Docker is a centralized platform for packaging, deploying, and running applications. Before Docker, many users faced the problem that a particular code is running in the developer's system but not in the user's system. So, the main reason to develop docker is to help developers to develop applications easily, ship them into containers, and can be deployed anywhere.

## What is Docker?

Docker is an **open-source centralized platform designed** to create, deploy, and run applications. Docker uses a container on the host's operating system to run applications. It allows applications to use the same **Linux kernel** as a system on the host computer, rather than creating a whole virtual operating system. Containers ensure that our application works in any environment like development, test, or production.

Docker includes components such as **Docker client, Docker server, Docker machine, Docker hub, Docker compose,** etc.

### Docker Containers

Docker containers are the **lightweight** alternatives of the virtual machine. It allows developers to package up the application with all its libraries and dependencies, and ship it as a single package. The advantage of using a docker container is that you don't need to allocate any RAM and disk space for the applications. It automatically generates storage and space according to the application requirement.

### Virtual Machine

A virtual machine is a software that allows us to install and use other operating systems (Windows, Linux, and Debian) simultaneously on our machine. The operating system in which virtual machine runs are called virtualized operating systems. These virtualized operating systems can run programs and perform tasks that we perform in a real operating system.

### Containers Vs. Virtual Machine

| **Containers** | **Virtual Machine** |
| --- | --- |
| Integration in a container is faster and cheap. | Integration in virtual is slow and costly. |
| No wastage of memory. | Wastage of memory. |
| It uses the same kernel, but different distribution. | It uses multiple independent operating systems. |

## Why Docker?

Docker is designed to benefit both the Developer and System Administrator. There are the following reasons to use Docker -

* Docker allows us to easily install and run software without worrying about setup or dependencies.
* Developers use Docker to eliminate machine problems, i.e. "**but the code is working on my laptop**." when working on code together with co-workers.
* Operators use Docker to run and manage apps in isolated containers for better compute density.
* Enterprises use Docker to securely build agile software delivery pipelines to ship new application features faster and more securely.
* Since docker is not only used for the deployment, but it is also a great platform for development, that's why we can efficiently increase our customer's satisfaction.

## Advantages of Docker

* It runs the container in seconds instead of minutes.
* It uses less memory.
* It provides lightweight virtualization.
* It does not require a full operating system to run applications.
* It uses application dependencies to reduce the risk.
* Docker allows you to use a remote repository to share your container with others.
* It provides a continuous deployment and testing environment.

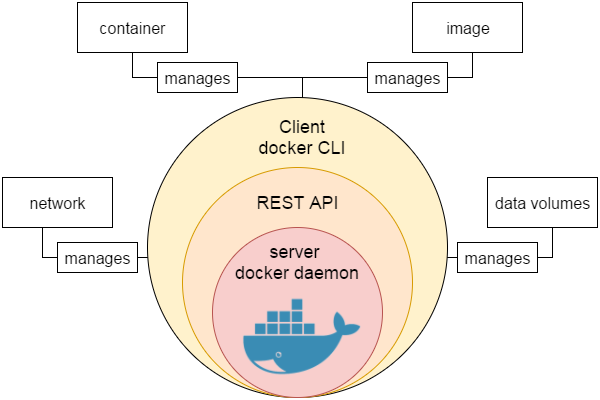
## Disadvantages of Docker

* It increases complexity due to an additional layer.
* In Docker, it is difficult to manage large amounts of containers.
* Some features such as container self -registration, container self-inspects, copying files from host to the container, and more are missing in Docker.
* Docker is not a good solution for applications that require a rich graphical interface.
* Docker provides cross-platform compatibility means if an application is designed to run in a Docker container on Windows, then it can't run on Linux or vice versa.

## Docker Engine

It is a client server application that contains the following major components.

* A server which is a type of long-running program called a daemon process.
* The REST API is used to specify interfaces that programs can use to talk to the daemon and instruct it what to do.
* A command line interface client.



# Docker Features

Although Docker provides lots of features, we are listing some major features:-

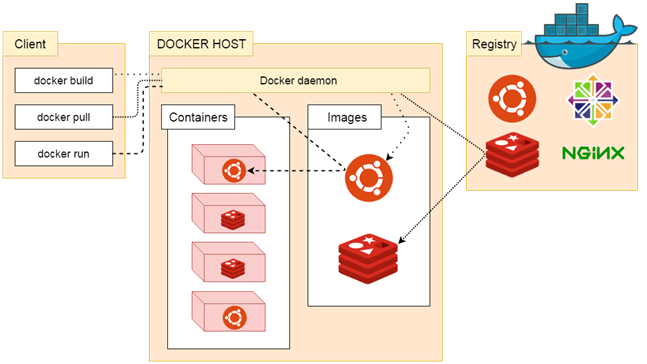
* **Easy and Faster Configuration:** We can deploy our code in less time and effort. As Docker can be used in a wide variety of environments, the requirements of the infrastructure are no longer linked with the environment of the application.
* **Increase productivity:** By easing technical configuration and rapid deployment of applications. Docker not only helps to execute the application in an isolated environment but also it has reduced the resources.
* **Application Isolation:** It provides containers that are used to run applications in isolation environments. Each container is independent of another and allows us to execute any kind of application.
* **Swarm:** It is a clustering and scheduling tool for Docker containers. Swarm uses the Docker API as its front end, which helps us to use various tools to control it. It also helps us to control a cluster of Docker hosts as a single virtual host. It's a self-organizing group of engines that is used to enable pluggable backends.
* **Routing Mesh:** It routes the incoming requests for published ports on available nodes to an active container. This feature enables the connection even if there is no task running on the node.
* **Services**: Services is a list of tasks that lets us specify the state of the container inside a cluster. Each task represents one instance of a container that should be running and Swarm schedules them across nodes.
* **Security Management:** It allows us to save secrets into the swarm itself and then choose to give services access to certain secrets.

### What is Docker daemon?

Docker daemon runs on the host operating system. It is responsible for running containers to manage docker services. Docker daemon communicates with other daemons. It offers various Docker objects such as images, containers, networking, and storage.

### Docker architecture

Docker follows Client-Server architecture, which includes the three main components that are **Docker Client**, **Docker Host**, and **Docker Registry**.



### 1. Docker Client

Docker client uses **commands** and **REST APIs** to communicate with the Docker Daemon (Server). When a client runs any docker command on the docker client terminal, the client terminal sends these docker commands to the Docker daemon. Docker daemon receives these commands from the docker client in the form of command and REST API's request.

#### Note: Docker Client has an ability to communicate with more than one docker daemon.

Docker Client uses Command Line Interface (CLI) to run the following commands -

docker build

docker pull

docker run

### 2. Docker Host

Docker Host is used to provide an environment to execute and run applications. It contains the docker daemon, images, containers, networks, and storage.

### 3. Docker Registry

Docker Registry manages and stores the Docker images.

There are two types of registries in the Docker -

**Public Registry -** Public Registry is also called Docker **hub**.

**Private Registry -** It is used to share images within the enterprise.

## Docker Objects: There are the following Docker Objects -

### Docker Images

Docker images are the **read-only binary templates** used to create Docker Containers. It uses a private container registry to share container images within the enterprise and also uses a public container registry to share container images within the whole world. Metadata is also used by docket images to describe the container's abilities.

### Docker Containers

Containers are the structural units of Docker, which is used to hold the entire package that is needed to run the application. The advantage of containers is that it requires very less resources.

In other words, we can say that the image is a template, and the container is a copy of that template.

### Docker Networking

Using Docker Networking, an isolated package can be communicated. Docker contains the following network drivers -

* **Bridge -** Bridge is a default network driver for the container. It is used when multiple docker communicates with the same docker host.
* **Host -** It is used when we don't need network isolation between the container and the host.
* **None -** It disables all the networking.
* **Overlay -** Overlay offers Swarm services to communicate with each other. It enables containers to run on the different docker hosts.
* **Macvlan -** Macvlan is used when we want to assign MAC addresses to the containers.

### Docker Storage

Docker Storage is used to store data on the container. Docker offers the following options for the Storage -

* **Data Volume -** Data Volume provides the ability to create persistence storage. It also allows us to name volumes, list volumes, and containers associated with the volumes.
* **Directory Mounts -** It is one of the best options for docker storage. It mounts a host's directory into a container.
* **Storage Plugins -** It provides an ability to connect to external storage platforms.

## What is the Docker Platform?

Docker gives the ability to package and execute an application in a loosely separated environment known as a container. This separation and security permit us to execute several containers simultaneously on the specified host. The containers are lightweight and include everything required to execute the applications, so we don't need to depend on what is installed on the host currently. We can easily distribute containers while we work and ensure that everyone we distribute with receives a similar container that operates in a similar way.

Docker gives a platform and tooling facility to maintain the lifecycle of our containers:

* The container becomes a unit to distribute and test our application.
* Develop our application and its supporting elements using containers.
* When we are ready, deploy our application into our production environment as an orchestrated service or a container. It works similarly whether our production environment is a cloud provider, a local data center, or a combination of the two.

## Usage of Docker

* **Fast application delivery**

Docker accumulates the development lifecycle by permitting developers to operate in standardized environments with local containers, which give our services & applications. Containers are ideal for continuous delivery & continuous integration workflows.

* **Responsive scaling and deployment**

The container-based platform of Docker permits for highly compact workloads. The containers can execute on the local laptop, cloud providers, virtual or physical machines in the data center, or in a combination of the environments of a developer.

The lightweight nature and portability of Docker also make it easier to dynamically maintain workloads and scale up and tear down services and applications as business requirements dictate.

* **Running multiple workloads on a similar hardware**

Docker is fast and lightweight. It offers a cost-effective and viable replacement for hypervisor-based virtual machines, so we can use more of our server capacity to gain our business objectives. Docker is great for high-density platforms and for medium and small deployments where we require to work more using fewer resources.

# Docker Container and Image

Docker container is a running instance of an image. You can use Command Line Interface (CLI) commands to run, start, stop, move, or delete a container. You can also provide configuration for the network and environment variables. Docker container is an isolated and secure application platform, but it can share and access resources running in a different host or container.

An image is a read-only template with instructions for creating a Docker container. A docker image is described in a text file called a **Dockerfile**, which has a simple, well-defined syntax. An image does not have states and never changes. Docker Engine provides the core Docker technology that enables images and containers.

You can understand the container and image with the help of the following command.

1. $ docker run hello-world

The above command **docker run hello-world** has three parts.

1) **docker:** It is a docker engine and used to run docker programs. It tells the operating system that you are running a docker program.

2) **run:** This subcommand is used to create and run a docker container.

3) **hello-world:** It is the name of an image. You need to specify the name of an image which is to load into the container.

# Docker Dockerfile

A Dockerfile is a text document that contains commands that are used to assemble an image. We can use any command that call on the command line. Docker builds images automatically by reading the instructions from the Dockerfile.

The docker build command is used to build an image from the Dockerfile. You can use the -f flag with docker build to point to a Dockerfile anywhere in your file system.

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

## Dockerfile Instructions

The instructions are not case-sensitive but you must follow conventions which recommend to use uppercase.

Docker runs instructions of Dockerfile in top to bottom order. The first instruction must be **FROM** in order to specify the Base Image.

A statement begin with # treated as a comment. You can use RUN, CMD, FROM, EXPOSE, ENV etc instructions in your Dockerfile.

Here, we are listing some commonly used instructions.

### FROM

This instruction is used to set the Base Image for the subsequent instructions. A valid Dockerfile must have FROM as its first instruction.

Ex. FROM ubuntu

### LABEL

We can add labels to an image to organize images of our project. We need to use LABEL instructions to set labels for the image.

Ex. LABEL vendorl = "JavaTpoint"

### RUN

This instruction is used to execute any command of the current image.

Ex. RUN /bin/bash -c 'source $HOME/.bashrc; echo $HOME'

### CMD

This is used to execute applications by the image. We should use CMD always in the following form

1. CMD ["executable", "param1", "param2"?]

This is the preferred way to use CMD. There can be only one CMD in a Dockerfile. If we use more than one CMD, only the last one will execute.

### COPY

This instruction is used to copy new files or directories from source to the filesystem of the container at the destination.

Ex. COPY abc/ /xyz

**Rules**AD

* The source path must be inside the context of the build. We cannot COPY ../something /something because the first step of a docker build is to send the context directory (and subdirectories) to the docker daemon.
* If source is a directory, the entire contents of the directory are copied including file system metadata.

### WORKDIR

The WORKDIR is used to set the working directory for any RUN, CMD and COPY instruction that follows it in the Dockerfile. If the work directory does not exist, it will be created by default.

We can use WORKDIR multiple times in a Dockerfile.

Ex. WORKDIR /var/www/html

# Docker Java Application Example

Here, we are creating a Java application and running it by using docker. This example includes the following steps.

1. **Create a directory**

Directory is required to organize files. Create a directory by using the below command.

* 1. $ mkdir java-docker-app

1. **Create a Java File**

Now create a Java file. Save this file as **Hello.java** file.and Save it inside the directory **java-docker-app** as Hello.java.

**3. Create a Dockerfile**

After creating a Java file, we need to create a Dockerfile which contains instructions for the Docker. Dockerfile does not contain any file extension. So, save it simple with the Dockerfile name.  
**// Dockerfile**

* 1. FROM java:8
  2. COPY . /var/www/java
  3. WORKDIR /var/www/java
  4. RUN javac Hello.java
  5. CMD ["java", "Hello"]

Write all instructions in uppercase because it is convention. Put this file inside the **java-docker-app** directory. Now we have Dockerfile parallel to Hello.java inside the **java-docker-app** directory.

**4. Build Docker Image**

After creating Dockerfile, we are changing the working directory.

* 1. $ cd java-docker-app

Now, create an image by following the below command. we must login as root in order to create an image. In this example, we have switched to as a root user. In the following command, **java-app** is name of the image. We can have any name for our docker image.

* 1. $ docker build -t java-app .

After successfully building the image. Now, we can run our docker image.

**5. Run Docker Image**

After creating the image successfully. Now we can run docker by using run command. The following command is used to run java-app.

* 1. $ docker run java-app

Here, we can see that after running the java-app it produced an output.

# Docker Push Repository

We can push our Docker image to a global repository. It is a public repository provided by Docker officially. It allows us to put our docker image on the server. It is helpful when we want to access our docker image from global. Follow the following steps to push a custom image on the Docker hub.

1. **login to hub.docker.com**

We need to login to our Docker hub. If you don't have one, **create it first.**

* 1. $ docker login

1. **Tag Docker Image**

After login, we need to tag our docker image that we want to push. Following command is used to tag the docker image.

* 1. $ docker tag image-name username/image-name

**username** refers to our dockerid or the username which is used to login.  
**image-name** is the name of our docker image present on our system.

1. **Push Docker Image**

The following command is used to push the docker image to the docker hub repository.

* 1. $ docker push username/image-name

Now, login into our account at hub.docker.com and check our dashboard. It will have a new docker image named.

# Docker Useful Commands

### Check Docker version: $ docker version

### Build Docker Image from a Dockerfile

1. $ docker build -t image-name docker-file-location

**-t** : it is used to tag Docker images with the provided name.

### Run Docker Image: $ docker run -d image-name

**-d** : It is used to create a daemon process.

### Check available Docker images: $ docker images

### Check for latest running container: $ docker ps -l

**-l** : it is used to show the latest available container.

### Check all running containers: $ docker ps -a

**-a** : It is used to show all available containers.

### Stop running container**:** $ docker stop container\_id

### Delete an image: $ docker rmi image-name

### Delete all images: $ docker rmi $(docker images -q)

### Delete all images forcefully: $ docker rmi -r $(docker images -q)

**-r** : It is used to delete images forcefully.

### Delete all containers: $ docker rm $(docker ps -a -q)

### Enter into Docker container: $ docker exec -it container-id bash

# 

# Docker Cloud

Docker provides us the facility to store and fetch docker images on the cloud registry. We can store dockerized images either privately or publicly. It is a full GUI interface that allows us to manage builds, images, swarms, nodes and apps.

We need to have Docker ID to access and control images. If we don't have, create it first.

## Creating Repository

To create Docker cloud repository, click on the create repository +button available on the welcome page at the bottom.

After filling the details, we should make this repository public. Now, just click on the create button at the bottom. It will create a repository for us.

# 

# Docker Compose

It is a tool which is used to create and start a Docker application by using a single command. We can use it to configure our application's services.

It is a great tool for development, testing, and staging environments.

It provides the following commands for managing the whole lifecycle of our application.

* Start, stop and rebuild services
* View the status of running services
* Stream the log output of running services
* Run a one-off command on a service

**To implement compose, it consists of the following steps.**

1. Put Application environment variables inside the Dockerfile to access publicly.
2. Provide services name in the docker-compose.yml file so they can be run together in an isolated environment.
3. run docker-compose up and Compose will start and run your entire app.

A typical **docker-compose.yml** file has the following format and arguments.

**// docker-compose.yml**

1. version: '3'
2. services:
3. web:
4. build: .
5. ports:
6. - "5000:5000"
7. volumes:
8. - .:/code
9. - logvolume01:/var/log
10. links:
11. - redis
12. redis:
13. image: redis
14. volumes:
15. logvolume01: {}

## Installing Docker Compose

Following are the instructions to install Docker Compose in Linux Ubuntu.

curl -L <https://github.com/docker/compose/releases/download/1.12.0/docker-compose-> `uname -s`-`uname -m` **>** /usr/local/bin/docker-compose

If It says, permission denied. So, make the file executable.

1. $ sudo chmod +x /usr/local/bin/docker-compose

## Running Application using Docker Compose

Follow the following example

1) Create a Directory

1. $ mkdir docker-compose-example
2. $ cd docker-compose-example

2) Create a file **app.py.**

1. from flask import Flask
2. from redis import Redis
3. app = Flask(\_\_name\_\_)
4. redis = Redis(host='redis', port=6379)
5. @app.route('/')
6. def hello():
7. count = redis.incr('hits')
8. return 'Hello World! I have been seen {} times.\n'.format(count)
9. if \_\_name\_\_ == "\_\_main\_\_":
10. app.run(host="0.0.0.0", debug=True)

3) Create a file **requirements.txt.**

1. **flask**
2. **redis**

4) Create a Dockerfile.

1. FROM python:3.4-alpine
2. ADD . /code
3. WORKDIR /code
4. RUN pip install -r requirements.txt
5. CMD ["python", "app.py"]

5) Create a Compose File. **// docker-compose.yml**

1. version: '2'
2. services:
3. web:
4. build: .
5. ports:
6. - "5000:5000"
7. volumes:
8. - .:/code
9. redis:
10. image: "redis:alpine"

6) Build and Run Docker App with Compose

$ docker-compose up

Now, we can see the output by following the running http url.**(0.0.0.0:5000)**

Each time, when we refresh the page. It shows the counter incremented by 1.