[**Getting started | Docker Docs**](https://docs.docker.com/get-started/)

[**https://github.com/wsargent/docker-cheat-sheet#why-docker**](https://github.com/wsargent/docker-cheat-sheet#why-docker)

**Docker basics:**

**Docker:**

Docker is an open platform for developing, shipping, and running applications. Docker enables you to separate your applications from your infrastructure so you can deliver software quickly.

Docker provides the ability to package and run an application in a loosely isolated environment called a container. The isolation and security lets you run many containers simultaneously on a given host. Containers are lightweight and contain everything needed to run the application, so you don't need to rely on what's installed on the host. You can share containers while you work, and be sure that everyone you share with gets the same container that works in the same way.

**Container:**

Runtime instance of a Docker Image.

A way to package applications with all the necessary dependencies and configuration.

* Layers of images
* Mostly Linux Base Image, because small in size

**Image:**

An image is a read-only template with instructions for creating a Docker container.

The actual package where the code, configuration, dependencies, libraries are included .

Artifact, that can be move around

**Difference between Docker and Virtual Machine:**

The OS consists of three layers: Hardware, OS Kernel and Application.

Docker virtualizes the application layer. It uses host kernel

VM virtualizes the Kernel and application layer. It uses its own kernel

Here's a simple analogy: think of a Docker image as a recipe, and a Docker container as a dish prepared using that recipe.

Example:

Let's say you want to run a web application built with Node.js. You can create a Docker image that includes the Node.js runtime, your application code, and any necessary dependencies. Then, you can spin up multiple Docker containers from that image, each running an instance of your web application.

1. Create a Dockerfile (a recipe for building Docker images) for your Node.js application:
2. Build a Docker image from the Dockerfile:
3. Run a Docker container from the image:

**Dockerfile:**

<https://medium.com/@anshita.bhasin/a-step-by-step-guide-to-create-dockerfile-9e3744d38d11>

**Docker Volume**

1. docker volume create <volume-name>
2. docker run -dp 127.0.0.1:3000:3000 --mount type=volume,src=todo-db,target=/etc/todos getting-started
3. docker volume inspect todo-db

**Docker commands:**

Docker pull <image-name>

Docker run -d -p –name –network -e <image-name>

Docker build -t <image-name> .

Docker images

Docker ps

Docker stop <container-name>

Docker rm <container-name>

Docker rmi <image-name>

Docker volume create <volume-name>

Docker network create <network-name>

## **Key Docker concepts**

* **Images**

A Docker image is a read-only template with instructions for creating a Docker container. Images are built from Dockerfiles and contain a set of predefined layers that make up an image. Images can be shared via Docker registries like Docker Hub.

* **Containers**

A Docker container is a runtime instance of an image. Containers run the application in an isolated environment and have their own filesystem, CPU, memory, process space, and network interfaces. However, they share the host OS kernel, so they are more lightweight than VMs. Containers can be started, stopped, committed (converted into images), and deleted.

* **Dockerfile**

A [Dockerfile](https://staragile.com/blog/what-is-a-dockerfile) is a text document with instructions for building a Docker image. It contains a set of commands and arguments that Docker follows to auto-generate a Docker image. Using a Dockerfile, you can create an image containing your application and all its dependencies.

* **Docker Hub**

Docker Hub is a SaaS service provided by Docker for sharing and managing Docker images. It is a public Docker registry with thousands of images that can be downloaded and used locally. You can also push your own images to Docker Hub to share with others.

* **Docker Registry**

A Docker registry stores Docker images. Docker Hub is Docker's public registry service, but you can also set up private registries to store and share images internally within your organization.

* **Docker Compose**

Docker Compose is a tool for defining and running multi-container Docker applications. With Docker Compose, you use a YAML file to configure your application's services, and then you can start all the services with a single command. Docker Compose is helpful for development environments where you want to combine multiple services.

* **Docker Engine**

The Docker Engine powers the Docker platform. It consists of a Docker daemon, a CLI, and a REST API. The Docker daemon is what actually builds, runs, and manages Docker containers. The Docker CLI and REST API allow you to interact with the Docker daemon.

**ECS**

You must architect your applications so that they can run on containers. A container is a standardized unit of software development that holds everything that your software application requires to run. This includes relevant code, runtime, system tools, and system libraries. Containers are created from a read-only template that's called an image. Images are typically built from a Dockerfile. A Dockerfile is a plaintext file that contains the instructions for building a container. After they're built, these images are stored in a registry such as Amazon ECR where they can be downloaded from.

After you create and store your image, you create an Amazon ECS task definition. A task definition is a blueprint for your application. It is a text file in JSON format that describes the parameters and one or more containers that form your application. For example, you can use it to specify the image and parameters for the operating system, which containers to use, which ports to open for your application, and what data volumes to use with the containers in the task. The specific parameters available for your task definition depend on the needs of your specific application.

After you define your task definition, you deploy it as either a service or a task on your cluster. A cluster is a logical grouping of tasks or services that runs on the capacity infrastructure that is registered to a cluster.

A task is the instantiation of a task definition within a cluster. You can run a standalone task, or you can run a task as part of a service. You can use an Amazon ECS service to run and maintain your desired number of tasks simultaneously in an Amazon ECS cluster. How it works is that, if any of your tasks fail or stop for any reason, the Amazon ECS service scheduler launches another instance based on your task definition. It does this to replace it and thereby maintain your desired number of tasks in the service.

**Dockerfile Syntax:**

### **1. FROM**

A FROM statement defines which image to download and start from. It must be the first command in your Dockerfile. A Dockerfile can have multiple FROM statements which means the Dockerfile produces more than one image.

Example:

FROM java: 8

### **2. MAINTAINER**

This statement is a kind of documentation, which defines the author who is creating this Dockerfile or who should you contact if it has bugs.

Example:

MAINTAINER Firstname Lastname <example@geeksforgeeks.com>

### **3. RUN**

The RUN statement defines running a command through the shell, waiting for it to finish, and saving the result. It tells what process will be running inside the container at the run time.

Example:

RUN unzip install.zip /opt/install

RUN echo hello

### **4. ADD**

If we define to add some files, ADD statement is used. It basically gives instructions to copy new files, directories, or remote file URLs and then adds them to the filesystem of the image.  
To sum up it can add local files, contents of tar archives as well as URLs.

Example:

Local Files: ADD run.sh /run.sh

Tar Archives: ADD project.tar.gz /install/

URLs: ADD https://project.example-gfg.com/downloads/1.0/testingproject.rpm/test

### **5. ENV**

ENV statement sets the environment variables both during the build and when running the result. It can be used in the Dockerfile and any scripts it calls. It can be used in the Dockerfile as well as any scripts that the Dockerfile calls. These are also persistent with the container and can be referred to at any moment.

Example:

ENV URL\_POST=production.example-gfg.com

### **6. ENTRYPOINT**

It specifies the starting of the expression to use when starting your container. Simply ENTRYPOINT specifies the start of the command to run. If your container acts as a command-line program, you can use ENTRYPOINT.

Example:

ENTRYPOINT ["/start.sh"]

### **7. CMD**

CMD specifies the whole command to run. We can say CMD is the default argument passed into the ENTRYPOINT. The main purpose of the CMD command is to launch the software required in a container.

Example:

CMD ["program-foreground"]

CMD ["executable", "program1", "program2"]

Note: If you have both ENVIRONMENT and CMD, they are combined together.

### **8. EXPOSE**

EXPOSE statement maps a port into the container. The ports can be TCP or UDP but by default, it is TCP.

Example:

EXPOSE 3030

### **9. VOLUME**

The VOLUME statement defines shared volumes or ephemeral volumes depending upon whether you have one or two arguments.

Example:

1. If you have two arguments, it maps a host path into a container path.

VOLUME ["/host/path" "/container/path/"]

2. If you have one arguments, it creates a volume that can be inherited by the later containers.

VOLUME ["/shared-data"]

### **10. WORKDIR**

As the name suggests, WORKDIR sets the directory that the container starts in. Its main purpose is to set the working directory for all future Dockerfile commands.

Example:

WORKDIR /directory-name

### **11. USER**

It sets which user’s container will run as. This can be useful if you have shared network directories involved that assume a fixed username or a fixed user number.

Example:

USER geeksforgeeks

USER 4000

### **12. ARG**

A variable that can be provided at build time is defined by an ARG Instruction. Once it has been specified in the Dockerfile, you can specify it using the –build-arg switch when creating the image. The Dockerfile supports multiple ARG instructions. The only instruction in the Dockerfile that can come before the FROM instruction is ARG.

After the image is created, ARG values are not accessible. An ARG variable value won’t be accessible to a running container.

### Example

ARG image\_name=latest

FROM centos:$image\_name

docker build -t <image-name>:<tag> --build-arg image\_name=centos8 .

**Docker Troubleshooting Commands:**

1. **Check container logs**

Docker ps -a

Docker logs <container-id>/ <container-name>

docker attach $CONTAINER

1. **Logs of container in real-time**

Docker logs -f <container-id>

1. **Inspect a container**

Docker inspect <container-id>

1. **View container resource usage**

Docker stats

1. **Debugging inside a container**

Docker exec -it <container-id> /bin/bash

1. **Check running processes within the container**

Docker top <container-id>

**How to automate containers in case of failure?**

Docker provides [restart policies](https://docs.docker.com/engine/reference/run/#restart-policies---restart) to control whether your containers start automatically when they exit, or when Docker restarts.

1. **no**

Don't automatically restart the container. (Default)

**docker run -d --restart no redis**

1. **on-failure[:max-retries]**

Restart the container if it exits due to an error, which manifests as a non-zero exit code. Optionally, limit the number of times the Docker daemon attempts to restart the container using the :max-retries option.

**docker run -d --restart on-failure:5 redis**

1. **always**

Always restart the container if it stops. If it's manually stopped, it's restarted only when Docker daemon restarts or the container itself is manually restarted.

**docker run -d --restart always redis**

1. **unless-stopped**

Similar to always, except that when the container is stopped (manually or otherwise), it isn't restarted even after Docker daemon restarts.

**docker run -d --restart unless-stopped redis**

**How to adjust resources in docker for intense workloads?**

We can add runtime constraints to resources.

**Dynamic Resource Allocation:**

Use Docker's update commands to dynamically adjust resources for running containers based on the current workload.

**docker update --cpus="2.0" --memory="2G" <container\_id>**

**How to optimize docker images?**

1. **Minimize the number of layers:**

Minimize the number of levels in your [Dockerfile](https://www.geeksforgeeks.org/what-is-dockerfile/) by combining instructions into a single RUN directive for related instructions.

Example:

FROM base\_image

RUN apt-get update && \

apt-get install -y package1 package2 && \

apt-get clean

1. **Use Minimal Base Image:**

For reducing the size and resource consumption of [Docker containers](https://www.geeksforgeeks.org/virtualisation-with-docker-containers/), use Alpine Linux as a light base image

Example:

FROM node:14-alpine

1. **Use Multi-stage build:**

Multi-stage builds enable you to create smaller images by using multiple FROM instructions and copying only the necessary files between stages.

Example:

# Build stage

FROM node:14-alpine as build

WORKDIR /app

COPY package\*.json ./

RUN npm install

COPY . .

RUN npm run build

# Production stage

FROM node:14-alpine as production

WORKDIR /app

COPY --from=build /app/package\*.json ./

RUN npm ci --production

COPY --from=build /app/dist ./dist

CMD ["npm", "start"]

1. **Use dockerignore**

Example .dockerignore:

.git

node\_modules

\*.log

1. **Remove unnecessary files**

In a multi-stage build, copy only the necessary files from the build stage to the production stage:

COPY --from=build /app/dist ./dist

1. **Compress artifacts**

Before adding the build artifacts in your Docker image, reduce their size using technologies like gzip, tar, or zip.

Example:

FROM base\_image AS builder

# Build your application

FROM base\_image

COPY --from=builder /app /app

RUN tar -czf /app.tar.gz /app

1. **Leverage cache builds**

Docker caches intermediate layers to speed up the build process. Place the most stable instructions (least likely to change) at the top to take advantage of this caching mechanism.

Example:

# Set base image

FROM ubuntu:20.04 AS builder

# Install build dependencies (least likely to change)

RUN apt-get update && apt-get install -y \

build-essential \

&& rm -rf /var/lib/apt/lists/\*

# Copy only necessary build files (potentially changing)

WORKDIR /app

COPY . .

# Build the application (most likely to change)

RUN make

# Final stage for production image

FROM alpine:latest

# Copy built application from the builder stage

COPY --from=builder /app/app /app

# Set entry point

ENTRYPOINT ["/app"]

**Dockerize the app and deploy it on EC2:**

#!/bin/bash

sudo su

yum update -y

yum install docker -y

service docker start

systemctl enable docker

usermod -a -G docker ec2-user

docker pull nehapatil108/api-flask-app:latest

docker run -d -p 80:80 --restart always nehapatil108/api-flask-app:latest

**Stress commands:**

sudo yum update -y

sudo amazon-linux-extras install epel -y

sudo amazon-linux-extras install epel -y

sudo yum install stress -y

# Stress the CPU with 4 workers for 60 seconds

stress --cpu 4 --timeout 60