

# DOCKER - Zero To Beginner



## What is Docker ?

Docker is an open source platform that enables developers to build, deploy, run, update and manage containers—standardized, executable components that combine application source code with the operating system (OS) libraries and dependencies required to run that code in any environment.

## Docker terminologies :

**Images:** The blueprints of our application which form the basis of containers. These contain all of the configuration settings that define the isolated environment.

Docker images contain executable application source code as well as all the tools, libraries and dependencies that the application code needs to run as a container. When you run the Docker image, it becomes one instance (or multiple instances) of the container.

Multiple Docker images can be created from a single base image

Docker images are made up of layers and each layer corresponds to a version of the image. Whenever a developer makes changes to the image, a new top layer is created and this top layer replaces the previous top layer as the current version of the image. Previous layers are saved for rollbacks or to be re-used in other projects.

Each time a container is created from a Docker image, yet another new layer called the container layer is created. Changes made to the container—such as the addition or deletion of files—are saved to the container layer and exist only while the container is running.

This iterative image-creation process enables increased overall efficiency since multiple live container instances can run from just a single base image and when they do so, they leverage a common stack.

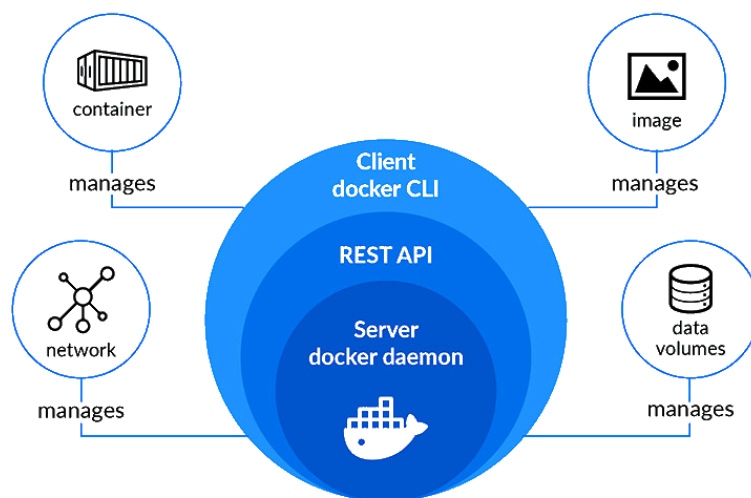
**Containers:** Are instances of a Docker image and are what run the actual application. A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings.

Container images become containers at runtime and in the case of Docker containers – images become containers when they run on Docker Engine.

We'll learn more about containers later in this session.

**Docker Daemon:** That background service running on the host that listens to API calls (via the Docker client), manages images and building, running and distributing containers. The Daemon is the process that runs in the operating system which the client talks to – playing the role of the broker.

Docker daemon is a service that creates and manages Docker images, using the commands from the client. Essentially the Docker daemon serves as the control center of your Docker implementation. The server on which Docker daemon runs is called the Docker host



## Docker Client:

Docker CLI is the most commonly used tool. It is the command-line interface used to control the Docker service. The Docker CLI enables you to interact with the system. To do this, we can use the Docker command, which will allow us to send and receive messages from the Docker daemon. The Docker command executes commands, creates images and networks, starts and stops containers, and more.

The Docker CLI receives commands, checks to ensure they are correctly formatted, and then turns them into a REST API call. The Docker daemon listens for requests from Docker

CLI and processes them according to its configuration. It is responsible for executing commands, processing requests, monitoring how containers run, mounting volumes, starting and stopping containers, and more.

The Docker client and daemon can both run on the same device. Alternatively, you can connect a Docker client to a remote Docker daemon. It allows you to manage a remote machine's Docker system. After all, the Docker client and daemon communicate with each other using a REST API over UNIX sockets or a network interface.

**Docker Hub:** A registry of Docker images containing all available Docker images. A user can have their own registry, from which they can pull images.

## Why are containers useful?

Portability – the isolated environment that containers provide effectively means the container is decoupled from the environment in which they run. Basically, they don't care much about the environment in which they run, which means they can be run in many different environments with different operating systems and hardware platforms.

Consistency – since the containers are decoupled from the environment in which they run, you can be sure that they operate the same, regardless of where they are deployed. The isolated environment that they provide is the same across different deployment environments.

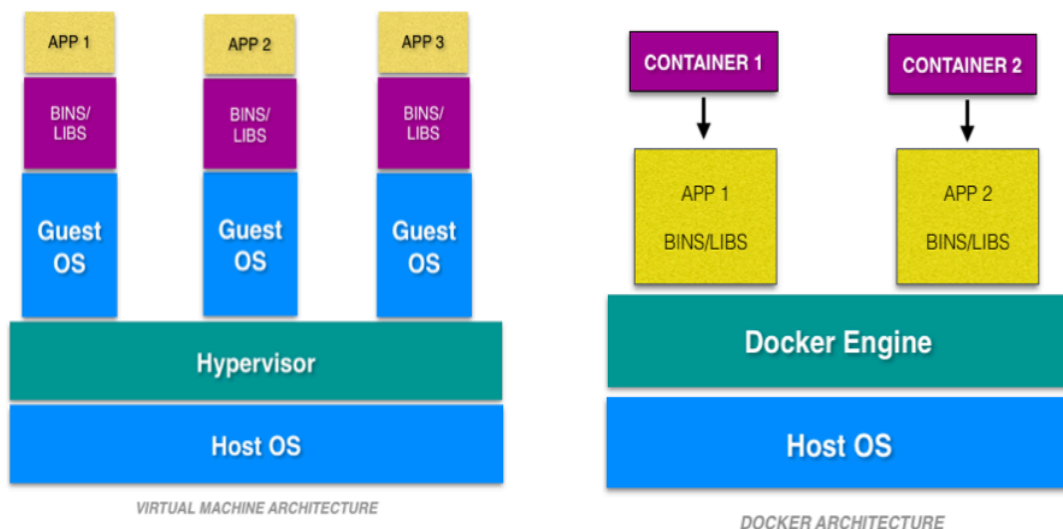
Speed to deploy – for the same reasons as above. There is no need for considerations around how the application will operate in a production environment. If it runs in a container in one environment (say, your local machine), then it can be made to run in a container in another environment (say, in a cloud provider) very quickly.

The parallel to the hypervisor layer with containers is the Docker daemon (assuming you're using Docker), it acts as the broker between the host OS and containers. It comes with less computational overhead than hypervisor software (as depicted by the thinner box in the figure above), again making containers more lightweight compared to VMs.

VMs suffer from duplication: many of the capabilities and features of the guest OS(s) are found in the host OS, so why not just use the host OS? This is what containers aim to do, whilst still providing isolation and decoupling from software in the host machine. With containers, only the things that the app absolutely needs are copied into the container, as opposed to VMs where the whole OS is installed – even the things from the OS that aren't used by the app.

## Virtual Machines VS Containers :

A virtual machine (VM) is another way of creating an isolated environment. A VM is effectively an individual computer that lives inside a host machine; multiple VMs can live inside a single host machine. VMs are created by virtualising the host machine's underlying hardware (processing, memory and disk). The hardware is virtualised and split up, with a piece representing a portion of the underlying physical hardware, which a VM can be run on.



As you can see in the figure above, the thing that sits between the VMs and the host is the hypervisor layer. The hypervisor is a piece of software that virtualises the host's hardware and acts as the broker: managing the virtualised hardware and feeding resources to the VMs.

This virtualisation process brings with it substantial computational overhead. Furthermore, since each VM is basically its own machine, they have their own OS installed, which typically require tens of gigabytes of storage, and which therefore takes time to install, which has to be done every time you want to spin up a new VM.

On the other hand, Containers take a different approach to producing isolation: like VMs, containers live on top of a host machine and use its resources, however, instead of virtualising the underlying hardware, they virtualise the host OS. Meaning containers don't need to have their own OS, making them much more lightweight than VMs, and consequently quicker to spin up.

The parallel to the hypervisor layer with containers is the Docker daemon (assuming you're using Docker), it acts as the broker between the host OS and containers. It comes with less

computational overhead than hypervisor software (as depicted by the thinner box in the figure above), again making containers more lightweight compared to VMs.

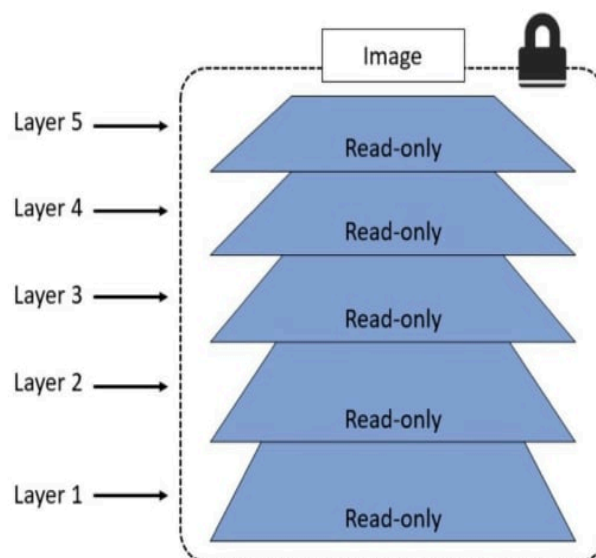
VMs suffer from duplication: many of the capabilities and features of the guest OS(s) are found in the host OS, so why not just use the host OS? This is what containers aim to do, whilst still providing isolation and decoupling from software in the host machine. With containers, only the things that the app absolutely needs are copied into the container, as opposed to VMs where the whole OS is installed – even the things from the OS that aren't used by the app.

## Docker File :

Docker builds images automatically by reading the instructions from a Dockerfile. It is a text file without any .txt extensions that contains all commands in order, needed to build a given image. It is always named **Dockerfile**.



Docker image consists of read-only layers each of which represents a Dockerfile instruction. The layers are stacked and each one is created by the change from the previous layer. For example, if I create a base layer of ubuntu and then in second instruction I install Python it will create a second layer. Likewise, if I do any changes by the instructions(RUN , COPY , ADD) it will create a new layer in that image. Containers are read-write layers that are created by docker images.



In simple words, a **Dockerfile** is a set of instructions that creates a stacked-layer for each instruction that collectively makes an image(which is a prototype or template for containers)

A Basic Docker file would look like :

```
FROM ubuntu:18.04
COPY . /app
RUN make /app
CMD python /app/app.py
```

Lets Learn about some basic dockerFile Commands :

- **FROM** - Defines a base image, it can be pulled from docker hub (for example- if we want to create a javascript application with node as backend then we need to have node as a base image, so it can run node application.).The **FROM** instruction specifies the base image that the container will be built on top of. This instruction is typically the first one in a Dockerfile and is used to set the base image for the container
- **RUN** - Executes command in a new image layer( we can have multiple run commands )
- **CMD** - Command to be executed when running a container( It is asked to have one CMD command, If a Dockerfile has multiple CMDs, it only applies the instructions from the last one.

```
CMD ["executable","param1","param2",...]
```

For example,

```
CMD ["npm", "start"]
```

This instruction tells Docker to run the command **npm start** when a container is created from the image. This command will start the Node.js application that was installed in the container using the **npm install** command.

- **ENV** - Use the **ENV** instruction to set environment variables inside the image which will be available during build time as well as in a running container. For example, to set the **NODE\_ENV** environment variable to **production**, you

would use the following command:

```
ENV NODE_ENV production
```

- **COPY** - It is used to copy your local files/directories to Docker Container. It is used to copy all files and directories from the current directory on the host machine to the current directory within the container.
- **ADD** - It is more feature-rich version of the COPY instruction. COPY is preferred over ADD. Major difference b/w ADD and COPY is that ADD allows you to copy from URL that is the source can be URL but in COPY it can only have local ones.
- **ENTRYPOINT** - Define a container's executable (You cannot override and ENTRYPOINT when starting a container unless you add the --entrypoint flag.)
- **VOLUME** - It defines which directory in an image should be treated as a volume. The volume will be given a random name which can be found using docker inspect command.
- **WORKDIR** - Defines the working directory for subsequent instructions in the Dockerfile(Important point to remember that it doesn't create a new intermediate layer in Image)  
This means that any commands that are run in the container will be executed relative to the specified directory.
- Use the **EXPOSE** command to tell Docker which ports the container will listen on at runtime. For example, if your application listens on port 9000, you would use the following command:

```
EXPOSE 9000
```

## **.dockerignore :**

If some files should be prevented from being copied into the Docker image(it can be sensitive informations like .env files which contains API keys or any other files that are not much important), a .dockerignore file can be added at the same level as the Dockerfile where files that should not be copied over into the Docker image can be specified. By this if we are using a COPY or ADD instruction in a Dockerfile to specify the files to be added into a Docker image, any file specified in the .dockerignore file will be ignored and not added into the Docker image.

## Difference between RUN,CMD and ENTRYPOINT?

**RUN** - RUN instruction allows you to install your application and packages required for it. It executes any commands on top of the current image and creates a new layer by committing the results. Often you will find multiple RUN instructions in a Dockerfile.

**RUN apt-get install python**

**CMD** - CMD instruction allows you to set a default command, which will be executed only when you run container without specifying a command. If Docker container runs with a command, the default command will be ignored. If Dockerfile has more than one CMD instruction, all but last CMD instructions are ignored.

**CMD "echo" "Hello World!"**

**ENTRYPOINT** - ENTRYPOINT instruction allows you to configure a container that will run as an executable. It looks similar to CMD, because it also allows you to specify a command with parameters. The difference is ENTRYPOINT command and parameters are not ignored when Docker container runs with command line parameters.

## BUILD YOUR FIRST DOCKER IMAGE :)

### 1 . Let's build a Hello World Docker Image With Python.

(Make sure to have Dockerfile in same directory as hello.py )

Dockerfile :

```
FROM python:3
WORKDIR /usr/src/app
COPY hello.py ./
CMD ["python" , "hello.py"]
```

hello.py :



```
print("MY FIRST DOCKER IMAGE")
```

To build a image from this dockerFile ,

```
docker build -t helloWorld
```

To list all the images ,

```
docker images
```

To run a image ,

```
Docker run helloWorld
```

After running the run command , we can see "MY FIRST DOCKER IMAGE".

## 2.Run a Simple Nginx Server :

(Make sure to have Dockerfile in same directory as index.html)

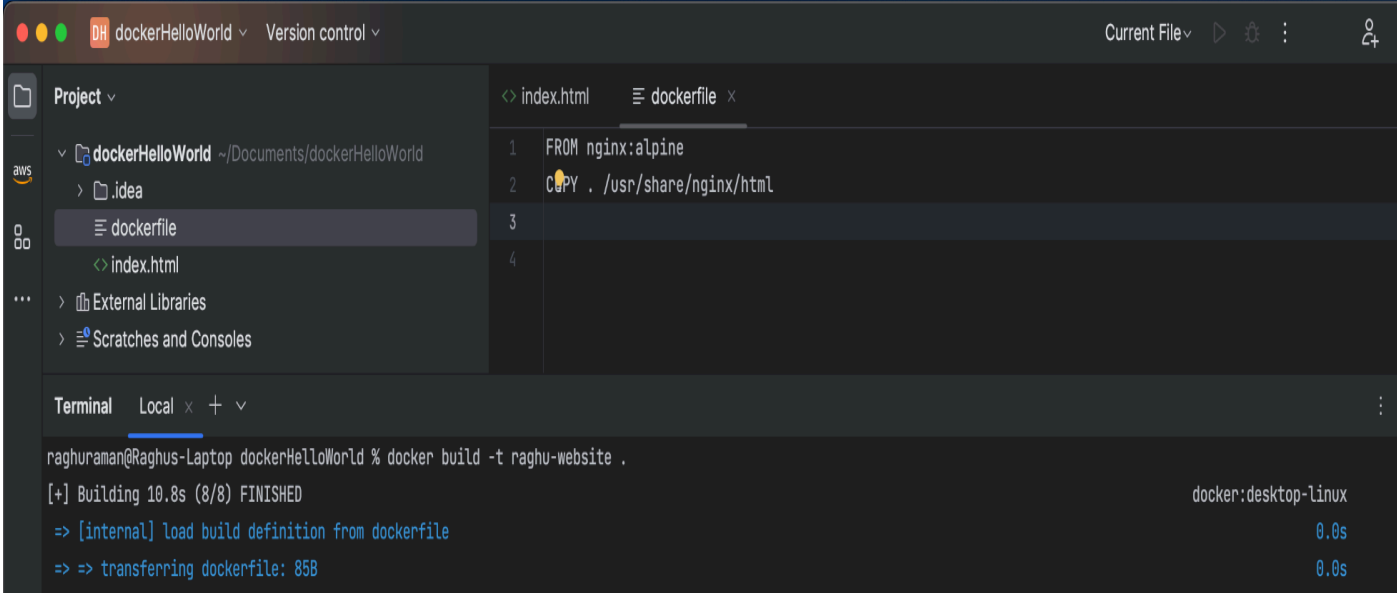
Dockerfile :

```
FROM nginx:alpine
COPY . /usr/share/nginx/html
```

Index.html :

```
<h1> HELLO , I'm RAGHU </h1>
```

Lets build Docker Image :



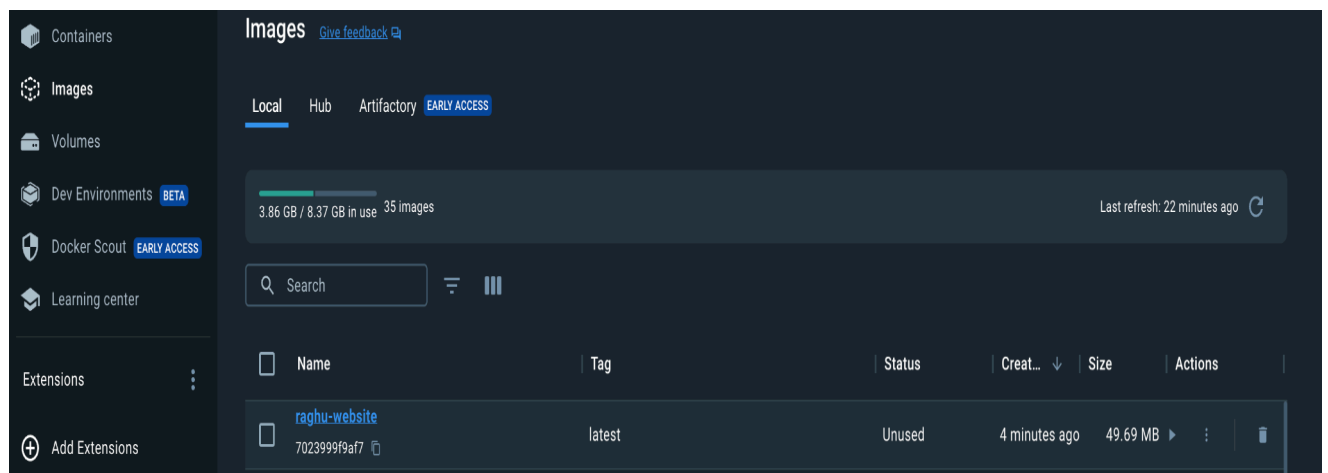
The screenshot shows the Visual Studio Code interface. The Project Explorer on the left shows the file structure of the 'dockerHelloWorld' project, including 'dockerfile' and 'index.html'. The Editor window displays the 'dockerfile' with the following content:

```
1 FROM nginx:alpine
2 COPY . /usr/share/nginx/html
3
4
```

The Terminal window at the bottom shows the command 'docker build -t raghu-website .' being executed. The output indicates that the build was successful, taking 10.8 seconds, and the image was transferred successfully.

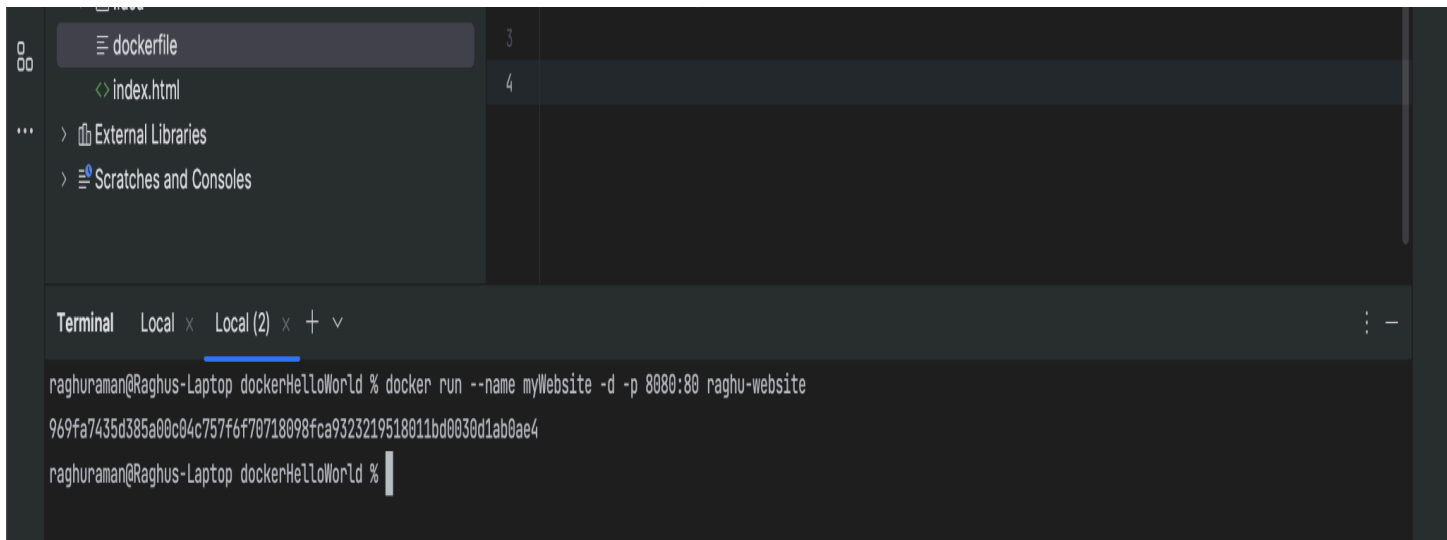
```
raghuraman@Raghus-Laptop dockerHelloWorld % docker build -t raghu-website .
[+] Building 10.8s (8/8) FINISHED
=> [internal] load build definition from dockerfile
=> => transferring dockerfile: 85B
```

In docker we can see the image we've just built :



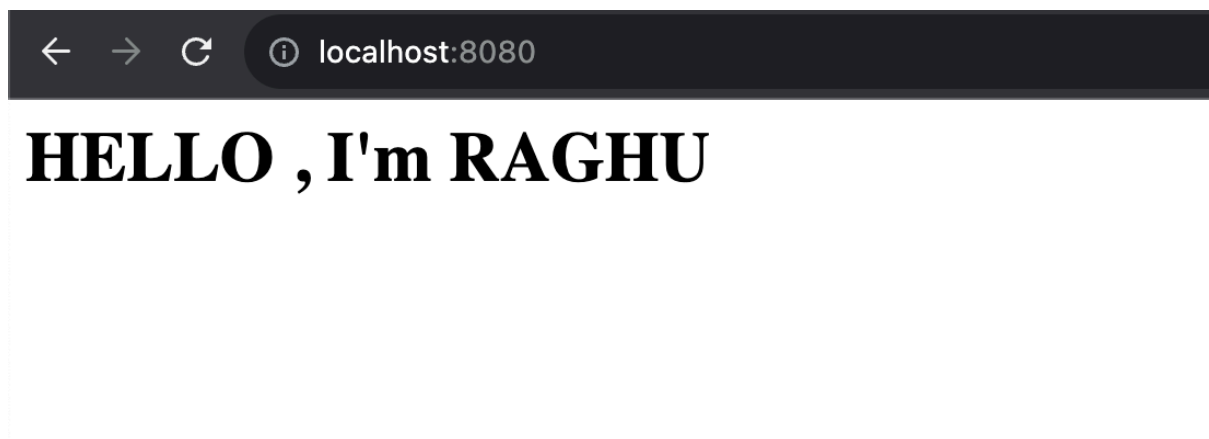
Lets run the Image in port 8080 ,

**docker run --name myWebsite -d -p 8080:80 raghu-website**



```
raghuraman@Raghus-Laptop dockerHelloWorld % docker run --name myWebsite -d -p 8080:80 raghu-website
969fa7435d385a00c04c757f6f70718098fca9323219518011bd0030d1ab0ae4
raghuraman@Raghus-Laptop dockerHelloWorld %
```

And when we hit , <http://localhost:8080/> ,



Try the same with Node and java images .

**Let's Learn how to  
run a image inside  
a Kubernetes  
Cluster with  
MiniKube in  
upcoming  
Chapters.**