Docker

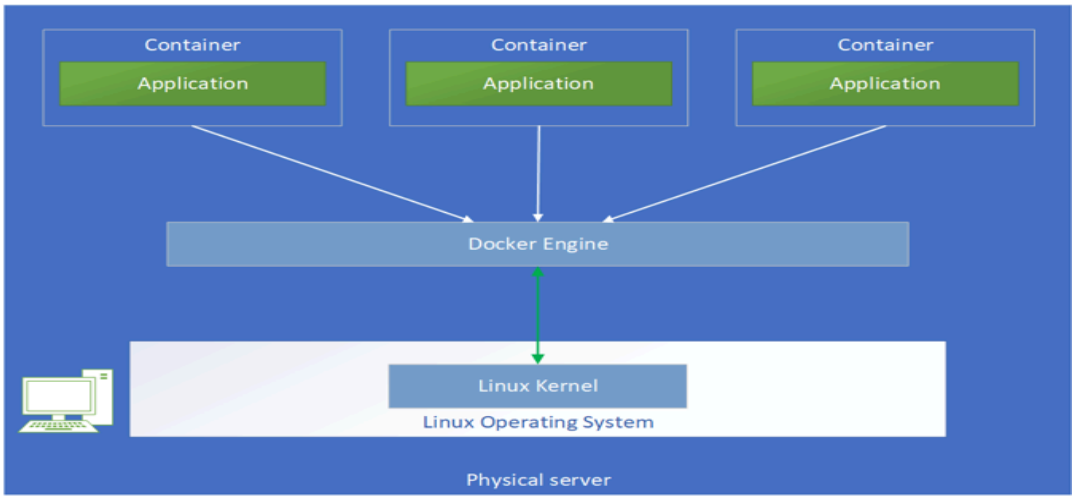
What is Docker?

Docker is a framework for building, running, and managing containers on servers and the cloud. It includes tools (commands, daemon) and the Dockerfile format. Traditionally, web apps ran on physical servers with manual setup, but cloud computing now uses redundant, software-based servers. Technologies like Linux namespaces and cgroups enable virtualization, making containers a lightweight mix of Linux OS and a localized runtime for efficient deployment.

Docker Containers

Container technology is categorized into three different categories:
1. Builder: Tools for creating containers (e.g., Dockerfile).
2. Engine: Applications for running containers like docker and dockerd daemon.
3. Orchestration: Tools for managing multiple containers such as Kubernetes and OKD.

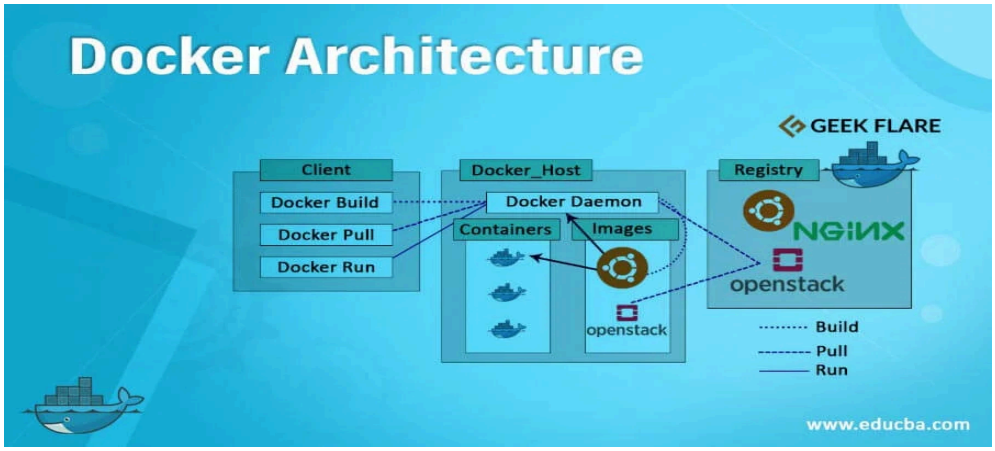
Containers bundle both applications and configurations, reducing setup time compared to traditional installations. Docker Hub provides repositories with container images for easy deployment.



Docker Core Architecture

When Docker was initially launched, it had a monolithic architecture. Now it is separated into the following three different components:

1. Docker Engine (dockerd)
2. docker-containerd (containerd)
3. docker-runc (runc)



Docker Engine:-

->Docker engine comprises the docker daemon, an API interface, and Docker CLI. Docker daemon (dockerd) runs continuously as dockerd systemd service. It is responsible for building the docker images.

To manage images and run containers, dockerd calls the docker-containerd APIs.

docker-containerd (containerd):-

->containerd is another system daemon service than is responsible for downloading the docker images and running them as a container. It exposes its API to receive instructions from the dockerd service

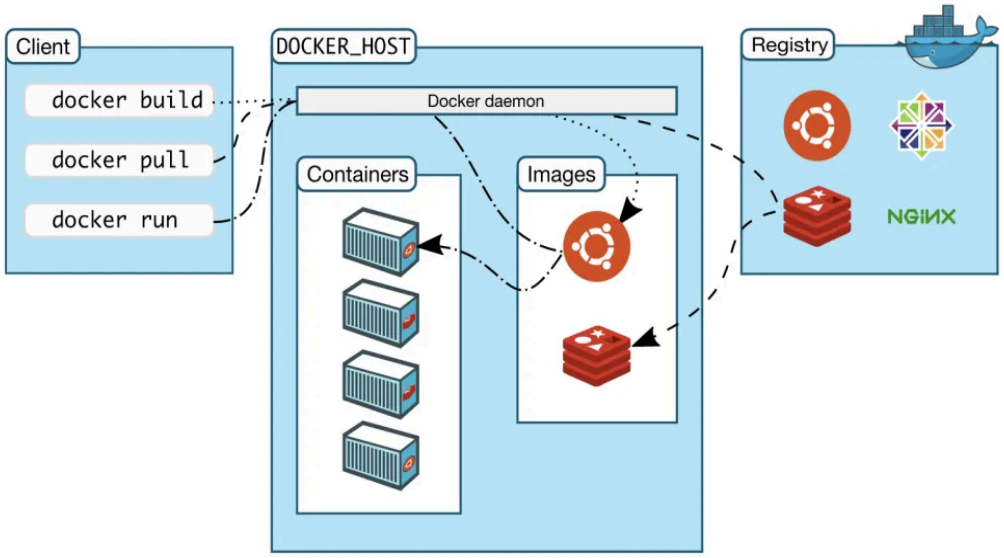
docker-runc:-

->runc is the container runtime responsible for creating the namespaces and cgroups required for a container. It then runs the container commands inside those namespaces. runc runtime is implemented as per the OCI specification.

Note : containerd is responsible for managing the container and runc is responsible for running the containers (create namespaces, cgroups and run commands inside the container) with the inputs from containerd

How Does Docker Work?

We will understand Docker Workflow by having a clear look at its high-level docker architecture.



1. Docker Host:-

a. Provides the environment to execute and run applications.

b. Contains the Docker daemon (dockerd), images, containers, networks, and storage.

2. Docker Daemon (dockerd):-

a. Monitors API requests and manages Docker objects (containers, images, volumes, networks).

b. Can communicate with other daemons for managing Docker services.

c. Listens to the docker.sock UNIX socket by default; can be exposed for remote access.

3. Docker Client:-

a. The primary interface for users to interact with Docker.

b. Sends commands via CLI (docker build, docker pull, docker run) to dockerd.

c. Can communicate with multiple Docker daemons.

docker-compose file

Introduction:-

-> Docker Compose is a Docker tool used to define and run multi-container applications. With Compose, you use a YAML file to configure your application's services and create all the app's services from that configuration.In other words, docker-compose is an automated multi-container workflow.

The most popular features of Docker Compose are:-

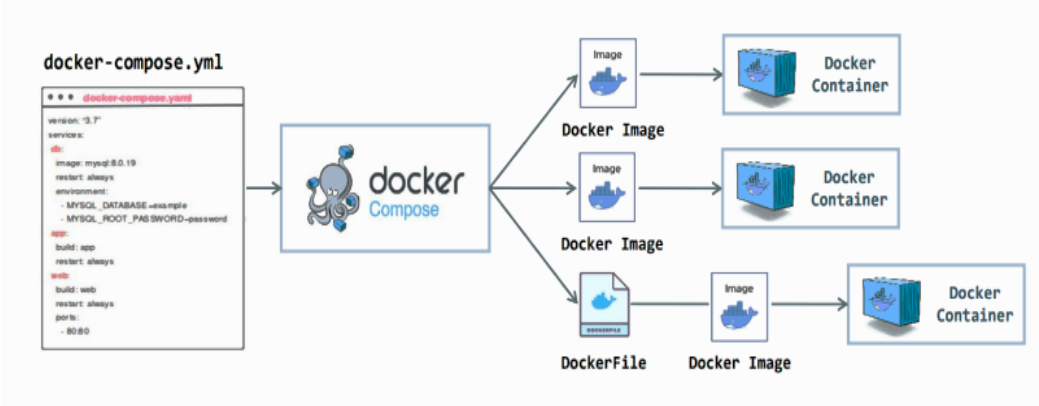
a. Multiple isolated environments on a single host

b. Preserve volume data when containers are created

c. Only recreate containers that have changed

d. Variables and moving a composition between environments

e. Orchestrate multiple containers that work together



Docker Compose file structure

Docker Compose files work by applying multiple commands that are declared within a single docker-compose.yml configuration file.

There are four main things almost every Compose-File should have which include:

a. The version of the compose file

b. The services which will be built

c. All used volumes

d. The networks which connect the different services

The basic structure of a Docker Compose YAML file looks like this:

```
version: '3.8'

services:
   db:
     image: mysql:8.0.34
     volumes:
       - db_data:/var/lib/mysql
     restart: on-failure
     environment:
       MYSQL_ROOT_PASSWORD: somewordpress
       MYSQL_DATABASE: wordpress
       MYSQL_USER: wordpress
       MYSQL_PASSWORD: wordpress
     networks:
       - malikdha-network
    web:
        build:
            context: .
            dockerfile: docker/php.Dockerfile
        ports:
            - '9000:9000'
        env_file:
            - '.env'
        volumes:
            - type: bind
              source: src/
              target: /var/www/html
        networks:
            - malidkha-network
        depends_on:
            db:
```

```
volumes:
    db_data:
networks:
    malikdha-network:
```

This Docker Compose file sets up a PHP application with a MySQL database, treating each service as a separate container.

a. version: '3.8' – Specifies the Compose file version, ensuring feature compatibility.

b. services – Defines containers (web and db).

c. db – Uses a MySQL image from Docker Hub.

d. web – Built from a custom Dockerfile (docker/php.Dockerfile).

e. build – Specifies the build context.

f. ports – Maps container ports to the host (9000:9000).

g. env_file – Loads environment variables from .env.

h. volumes – Mounts the PHP code directory and persists database data (db_data).

i. networks – Defines a custom network for communication.

j. depends_on – Ensures db starts before web.

k. image – Pulls a pre-built image when no Dockerfile is used.

l. environment – Sets environment variables inside the container.

m. restart – Ensures automatic container restarts on failure.

Docker Compose commands

Basic commands:

We will list and discuss the main and most commonly used docker-composecommands.

Command	Description
`$ docker-compose build`	Build or rebuild images from docker-compose.yml
`$ docker-compose run`	Create and start a container for a service
`$ docker-compose up`	Build (if needed), create, start, and attach
`$ docker-compose down`	Stop and remove containers, networks, volumes
`$ docker-compose kill`	Force stop containers with `SIGKILL`
`$ docker-compose start`	Start existing containers
`$ docker-compose restart`	Restart stopped/running services
`$ docker-compose stop`	Stop running containers without removing them
`$ docker-compose pause`	Pause running containers
`$ docker-compose unpause`	Unpause paused containers
`$ docker-compose rm`	Remove stopped service containers
`$ docker-compose images`	List images used by created containers
`$ docker-compose ps`	List running containers
`$ docker compose ls`	List running compose projects
`$ docker compose cp`	Copy files/folders between container & host
`$ docker-compose cp ~/. /var/www/html/app`	Copy from host to service
`$ docker-compose cp web:/var/www/html/app ~/`	Copy from service to host

Docker-compose file reference

Building:-

The base image of a container can be defined by either using a preexisting image that is available on DockerHub (or locally) or by building images using a Dockerfile.

```yaml
web:
  # build from Dockerfile
  build: .
  # build from custom Dockerfile
  build:
    context: ./dir
    dockerfile: Dockerfile.dev
  # build from image
  image: ubuntu
  image: ubuntu:22.04
  image: a4bc65fd
```

Note : Take good care of the build context, as it affects our build paths and .dockerignore files. One way i always use , is specifying context: . and put .dockerignore file in the root directory, that way your build context is always your root directory.

Ports:-

->ports is used to map the container's ports to the host machine. expose instead is used to publish the ports to the linked services of the container and not to the host system.

```yaml
ports:
  - "3000"
  - "8000:80"  # guest:host
# expose ports to linked services (not to host)
expose: ["3000"]
```

Commands:-

->Commands are used to execute actions once the container is started and act as a replacement for the CMD action in your Dockerfile (if multiple commands are specified , only the last one takes effect).

```
# command to execute
command: npm run start
command: [npm, run, start]

# override the entrypoint
entrypoint: /app/start.sh
entrypoint: [php, -d, vendor/bin/phpunit]
```

Environment variables:-

->Environment variables are used to bring configuration data into your applications. There are many different options of passing environment variables in our Compose file which we will explore below:

```
# environment vars
environment:
  WORK_ENV: development # Setting an environment variable
environment:
  - WORK_ENV=development # Setting an environment variable
environment:
  - WORK_ENV # Passing an environment variable


# environment vars from file
env_file: .env
env_file: [.env, .development.env]
```

Dependencies:-

->Dependencies in Docker are used to make sure that a specific service is available before the dependent container starts. This is often used if you have a service that can't be used without another one e.g. a Web application without its database.

```
# makes the `db` service available as the hostname `database`
# (implies depends_on)
links:
  - db:database
  - redis

# make sure `db` is alive before starting
depends_on:
  - db
```

Volumes:-

->Volumes are Docker's preferred way of persisting data which is generated and used by Docker containers. They are completely managed by Docker and can be used to share data between containers and the Host system.

```
volumes:
  # Just specify a path and let the Engine create a volume(Anonymous volume)
  - /var/lib/mysql
  # bind mapping
  - /opt/data:/var/lib/mysql.
  # named volume
  - datavolume:/var/lib/mysql

  # explicilty (type is "bind" or "volume"...)
  - type: bind
    source: src/
    target: /var/www/html
```

Docker Restart Policy:-

->Restart policies are strategies we can use to restart Docker containers automatically and manage their lifecycles.

Given that containers can fail unexpectedly, Docker has safeguards to prevent services from running into a restart loop. In case of a failure, restart policies don't take effect unless the container runs successfully for at least 10 seconds.

Docker restart policy options:-

a. no – Containers won't restart automatically.

b. on-failure[:max-retries] – Restarts only if the container exits with an error, with optional max retries.

c. always – Always restarts the container if it stops.

d. unless-stopped – Restarts unless manually stopped or by the Docker daemon.

```
db:
  restart: on-failure
```

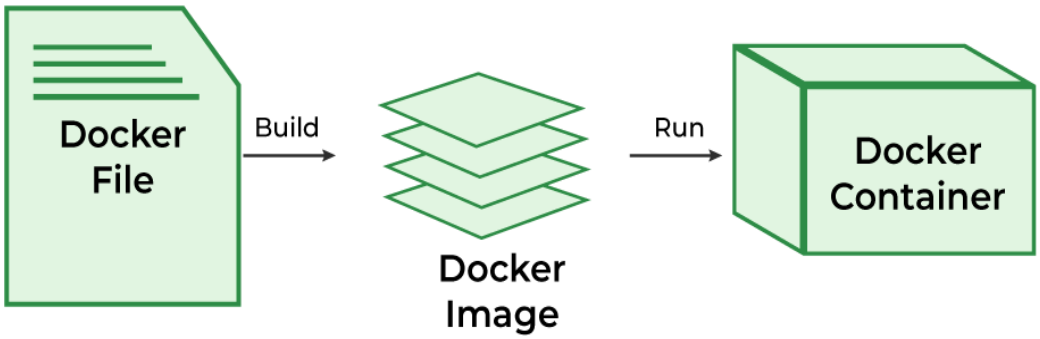
Dockerfile

->Dockerfiles are text files that list instructions for the Docker daemon to follow when building a container image

When you execute the docker build command, the lines in your Dockerfile are processed sequentially to assemble your image (in a layered fashion).

The basic Dockerfile syntax looks as follows:

```
# Comment
INSTRUCTION arguments
```



Lines starting with a # character are interpreted as comments. They'll be ignored by the parser, so you can use them to document your Dockerfile.

Example

```
FROM ubuntu:22.04
COPY . /app
RUN make /app
CMD python /app/app.py
```

In the example above, each instruction creates one layer:-

a. FROM creates a layer from the ubuntu:22.04 Docker image.

b. COPY adds files from your Docker client's current directory.

c. RUN builds your application with make.

d. CMD specifies what command to run within the container.

Dockerfile Instructions Reference

Docker supports over 15 different Dockerfile instructions for adding content to your image and setting configuration parameters. We will discuss all of them. It runs instructions in a Dockerfile in order. A Dockerfile must begin with a FROM instruction. (ARG is the only instruction that may precede FROM in the Dockerfile.)

List of the instructions

```
FROM nginx:latest
```

We use FROM to specify the base image we want to start from. (base image and also alias for multi-stage build)

The tag values are optional. If you omit it, the builder assumes a latest tag by default. The builder returns an error if it cannot find the tag value.

Set the Author field of the generated images.

```
LABEL name="malidkha"
LABEL email="malidkha.elmasnaoui@gmail.com"
LABEL description="This is a Dockerfile reference"
```

Labels are key-value pairs and simply adds custom metadata to your Docker Images

```
#shell form
RUN apt-get update -y && apt-get install -y procps
RUN chown -R malikdha:malikdha /var/log/nginx && \
    chown -R malikdha:malikdha /etc/nginx

#exec form
RUN ["apt-get", "update", "-y"]
```

RUN is used to run commands during the image build process.

(installing packages using apt , npm , composer ..)

Shell form : RUN COMMAND (by default is /bin/sh -c)

Exec form : RUN ["", "", ""]

Change shell In which to run the command : RUN ["/bin/bash", "-c", "echo hello from bash"]

```
#shell form
CMD nginx -g daemon off

#exec form
CMD ["/usr/sbin/nginx", "-g", "daemon off;"]
```

CMD: Executes a command within a running container

Shell form : CMD Exec form : CMD ["executable", "param1", "param2"]

Change shell In which to execute the command : CMD ["/bin/bash", "-c", "echo hello from bash"]

CMD Runs when the containers starts , only one can be executed, the last one

We can override the CMD instruction using the docker run command. If we specify a CMD in our Dockerfile and one on the docker run command line, then the command line will override the Dockerfile 's CMD instruction.

```
#shell form
ENTRYPOINT nginx -g daemon off

#exec form
ENTRYPOINT ["/usr/sbin/nginx", "-g", "daemon off;"]
```

ENTRYPOINT: Defines the command executed when the container starts.

Shell form: ENTRYPOINT command param1 param2 (default /bin/sh -c).

Exec form: ENTRYPOINT ["executable", "param1", "param2"].

Override with --entrypoint COMMAND in docker run.

```
ENTRYPOINT ["/usr/sbin/nginx"]
CMD ["-g", "daemon off;"]
```

CMD: Provides default arguments for ENTRYPOINT (if defined).

SHELL: Changes the default shell (default is ["/bin/sh", "-c"]).

Example: SHELL ["/bin/bash", "-c"].

EXPOSE: Informs Docker that the container listens on specific ports (does not open them).

Example: EXPOSE 80/tcp.

ARG: Defines build-time variables (not available at runtime).

Example: ARG NAME=value.

ENV: Defines environment variables accessible during build and runtime.

Example: ENV NAME=value.

COPY: Copies files from the host to the container.

ADD: Similar to COPY but supports remote URLs and auto-extracts tar files.

WORKDIR: Sets the working directory inside the container.

Example: WORKDIR /var/www/html.

USER: Sets the user for subsequent commands.

Example: USER username.

VOLUME: Creates a mount point for persistent storage.

Example: VOLUME /var/log/nginx.

ONBUILD: Triggers actions when the image is used as a base image.

Example: ONBUILD RUN apt-get update.

HEALTHCHECK: Monitors container health.

Example:

dockerfile

```
HEALTHCHECK --interval=30s --timeout=3s \
  CMD curl -f http://localhost/ || exit 1
```

Healthcheck Exit Codes:

0: Healthy

1: Unhealthy

2: Reserved (do not use)

Multi-Stage Build

a. Allows defining multiple stages in a Dockerfile.

b. Optimizes image size by removing unnecessary files.

c. Improves build performance with caching.

How it Works:-

a. Multiple FROM statements create build stages.

b. Use COPY --from= to copy artifacts between stages.

Example:

```dockerfile
# Build stage
FROM golang AS build
WORKDIR /app
COPY . .
RUN go build -o app
ENTRYPOINT ["./app"]

# Final minimal image
FROM alpine:latest
RUN apk --no-cache add ca-certificates
WORKDIR /root/
COPY --from=build /app/app .
CMD ["./app"]
```

a. Naming Stages: Use AS in FROM.

b. Stopping at a Specific Stage: docker build --target -t .

c. Copying from External Images: COPY --from=nginx:latest /etc/nginx/nginx.conf /nginx.conf

.dockerignore File:-

->Excludes unnecessary files from the Docker build context.

Improves build speed and security.

Benefits:

a. Prevents cache invalidation from frequent updates.

b. Reduces image size → Faster build, pull, and push.

c. Prevents sensitive data leaks (.aws, .env, .git).

Example `.dockerignore`**:**

```
**/node_modules/
npm-debug.log
.env
.aws
.git
Dockerfile
docker-compose*
**/*.swp
```