**Docker:**

Docker is a computer program that performs operating system level virtualization also known as containerization.

You can deploy and run distributed applications without launching a entire virtual machine for each application.

**Why Docker:**

Docker makes it really easy to install run software without worrying and setup or dependencies.

**How Docker is working:**

Namespacing: Isolate the resources per process.

Control groups or cgroups: Limit the amount used per process.

Docker technology uses the Linux kernel and features of the kernel, like Cgroups and namespaces, to segregate processes so that they can run independently.

This independence is the intention of containers the ability to run multiple processes and application separately from one to another to make better use of your infrastructure.

Cgroups allow you to allocate resources from the underlying host machine — such as CPU time, system memory, network bandwidth, or combinations of these resources — among user-defined groups of tasks(processes) running on a system.

**Docker daemon:**

Docker Daemon is a server which interacts with the operating system and performs all kind of services.

Docker Daemon listens and receive REST API request and performs the operation after communicated with docker components and services.

Docker daemon creating and managing the docker images, containers, networks and volumes.

**Docker Volumes:**

Docker Volumes are the best way to store the persistent data that your application consume and create.

**Docker Registry:**

A Docker Registry is the remote location where Docker Images are stored. You push images to a registry and pull images from a registry. You can host your own registry or use a provider’s registry.

**Docker HUB:**

Docker Hub is the largest registry of Docker images. It’s also the default registry. You can find images and store your own images on Docker Hub for free.

**Docker Repository:**

A Docker Repository is a collection of Docker images with the same name and different tags. The tag is the identifier for the image.

**Docker Engine:**

Docker Engine or Docker is the base engine installed on your host machine to build and run containers using Docker components and services.

**Docker Machine:**

Docker Machine is a tool that lets you install Docker Engine on virtual hosts.

Docker Machine enables you to provision multiple remote Docker hosts on various flavours of Linux.

Docker Machine is a tool for provisioning and managing your Dockerized hosts (hosts with Docker Engine on them)

Docker Machine has its own command line client docker-machine and the Docker Engine client, docker. You can use Machine to install Docker Engine on one or more virtual systems. These virtual systems can be local (as when you use Machine to install and run Docker Engine in Virtual Box on Mac or Windows) or remote (as when you use Machine to provision Dockerized hosts on cloud providers). The Dockerized hosts themselves can be thought of, and are sometimes referred to as, managed “machines”.

**Docker Container:**

A Docker container is a live running instance of a Docker image.

Docker Container is a portable executable package which includes applications and their dependencies.

**Docker Image:**

A Docker Image is a file of instructions which is used to create Containers.

**How are Docker container created?**

Docker File creates a Docker Image using the build command

A Docker Image contains all the project’s code.

Using Docker Image, any user can run the code in order to create Docker Containers

Once a Docker Image is built, it’s uploaded in a registry or a Docker Hub from the Docker Hub, users can get the Docker Image and build new containers.

**Docker File:**

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. Using docker build users can create an automated build that executes several command-line instructions in succession.

**Usage**

The [docker build](https://docs.docker.com/engine/reference/commandline/build/) command builds an image from a Dockerfile and a *context*. The build’s context is the set of files at a specified location PATH or URL. The PATH is a directory on your local filesystem. The URL is a Git repository location.

A context is processed recursively. So, a PATH includes any subdirectories and the URL includes the repository and its submodules.

The build is run by the Docker daemon, not by the CLI. The first thing a build process does is send the entire context (recursively) to the daemon. In most cases, it’s best to start with an empty directory as context and keep your Dockerfile in that directory. Add only the files needed for building the Dockerfile.

**Warning**: Do not use your root directory, /, as the PATH as it causes the build to transfer the entire contents of your hard drive to the Docker daemon.

**Docker Build Command**

Traditionally, the Dockerfile is called Dockerfile and located in the root of the context. You use the -f flag with docker build to point to a Dockerfile anywhere in your file system.

Docker Build F Flag

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

**Docker Tag command**

You can specify a repository and tag at which to save the new image if the build succeeds:

$ docker build -t shykes/myapp .

To tag the image into multiple repositories after the build, add multiple -t parameters when you run the build command:

$ docker build -t shykes/myapp:1.0.2 -t shykes/myapp:latest .

Note that each instruction is run independently, and causes a new image to be created - so RUN cd /tmp will not have any effect on the next instructions.

Whenever possible, Docker will re-use the intermediate images (cache), to accelerate the docker build process significantly. This is indicated by the Using cache message in the console output.

**Format of the Dockerfile**

**COMMENT**

Docker treats lines that *begin* with # as a comment, unless the line is a valid [parser directive](https://docs.docker.com/engine/reference/builder/#parser-directives). A # marker anywhere else in a line is treated as an argument. This allows statements like:

**Example:**

# Comment

RUN echo 'we are running some # of cool things'

Line continuation characters are not supported in comments.

**FROM**

A Dockerfile **must start with a `FROM` instruction**. The FROM instruction specifies the [Base Image](https://docs.docker.com/engine/reference/glossary/#base-image) from which you are building.

**Syntax:**

FROM <image> [AS <name>]

Or

FROM <image>[:<tag>] [AS <name>]

Or

FROM <image>[@<digest>] [AS <name>]

ARG is the only instruction that may precede FROM in the Dockerfile.

### Understand how ARG and FROM interact

FROM instructions support variables that are declared by any ARG instructions that occur before the first FROM.

ARG CODE\_VERSION=latest

FROM base:${CODE\_VERSION}

CMD /code/run-app

FROM extras:${CODE\_VERSION}

CMD /code/run-extras

An ARG declared before a FROM is outside of a build stage, so it can’t be used in any instruction after a FROM. To use the default value of an ARG declared before the first FROM use an ARG instruction without a value inside of a build stage:

**Example:**

ARG VERSION=latest

FROM busybox:$VERSION

ARG VERSION

RUN echo $VERSION > image\_version

## MAINTAINER (deprecated)

Optional, it contains the name of the maintainer of the image.

MAINTAINER <name>

The MAINTAINER instruction sets the *Author* field of the generated images. The LABEL instruction is a much more flexible version of this and you should use it instead, as it enables setting any metadata you require, and can be viewed easily, for example with docker inspect. To set a label corresponding to the MAINTAINER field you could use:

LABEL maintainer="SvenDowideit@home.org.au"

## LABEL

LABEL <key>=<value> <key>=<value> <key>=<value> ...

The LABEL instruction adds metadata to an image. A LABEL is a key-value pair. To include spaces within a LABEL value, use quotes and backslashes as you would in command-line parsing. A few usage examples:

LABEL "com.example.vendor"="ACME Incorporated"

LABEL com.example.label-with-value="foo"

LABEL version="1.0"

LABEL description="This text illustrates \

that label-values can span multiple lines."

**RUN**

The RUN instruction will execute any commands in a new layer on top of the current image and commit the results. The resulting committed image will be used for the next step in the Dockerfile.

RUN has two forms;

RUN <command> (called shell form)

RUN ["executable", "param1", "param2"] called exec form.

**RUN** exec form makes it possible to avoid shell string munging, and to RUN commands using a base image that does not contain the specified shell executable.

**Note**:

1. To use a different shell, other than ‘/bin/sh’, use the *exec* form passing in the desired shell. For example, RUN ["/bin/bash", "-c", "echo hello"]
2. The *exec* form is parsed as a JSON array, which means that you must use double-quotes (“) around words not single-quotes (‘).

**CMD**

Used for executing commands when we build a new container from the docker image.

CMD instruction has three forms:

* CMD ["executable","param1","param2"] (*exec* form, this is the preferred form)
* CMD ["param1","param2"] (as *default parameters to ENTRYPOINT*)
* CMD command param1 param2 (*shell* form)

There can only be one CMD instruction in a Dockerfile. If you list more than one CMD then only the last CMD will take effect.

If you use the shell form of the CMD, then the <command> will execute in /bin/sh –c

FROM ubuntu

CMD echo "This is a test." | wc -

If you want to **run your** <command> **without a shell** then you must express the command as a JSON array and give the full path to the executable. **This array form is the preferred format of CMD.** Any additional parameters must be individually expressed as strings in the array:

FROM ubuntu

CMD ["/usr/bin/wc","--help"]

**Note**: Don’t confuse RUN with CMD. RUN actually runs a command and commits the result; CMD does not execute anything at build time, but specifies the intended command for the image.

**ENTRY POINT**

Entrypoint sets the command and parameters that will be executed first when a container is run.

Any command line arguments passed to docker run <image> will be appended to the entrypoint command, and will override all elements specified using CMD. For example, docker run <image> bash will add the command argument bash to the end of the entrypoint.

#### The exec syntax

The **exec** form is where you specify commands and arguments as a JSON array. This means you need to use double quotes rather than single quotes.

ENTRYPOINT ["executable", "param1", "param2"]

Using this syntax, Docker will not use a command shell, which means that normal shell processing does not happen. If you need shell processing features, then you can start the JSON array with the shell command.

ENTRYPOINT [ "sh", "-c", "echo $HOME" ]

#### Using an entrypoint script

Another option is to use a script to run entrypoint commands for the container. By convention, it often includes **entrypoint** in the name. In this script, you can setup the app as well as load any configuration and environment variables. Here is an example of how you can run it in a Dockerfile with the ENTRYPOINT **exec** syntax.

COPY ./docker-entrypoint.sh /  
ENTRYPOINT ["/docker-entrypoint.sh"]  
CMD ["postgres"]

#### Overriding Entrypoint

You can override entrypoint instructions using the [docker run --entrypoint](https://docs.docker.com/engine/reference/run/#entrypoint-default-command-to-execute-at-runtime) or [docker-compose run --entrypoint](https://docs.docker.com/compose/reference/run/) flags.

**EXPOSE**

Get your ports right.

EXPOSE assumes TCP. You can also specify UDP:

EXPOSE 80/udp

To expose on both TCP and UDP, include two lines:

EXPOSE 80/tcp

EXPOSE 80/udp

In this case, if you use -P with docker run, the port will be exposed once for TCP and once for UDP. Remember that -P uses an ephemeral high-ordered host port on the host, so the port will not be the same for TCP and UDP.

Regardless of the EXPOSE settings, you can override them at runtime by using the -p flag. For example

docker run -p 80:80/tcp -p 80:80/udp ...

**ENV** - well, setting environment variables is pretty important.

ENV <key> <value>

ENV <key>=<value> ...

The ENV instruction sets the environment variable <key> to the value <value>. This value will be in the environment for all subsequent instructions in the build stage and can be [replaced inline](https://docs.docker.com/engine/reference/builder/#environment-replacement) in many as well.

The ENV instruction has two forms. The first form, ENV <key> <value>, will set a single variable to a value. The entire string after the first space will be treated as the <value> - including whitespace characters. The value will be interpreted for other environment variables, so quote characters will be removed if they are not escaped.

The second form, ENV <key>=<value> ..., allows for multiple variables to be set at one time. Notice that the second form uses the equals sign (=) in the syntax, while the first form does not. Like command line parsing, quotes and backslashes can be used to include spaces within values.

For example:

ENV myName="John Doe" myDog=Rex\ The\ Dog \

myCat=fluffy

and

ENV myName John Doe

ENV myDog Rex The Dog

ENV myCat fluffy

will yield the same net results in the final image.

The environment variables set using ENV will persist when a container is run from the resulting image. You can view the values using docker inspect, and change them using docker run --env <key>=<value>.

**Note**: Environment persistence can cause unexpected side effects. For example, setting ENV DEBIAN\_FRONTEND noninteractive may confuse apt-get users on a Debian-based image. To set a value for a single command, use RUN <key>=<value> <command>.

**COPY vs ADD** - these two are often confused, so I’ll explain the difference.

COPY and ADD are both Dockerfile instructions that serve similar purposes. They let you copy files from a specific location into a Docker image.

COPY takes in a src and destination. It only lets you copy in a local file or directory from your host (the machine building the Docker image) into the Docker image itself.

ADD lets you do that too, but it also supports 2 other sources. First, you can use a URL instead of a local file / directory. Secondly, you can extract a tar file from the source directly into the destination.

A valid use case for ADD is when you want to extract a local tar file into a specific directory in your Docker image. This is exactly what the [Alpine image](https://github.com/gliderlabs/docker-alpine/blob/c7368b846ee805b286d9034a39e0bbf40bc079b3/versions/library-3.5/Dockerfile) does with ADD rootfs.tar.gz /.

If you’re copying in local files to your Docker image, always use COPY because it’s more explicit.

## ADD

ADD has two forms:

* ADD [--chown=<user>:<group>] <src>... <dest>
* ADD [--chown=<user>:<group>] ["<src>",... "<dest>"] (this form is required for paths containing whitespace)

## COPY

COPY has two forms:

* COPY [--chown=<user>:<group>] <src>... <dest>
* COPY [--chown=<user>:<group>] ["<src>",... "<dest>"] (this form is required for paths containing whitespace)

**VOLUME** - another source of confusion, what’s the difference between Dockerfile VOLUME and container volumes?

You can use the VOLUME instruction in a Dockerfile to tell Docker that the stuff you store in that specific directory should be stored on the host file system not in the container file system. This implies that stuff stored in the volume will persist and be available also after you destroy the container

**USER** - when root is too mainstream.

**WORKDIR** - set the working directory.

**ONBUILD** - give more flexibility to your team and clients.

**Docker Compose:**

Docker Compose is used for running multiple containers as a single service

Here, containers run in isolation but can interact with each other

All Docker Compose files are YAML files.

In Docker Compose, a user can start all the services (containers) using a single command

**For example:**

If you have an application which requires NGINX server and Redis database, you can create a Docker Compose file which can run both the containers as a service without the need to start each one separately.

**Docker Commands:**

1. Check installed docker version
   1. **#docker --version**
2. Check docker information
   1. **#docker info**
3. Pull latest docker image
   1. **#docker pull centos:latest**
4. List the docker container and images
   1. **#docker images**
5. List out all running docker containers in the host.
   1. **#docker ps -a**
6. Create a container using create and run command
   1. **#docker create image\_name(centos)**
   2. O/p: d2b3acdb168dd17ecb852e9e380714fe884c0dc3ca6fcbf60a791f812705eef7
7. Run docker run command
   1. docker run is combination of process i.e it is first creating a container from the image and then starting the container.
   2. **docker run = docker create image + docker start container**
8. Check the docker container's CPU, Memory usage, Network I/O and Block I/O.
   1. **#docker stats container**
9. Stop and Start the docker container.
   1. **#docker start [Options] Container ID**
10. Stop and Stop the docker container.
    1. **#docker stop [Options] Container ID**
       * 1. (OR)
    2. **#docker kill container\_id**
11. Enter into the container
    1. **#docker exec -it container\_id bash**
12. Save the changes in container
    1. **#docker commit**