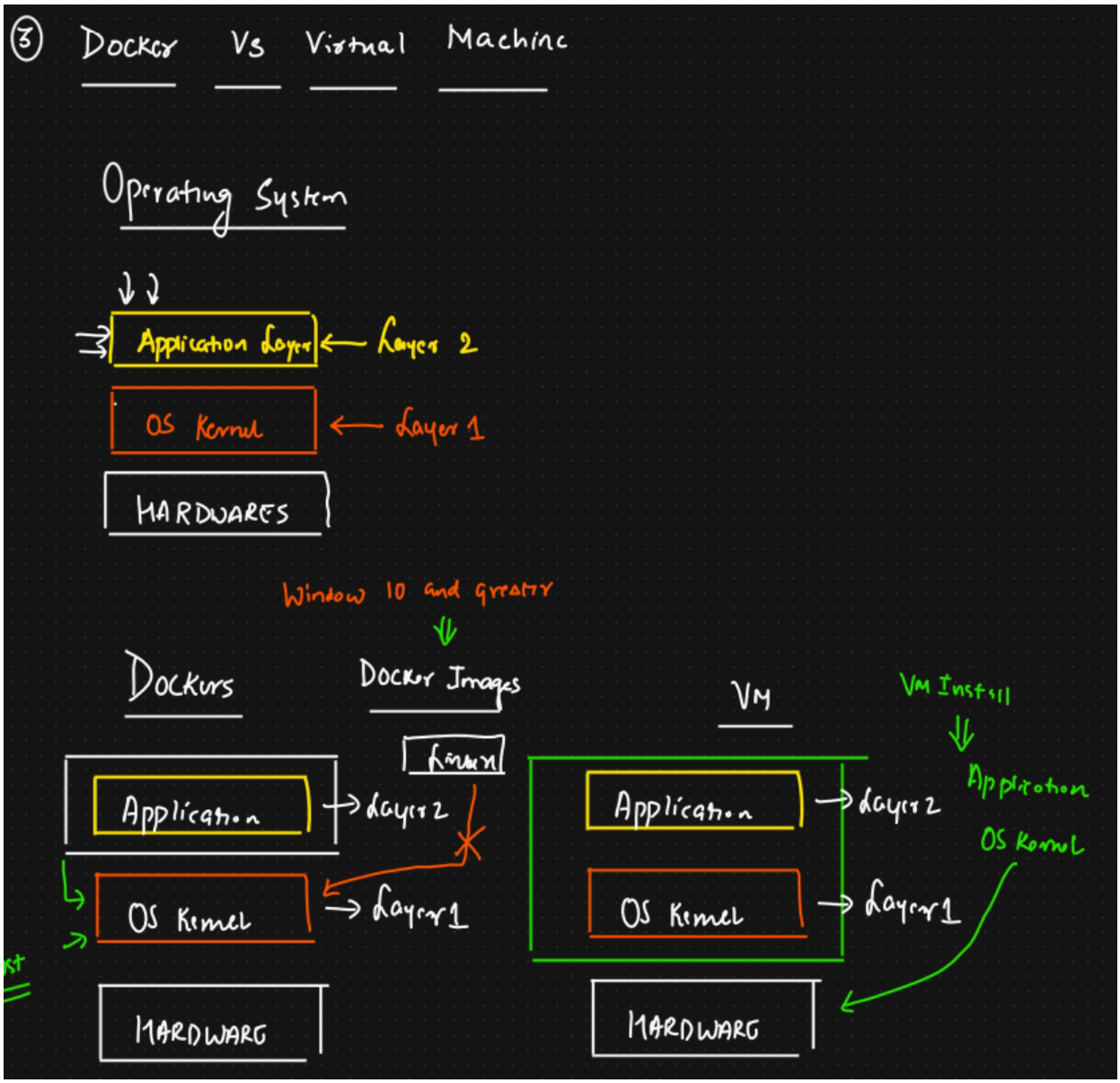


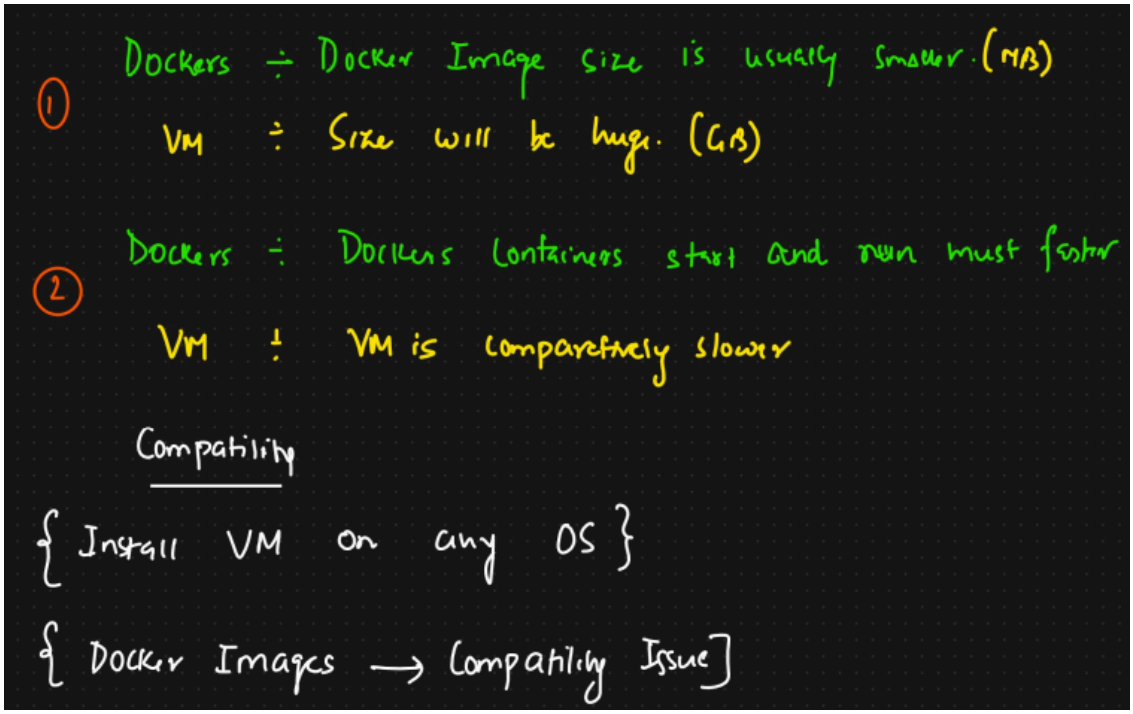


Containers vs. Virtual Machines (VMs)

Feature	Containers	Virtual Machines (VMs)
Isolation	Process-level isolation	Full OS isolation

OS	Share host OS kernel	Each VM has its own guest OS
Size	Lightweight (MBs)	Heavyweight (GBs)
Start-up Time	Fast (seconds)	Slow (minutes)
Resource Usage	Low (CPU, RAM)	High (CPU, RAM)
Performance	Near native	Slower due to hypervisor overhead
Density	High (many containers per host)	Low (few VMs per host)


③ Docker Vs Virtual Machine
Operating System
Application Layer
Layer 2
OS Kernal
Layer 1
HARDWARES
Window 10 and greater
Dockers
Docker Images
Application
Layer 2
OS Kernel
Layer 1
HARDWARE
VM
VM Install
Application
OS Kernel
Application
Layer 2
OS Kernel
Layer 1
HARDWARE



Docker Architecture (Client-Server Model)

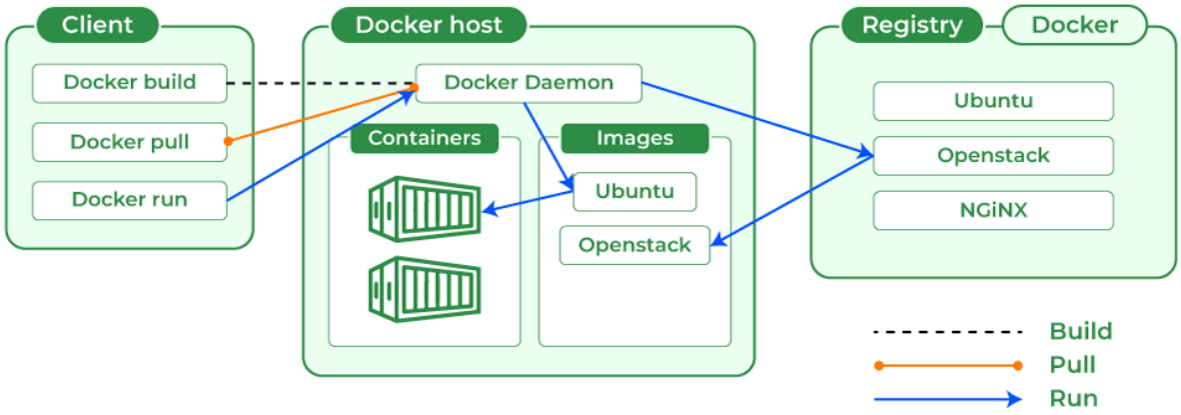
Docker uses a client-server architecture:

1. **Docker Client (`docker` CLI):** The primary user interface. You interact with Docker by using commands like `docker run`, `docker build`, etc. The client sends these commands to the Docker daemon.
2. **Docker Daemon (`dockerd`):** The Docker engine. A persistent background process that listens for Docker API requests and manages Docker objects like images, containers, networks, and volumes. It does the heavy lifting of building, running, and distributing containers.
3. **REST API:** The client communicates with the daemon using a REST API over a UNIX socket or a network interface. Other tools can also use this API to interact with Docker.
4. **Docker Registry:** Where Docker images are stored. When you run `docker pull` or `docker run`, the daemon pulls the required image from the configured registry. When you run `docker push`, the daemon pushes an image to the configured registry.

Key Components Deep Dive

- **Images & Layers:**
  - Images are built from a series of read-only layers.
  - Each instruction in a Dockerfile creates a new layer.
  - Layers are cached. If an instruction and its context haven't changed, Docker reuses the layer from the cache during subsequent builds, making builds faster.

    - When a container is created, a thin writable layer (the "container layer") is added on top of the image layers. All changes made inside the container (e.g., writing files) are stored in this layer.
- **Containers:**
    - Utilize Linux namespaces for isolation (e.g., `pid`, `net`, `mnt`, `uts`, `ipc`, `user`).
    - Use Control Groups (cgroups) for resource limiting (CPU, memory limits).
    - The container's filesystem is a union of the read-only image layers and the writable container layer.
- **Volumes:**
    - **Named Volumes:** Managed by Docker (`/var/lib/docker/volumes/` on Linux). Preferred way to persist data. Can be easily listed, inspected, removed, and shared between containers.
    - **Bind Mounts:** Map host filesystem path directly into the container. Tied to host path structure. Useful for development or accessing host configuration.
- **Networking:**
    - **Bridge (Default):** Creates a private internal network on the host. Containers on the same bridge network can communicate using container names/IPs. Docker sets up routing rules for external access via port mapping.
    - **Host:** Removes network isolation between the container and the host. The container shares the host's networking namespace.
    - **Overlay:** Used for multi-host networking, typically with Docker Swarm or Kubernetes. Allows containers running on different hosts to communicate securely.
    - **None:** Disables networking for the container.



Essential Docker Commands

(Format: `docker [OBJECT] [COMMAND] [OPTIONS] [ARGUMENTS]`)

Image Management:

- `docker pull <image_name>[:<tag>]`: Download an image from a registry (default: Docker Hub).
    - *Example:* `docker pull ubuntu:latest`
- `docker images` or `docker image ls`: List downloaded images.
- `docker build -t <image_name>[:<tag>] <path_to_dockerfile_directory>`: Build an image from a Dockerfile. Creating image from dockerfile.
    - `-t`: Tag the image (name:tag format).
    - `.`: Use the current directory as the build context.
    - *Example:* `docker build -t myapp:1.0 .`
- `docker rmi <image_id_or_name>...`: Remove one or more images.
    - `-f`: Force removal.
    - *Example:* `docker rmi myapp:1.0`
- `docker tag <source_image>[:<tag>] <target_image>[:<tag>]`: Tag an existing image.
    - *Example:* `docker tag myapp:1.0 myusername/myapp:latest`
- `docker push <image_name>[:<tag>]`: Upload an image to a registry.
    - *Example:* `docker push myusername/myapp:latest`
- `docker image inspect <image_id_or_name>`: Show detailed information about an image (layers, etc.).
- `docker history <image_id_or_name>`: Show the build history of an image (layers and commands).

Container Management:

- `docker run [OPTIONS] <image_name>[:<tag>] [COMMAND] [ARG...]`: Create and start a new container from an image.
    - `-d`: Run container in detached mode (in the background).
    - `-p <host_port>:<container_port>`: Publish a container's port(s) to the host.
    - `-v <host_path_or_volume_name>:<container_path>`: Mount a volume or bind mount.
    - `--name <container_name>`: Assign a name to the container.
    - `-it`: Run in interactive mode with a pseudo-TTY (often used for shells).
    - `--rm`: Automatically remove the container when it exits.
    - `--network <network_name>`: Connect the container to a specific network.
    - `-e <VAR_NAME>=<value>`: Set environment variables.
    - *Example (detached web server):* `docker run -d -p 8000:8000 --name aiengine-container aiengine:1.0`

    - If you have a .env file (e.g., config.env or .env) containing key-value pairs like:
    - Make sure the .env file is in the same directory you're running the command from, or provide a relative/full path.
    - `docker run -d -p 8000:8000 --name aiengine-container --env-file .env aiengine:1.0`
    - *Example (interactive shell):* `docker run -it --rm ubuntu:latest bash`
    - Docker Desktop provides a special DNS name — `host.docker.internal` — which allows containers to access services running on the **host machine**.
    - DB_HOST=host.docker.internal
- `docker ps`: List running containers.
    - `-a`: List all containers (running and stopped).
- `docker stop <container_id_or_name>...`: Stop one or more running containers.
- `docker start <container_id_or_name>...`: Start one or more stopped containers.
- `docker restart <container_id_or_name>...`: Restart one or more containers.
- `docker rm <container_id_or_name>...`: Remove one or more stopped containers.
    - `-f`: Force removal of a running container.
- `docker logs <container_id_or_name>`: Fetch the logs of a container.
    - `-f`: Follow log output.
- `docker exec [OPTIONS] <container_id_or_name> <command>`: Run a command inside a running container.
    - `-it`: For an interactive command (like a shell).
    - *Example:* `docker exec -it my-nginx bash`
- `docker inspect <container_id_or_name>`: Display detailed information about a container (IP address, volumes, etc.).
- `docker cp <src_path> <container>:<dest_path>` or `docker cp <container>:<src_path> <dest_path>`: Copy files/folders between a container and the local filesystem.

Volume Management:

- `docker volume create <volume_name>`: Create a named volume.
- `docker volume ls`: List volumes.
- `docker volume inspect <volume_name>`: Display detailed information on a volume.
- `docker volume rm <volume_name>...`: Remove one or more volumes (only if not used by any container).

- `docker volume prune`: Remove all unused local volumes.

Network Management:

- `docker network ls`: List networks.
- `docker network create <network_name>`: Create a custom bridge network.
- `docker network inspect <network_name>`: Display detailed information on a network.
- `docker network connect <network_name> <container_name>`: Connect a container to a network.
- `docker network disconnect <network_name> <container_name>`: Disconnect a container from a network.
- `docker network rm <network_name>...`: Remove one or more networks.
- `docker network prune`: Remove all unused networks.

System & Cleanup:

- `docker info`: Display system-wide information.
- `docker version`: Show the Docker version information.
- `docker system df`: Show Docker disk usage.
- `docker system prune`: Remove unused data (stopped containers, unused networks, dangling images, build cache).
    - `-a`: Remove all unused images (not just dangling ones).
    - `--volumes`: Prune volumes as well. **Use with caution!**

Multiple containers can use the same port but there will be only 1 port in host corresponding to that.

Dockerfile Explained (Common Instructions)

A `Dockerfile` is a script containing instructions to build a Docker image.

Key commands:

- `FROM` – base image (e.g., `python:3.9`)
- `COPY` – copy files into the container
- `RUN` – execute commands during build (e.g., install packages)
- `CMD` / `ENTRYPOINT` – command to run when container starts
- `EXPOSE` – declare port

Docker Compose (Managing Multi-Container Applications)

What it is:
A tool to define and run **multi-container** Docker applications using a YAML file (`docker-compose.yml`).

Use:
Great for setting up services like **web app + database** in a single command.

Docker Compose is a tool for defining and running multi-container Docker applications. It uses a YAML file (`docker-compose.yml` by default) to configure the application's services, networks, and volumes.

Why use it? Simplifies the management of applications with multiple interconnected services (e.g., a web application with a database and a caching layer).

Common Docker Compose Commands:

- `docker-compose up`: Build (if necessary), create, start, and attach to containers for a service.
  - `-d`: Run in detached mode.
  - `--build`: Force build images before starting containers.
- `docker-compose down`: Stop and remove containers, networks, and volumes defined in the Compose file.
  - `--volumes`: Remove named volumes declared in the `volumes` section. **Use with caution!**
- `docker-compose ps`: List containers managed by Compose.
- `docker-compose logs`: View logs from services.
  - `-f`: Follow logs.
- `docker-compose exec <service_name> <command>`: Execute a command in a running service container.
  - *Example:* `docker-compose exec webapp bash`
- `docker-compose build [service_name]`: Build or rebuild services.
- `docker-compose pull [service_name]`: Pull service images.
- `docker-compose stop [service_name]`: Stop services.
- `docker-compose start [service_name]`: Start services.
- `docker-compose rm [service_name]`: Remove stopped service containers.

The `.dockerignore` file is like `.gitignore`, but for Docker. It **tells Docker which files and folders to exclude** when building your image.

10. Common Use Cases

- **Development Environments:** Create consistent, isolated environments for developers.
- **CI/CD Pipelines:** Build, test, and deploy applications reliably and quickly.

- **Microservices:** Deploy and scale individual components of a larger application independently.