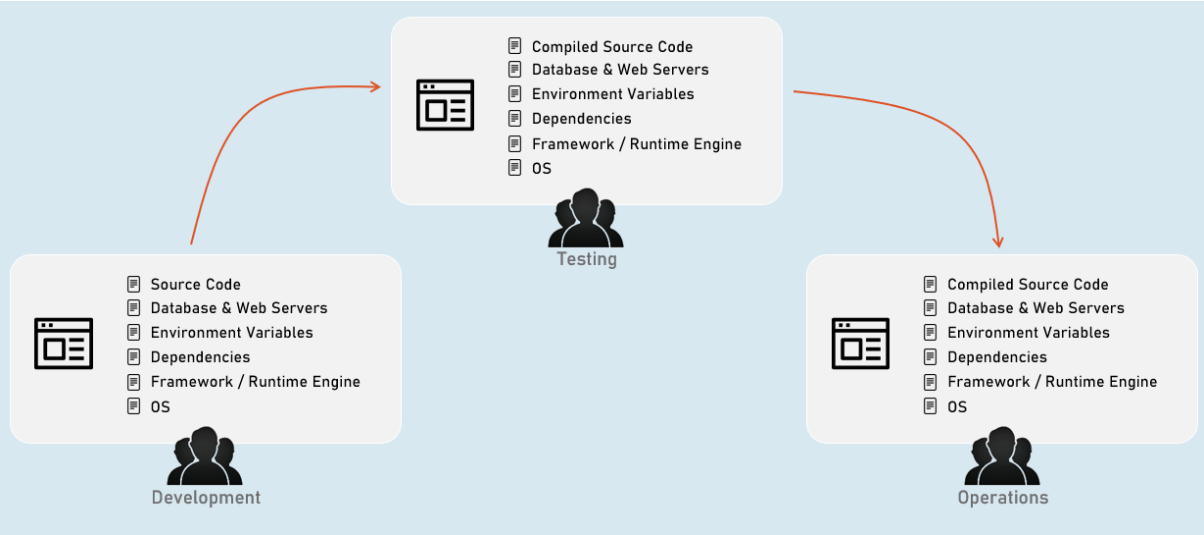
.NET Core Microservices – True Ultimate Guide

Section 4 – Docker – Cheat Sheet

Deployment Workflow



Hypervisor

Hypervisor is a software that lets you to create and run multiple virtual machines; where each virtual machine can have its own operating system based on pre-allocated hardware resources.

Architecture:

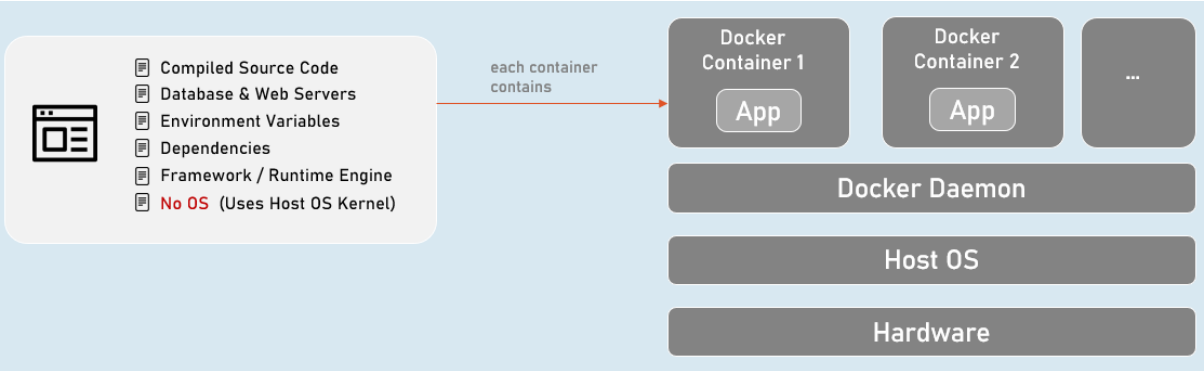
- App
- Guest OS
- Hypervisor (VMWare / VirtalBox etc.)
- Host OS
- Hardware

Problems:

- Manual effort to replicate environment.
- Large foot-print and wastage of hardware resources.
- Error-prone due to config mismatches, differences in versions of dependencies, missing dependencies or other human-errors.
- Difficult to scale-out as it needs substantial investment in hardware resources.

Introduction to Docker

Docker is a containerization tool that packages applications with their dependencies for consistent and efficient deployment across diverse environments.



Docker Images [vs] Containers

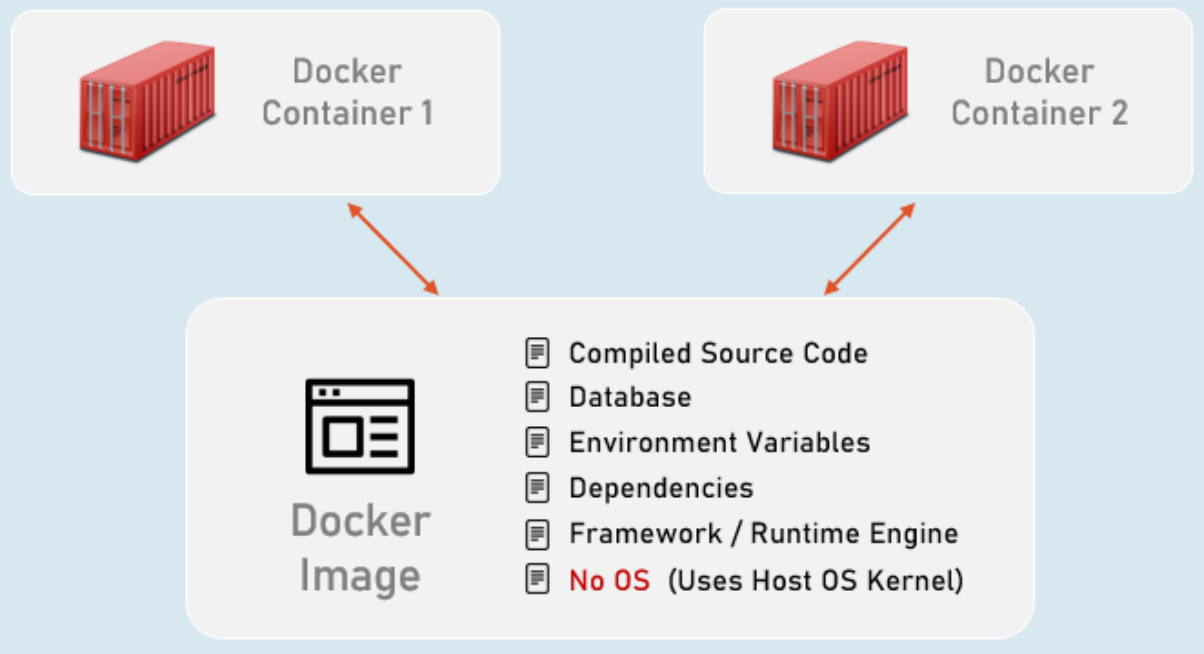
Docker Image

A self-contained package with all the components needed to run a software application, such as code, dependencies, and environment settings.

Docker Container

A lightweight, isolated, and executable instance created from a Docker image.

You can create multiple containers based on a docker image.



Benefits of Containerization

1. Isolation

Containers provide a level of isolation for applications, preventing conflicts with other applications or the host system.

Each container operates independently, enhancing security and reliability.

2. Portability

Containers encapsulate applications and their dependencies, ensuring consistency across various environments.

This portability enables developers to build, test, and deploy applications seamlessly across different platforms.

3. Resource Optimization

Containers share the host OS's kernel and use resources more efficiently than traditional virtual machines.

This allows for higher density on a given host, optimizing resource usage.

4. Consistency

Containers include everything needed to run an application, from compiled-code to dependencies.

This ensures consistency between development, testing, and production environments, reducing the likelihood of "it works on my machine" issues.

5. Scalability

Using containers with microservices lets you easily scale specific parts of your application by running each service in its own container, offering flexibility and efficient resource utilization.

6. Rapid Deployment

Containers can be started and stopped quickly, enabling rapid deployment and scaling of applications.

This agility is especially beneficial in dynamic and evolving development environments.

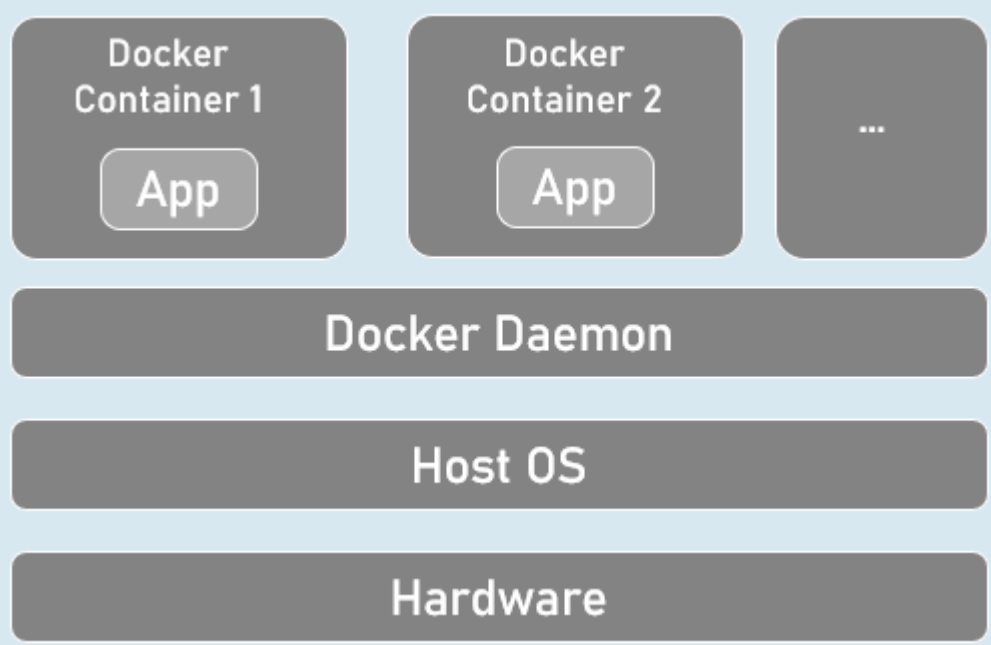
7. Versioning and Rollback

Docker enables versioning of images, making it easy to roll back to a previous version if issues arise.

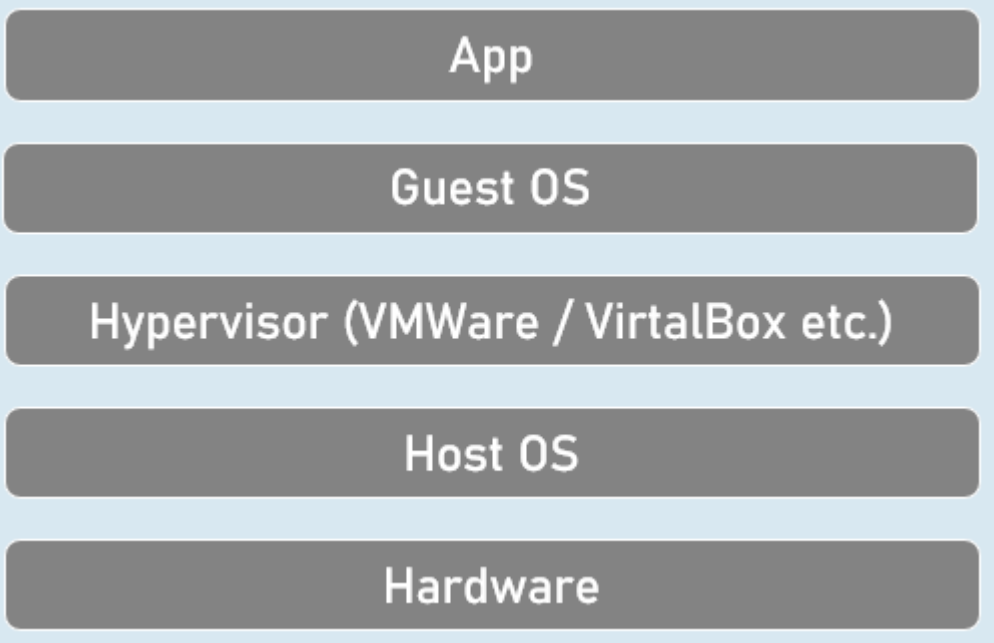
This enhances the ability to manage and control changes in application releases.

Containers vs Virtual Machines

Containers:



Virtual Machines using Hypervisor:



Differences:

1. Isolation

VMs: Provide strong isolation by virtualizing the entire operating system.

Containers: Offer lightweight isolation, sharing the host OS's kernel, resulting in less overhead and faster startup times.

2. Resource Utilization

VMs: Have a heavier footprint due to duplicating the entire OS for each instance.

Containers: Use fewer resources as they share the host OS's kernel, making them more efficient.

3. Portability

VMs: Can be less portable due to differences in underlying hypervisors.

Containers: Are highly portable, ensuring consistency across various environments.

4. Performance

VMs: Tend to have slightly higher overhead and longer startup times.

Containers: Have lower overhead and faster startup, promoting rapid scaling and deployment.

5. Scaling

VMs: Scale by deploying additional virtual machines, which can be slower.

Containers: Scale more quickly, allowing independent scaling of microservices.

6. Deployment Speed

VMs: Typically have longer deployment times due to the need to boot an entire OS.

Containers: Deploy quickly since they only need to start the application and its dependencies.

7. Use Cases

VMs: Suitable for running different operating systems on the same hardware.

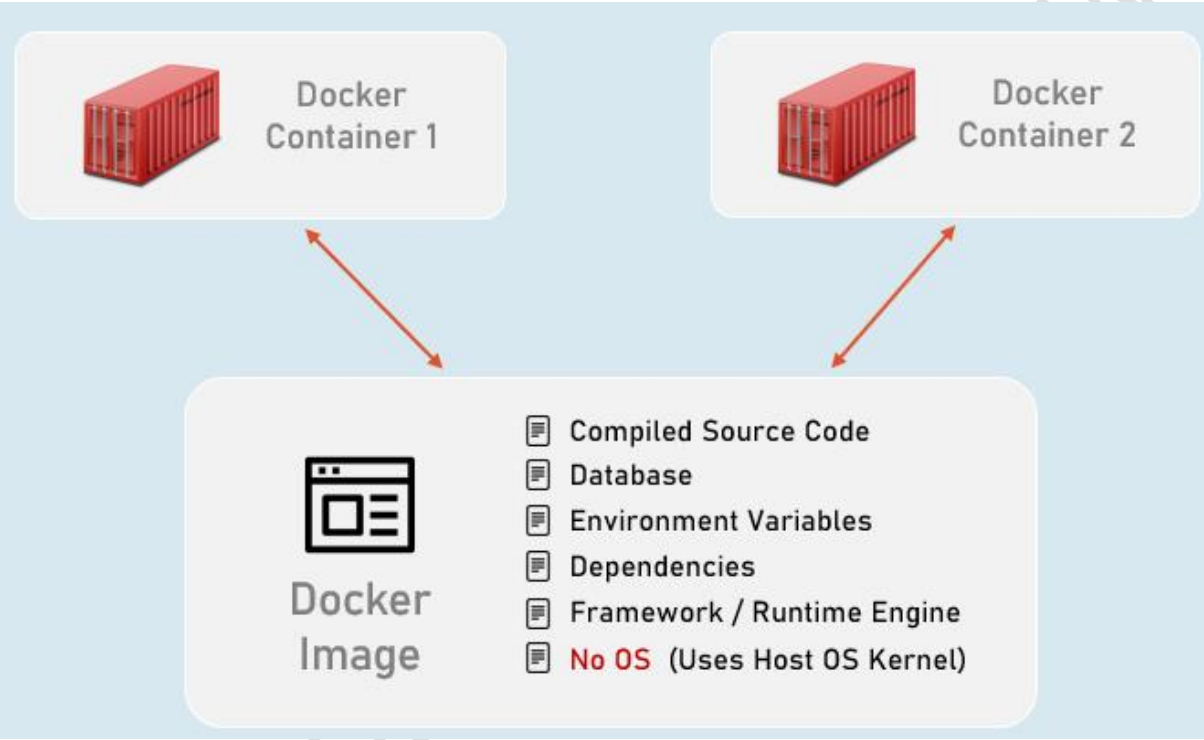
Containers: Ideal for microservices architectures, where services can run in isolated containers.

8. Overhead

VMs: Have higher overhead because of the need for separate OS instances.

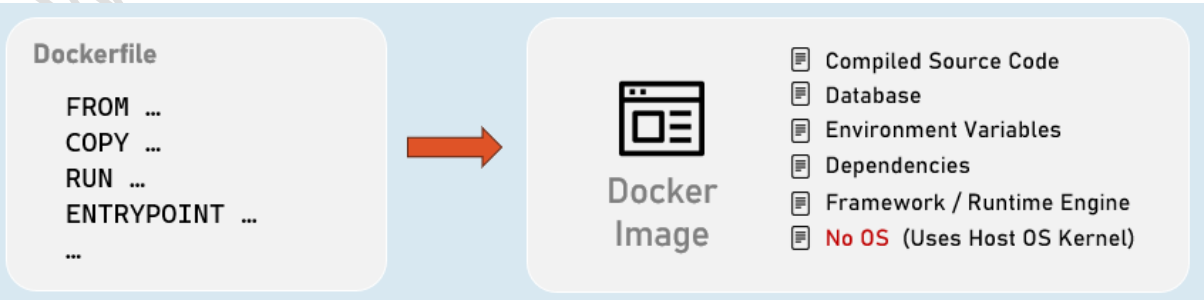
Containers: Have lower overhead, making them more lightweight.

How Docker Images and Containers Work Internally?



1. Dockerfile

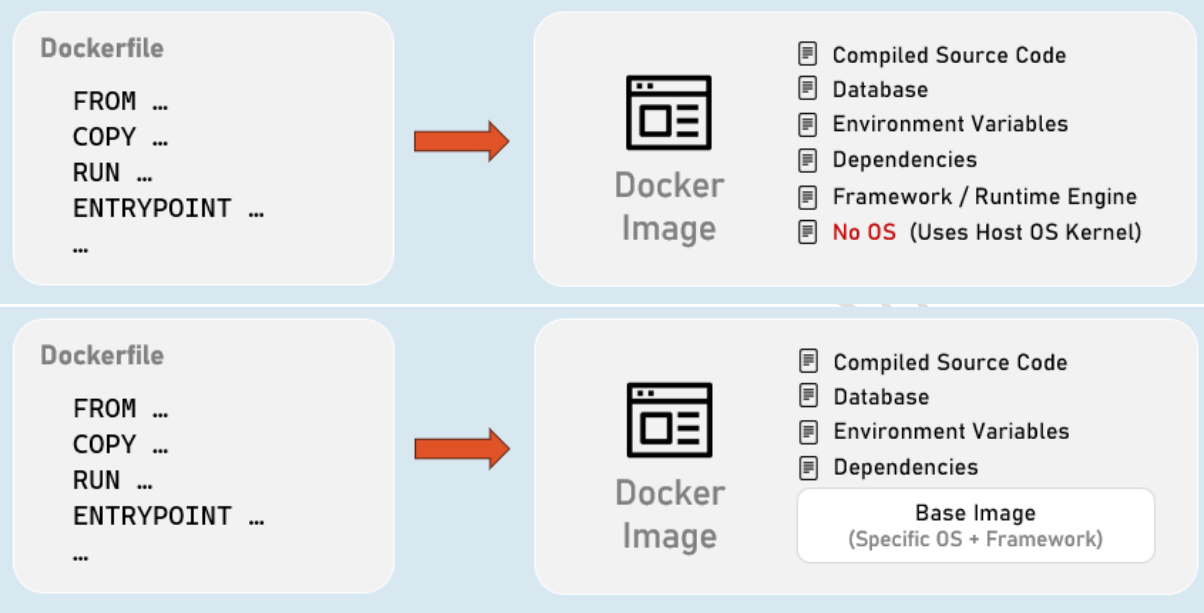
Images are created using Dockerfiles, which specify the steps to build the image, including copying files, installing dependencies, and configuring settings.



2. Base Image

Images are built from a base image that contains the essential OS components. A base docker image targets to a specific OS and contains a specific framework or runtime engine.

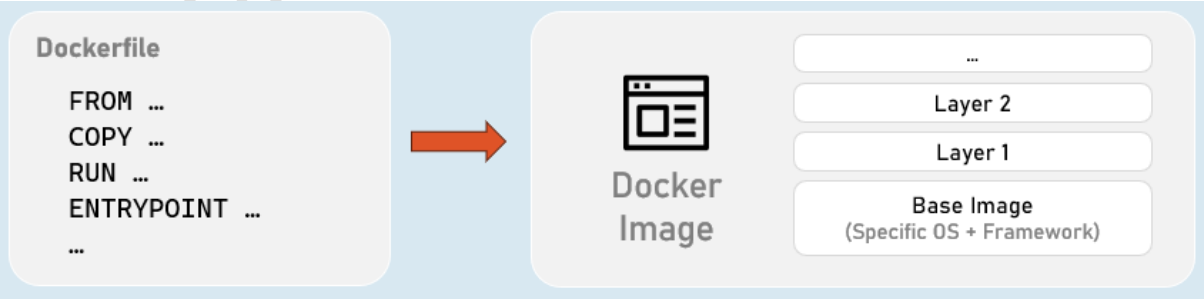
e.g:"dotnet/framework","dotnet/core","python-slim","node" etc.



3. Layered File System

Docker images use a layered file system. Each instruction of "Dockerfile" creates a new layer in the image.

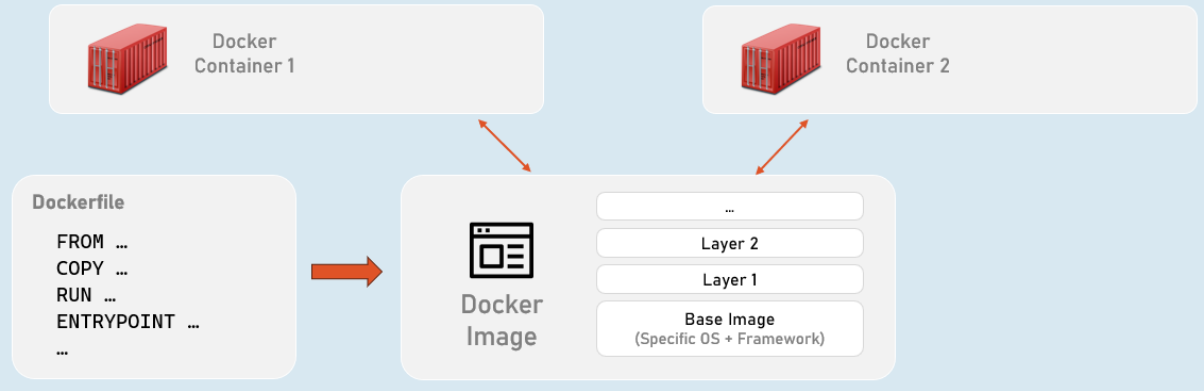
Each layer stores the changes made in the file system or configuration on top of its previous layer.



4. Instantiation

Containers are instances of Docker images.

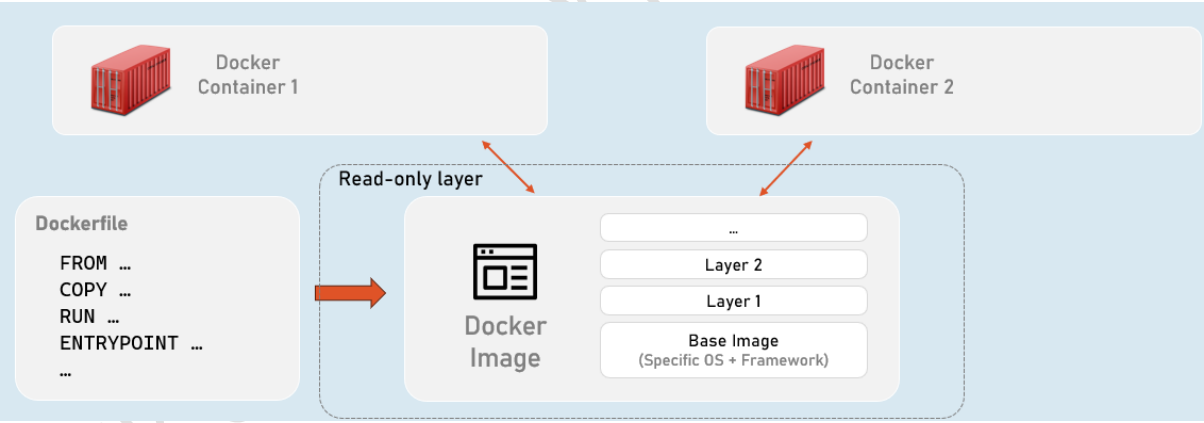
Containers doesn't store a copy of any data from its image; but can access the file system or environment variables of the parent image.



5. Immutable Images

Images are immutable, meaning once created, they cannot be changed.

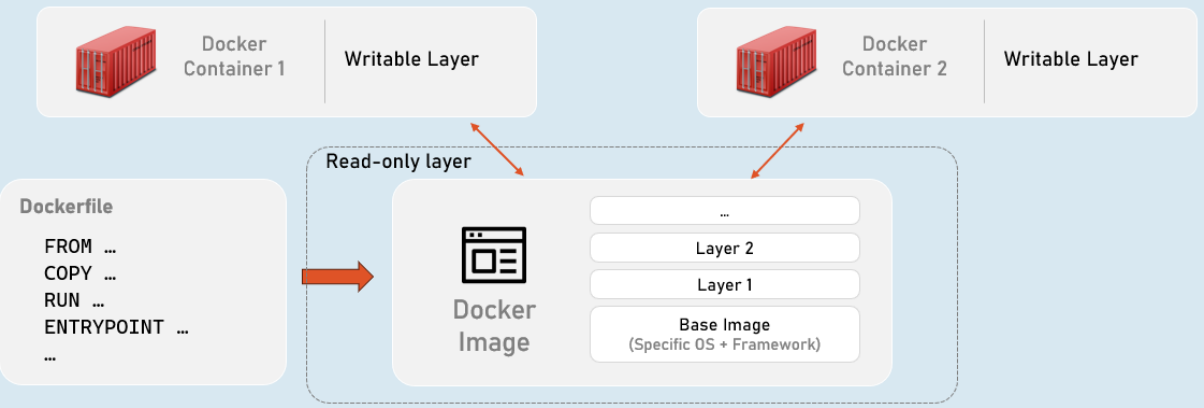
Once the final image is created, it's wrapped in a read-only layer.



6. Writable Containers

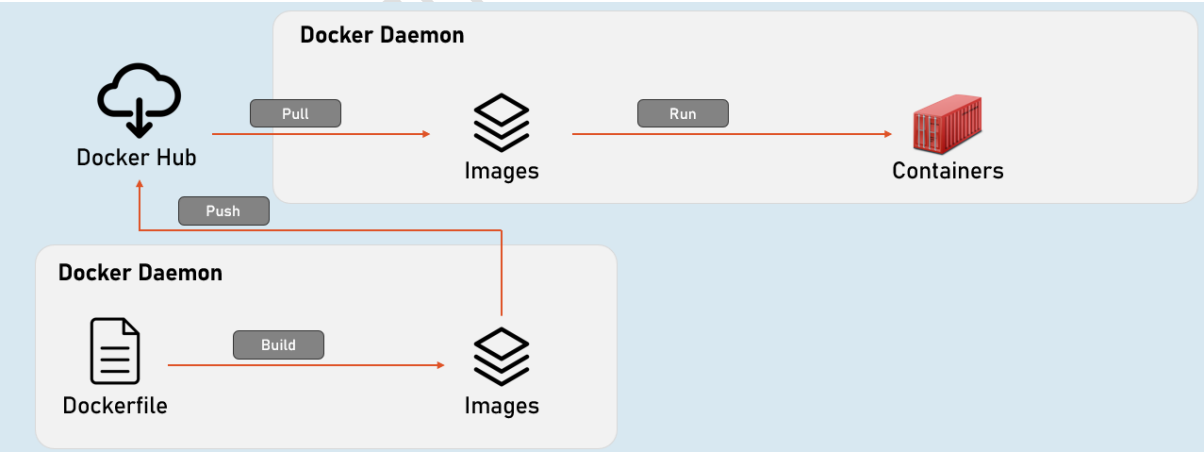
Containers have a thin, writable layer on top of the image which contains any changes made to the file system or temporary data generated while container is running.

Each change at run time would be stored as a layer in the writable layer of the container.

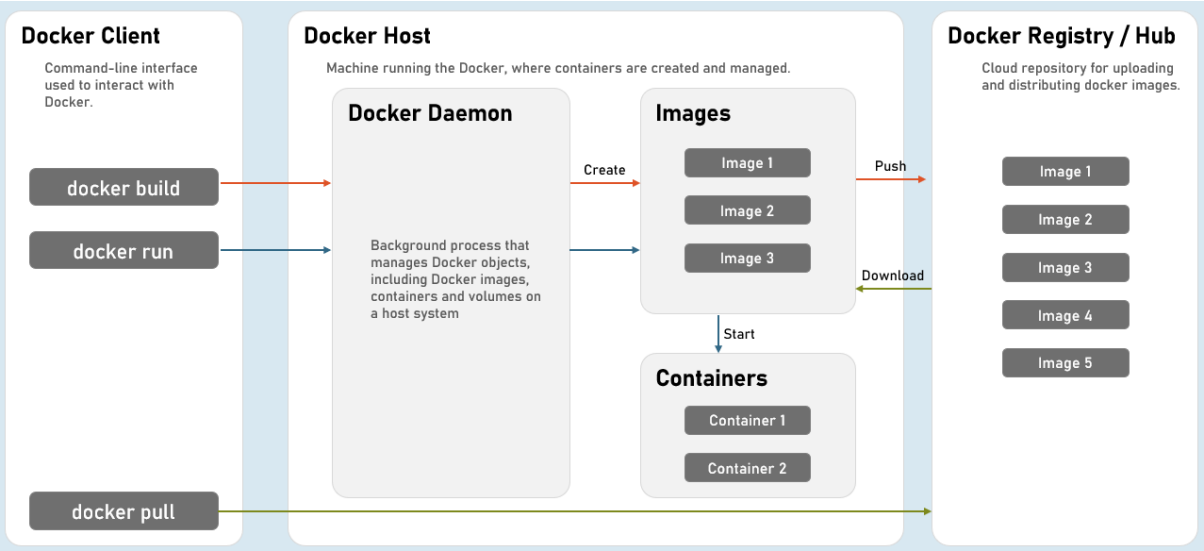


Docker Hub

Docker Hub is a cloud repository for uploading and distributing docker images.



Docker Architecture



Dockerfile

A Dockerfile is a script file that contains a set of instructions used to create a Docker image.

FROM

Specifies the base image for the container.

Syntax:FROM <image>:<tag>

USER

Specifies the user that should be switched to, before executing subsequent commands.

Syntax: USER <user_name_or_id>

WORKDIR

Sets the working directory (in the image file system).

Syntax: WORKDIR <path>

EXPOSE

Informs Docker that the container will listen to the specified internal port at run time.

Syntax: EXPOSE <port>

ARG

Defines variables that can be used within the build process or can be passed via "docker build" command.

Syntax: ARG <variable_name>=<default_value>

COPY

Copies specifies files or directories from the host machine into the image.

Syntax: COPY <source_path> <destination_path>

RUN

Executes the specified command during the build process.

Syntax: RUN <command>

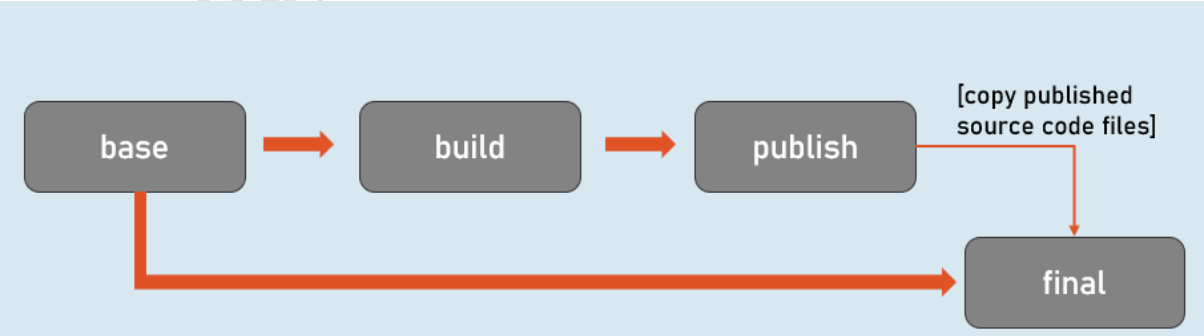
ENTRYPOINT

Specifies the primary command to be run when a container is started.

Syntax: ENTRYPOINT ["executable", "arg1", "arg2", "..."]

Multi-Stage Dockerfile

Multi-stage Dockerfile is a technique while helps in optimizing the final image size by creating intermediate images and eliminating unnecessary build artifacts.



Stage 1: "base"

FROM mcr.microsoft.com/dotnet/aspnet:8.0 AS base

USER app

WORKDIR /app

EXPOSE 8080

EXPOSE 8081

This stage sets up the base image for the application.

It uses the mcr.microsoft.com/dotnet/aspnet:8.0 image, sets the user to 'app', sets the working directory to /app, and exposes ports 8080 and 8081.

Stage 2: "build"

FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build

ARG BUILD_CONFIGURATION=Release

WORKDIR /src

COPY ["WebApp/WebApp.csproj", "App/"]

RUN dotnet restore "./WebApp/./WebApp.csproj"

COPY . .

WORKDIR "/src/WebApp"

RUN dotnet build "./WebApp.csproj" -c $BUILD_CONFIGURATION -o /app/build

This stage is responsible for building the application. It uses the "mcr.microsoft.com/dotnet/sdk:8.0" image and sets the build configuration to the specified value (default is Release).

It copies the project files, restores dependencies, and builds the application. The output is placed in the "/app/build" directory.

Stage 3: "publish"

FROM build AS publish

ARG BUILD_CONFIGURATION=Release

RUN dotnet publish "./WebApp.csproj" -c $BUILD_CONFIGURATION -o /app/publish /p:UseAppHost=false

This stage is used for publishing the application.

It uses the previous build stage (build) as its base, sets the build configuration, and runs the dotnet publish command.

The published files are placed in the /app/publish directory.

Stage 4: "final"

FROM base AS final

WORKDIR /app

COPY --from=publish /app/publish .

ENTRYPOINT ["dotnet", "WebApp.dll"]

This final stage uses the base image from the first stage (base).

It sets the working directory to /app, copies the published files from the publish stage into the current stage, and sets the entry point to start the application using dotnet.

The final image inherits only the necessary artifacts from the earlier stages, improving docker file security and reducing overall image size.

Creating Docker Images and Containers

docker build -t <image>:<tag> -f <Dockerfile_path> <context>

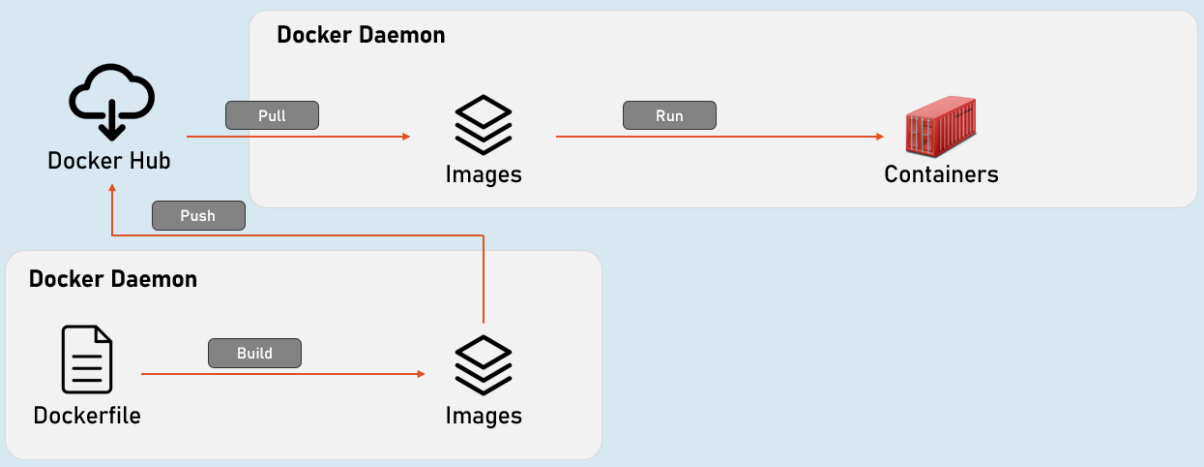
Builds (creates) a Docker image from a Dockerfile located in the specified path.

docker run --name <container_name> -e <env_variable>=<value> <image>:<tag>

Creates and starts a new container from the specified image.

Docker Hub Accounts

Docker Hub is a cloud repository for uploading and distributing docker images.



Docker Hub Repository

A repository is a collection of related Docker images, often organized by name and tags.

The combination of repository name and tag uniquely identifies a specific image.

Every tag (image) represents a specific version of the same Docker image.

Docker Tag

A tag is a label applied to a specific version of a Docker image within a repository.

It helps differentiate different versions or configurations of the same application.

Docker Hub Repositories

1. Public Repository

Visibility: Images in a public repository are visible to everyone. Anyone can search for, pull, and use these images.

Collaboration: Public repositories are suitable for open-source projects or scenarios where you want to share your containerized application with a broad audience.

Free Access: Typically, public repositories on Docker Hub are free to use.

2. Private Repository

Visibility: Images in a private repository are restricted, and access is controlled. Only authorized users can view or pull images from a private repository.

Security: Private repositories are essential for sensitive or proprietary applications where you want to control who can access and use your container images.

Paid Access: Private repositories often come with subscription fees. You may need a paid plan to create and manage private Docker images.

Pushing Docker Images

docker build -t <image>:<tag> -f <Dockerfile_path> <context>

Builds (creates) a Docker image from a Dockerfile located in the specified path.

docker tag <source_image>:<source_tag> <username>/<destination_image>:<destination_tag>

Create a tag or alias for an existing Docker image.

docker push <username>/<destination_image>:<destination_tag>

Pushes (uploads) the specified docker image into the specified Docker hub repository.

Docker on Linux

sudo apt-get update

`apt-get` is a command-line package management tool in Debian-based Linux distributions that facilitates the installation, upgrading, and removal of software packages, managing dependencies and providing a systematic approach to maintain the software ecosystem.

This command updates the "apt-get" command line tool to its latest version.

sudo apt install docker.io

Installs & starts Docker services on Ubuntu

Pulling Docker Images

sudo docker pull <username>/<image>:<tag>

Downloads the docker image from the container registry (Docker hub) into the local system & make it available for running containers.

sudo docker run -p <host_port>:<container_port> -e <env_variable>=<value> <username>/<image>:<tag>

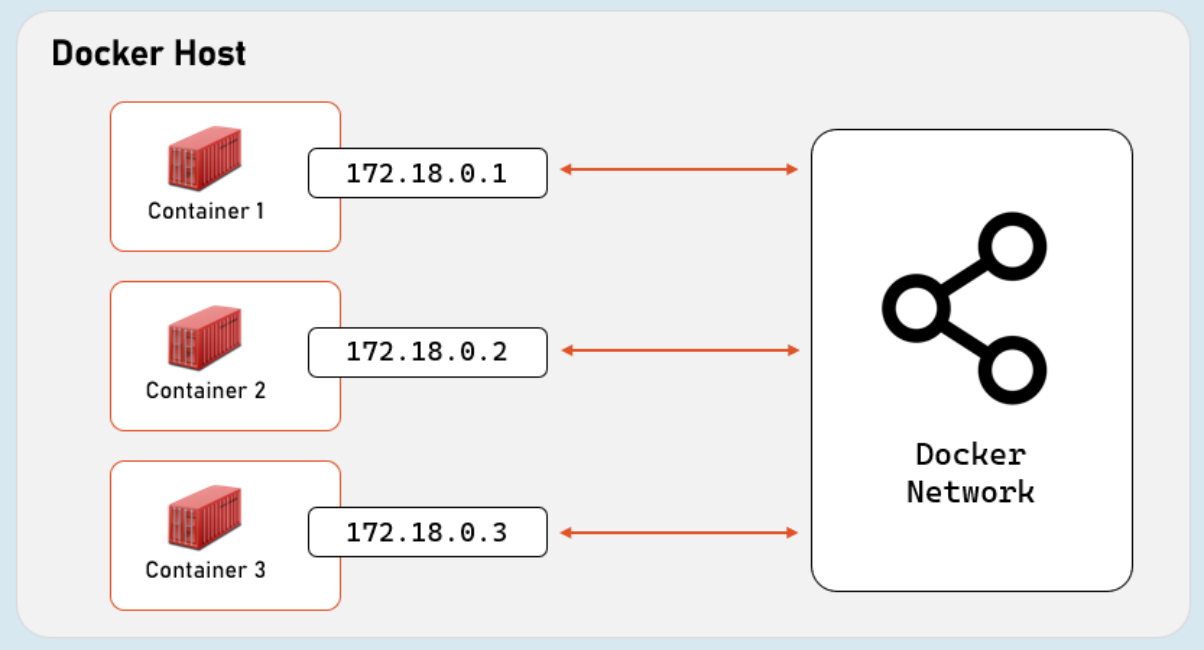
Creates and starts a new Docker container based on the specified Docker image.

It automatically pulls (downloads) the remote Docker image, if it is not pulled already.

If the remote Docker image is updated on Docker hub, you need to run "docker pull" command manually to download the updated docker image.

Docker Networks

A Docker network is a virtual network interface that enables communication between Docker containers.



Docker Network Drivers

1. Bridge

This is the default network driver in Docker.

It creates an internal network on the Docker host to which containers can connect.

Containers on the same bridge network can communicate with each other using their own IP addresses.

2. Host

3. None

4. Overlay

5. Macvlan