

Docker Tutorial – Introduction To Docker

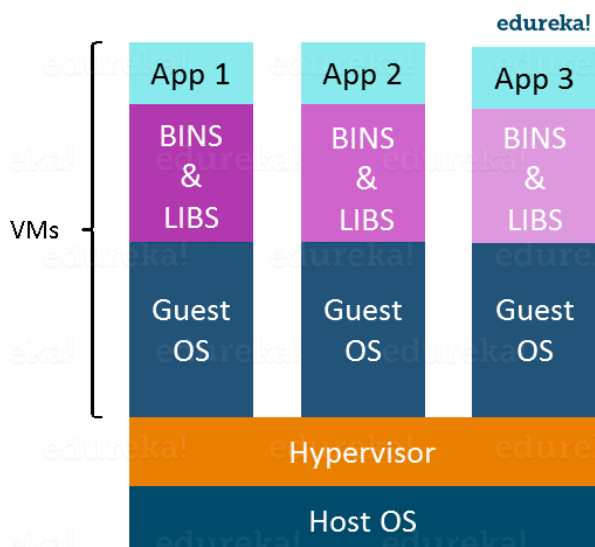
Docker Tutorial

Here you will be provided with both conceptual and practical knowledge of Docker, a cutting-edge containerization technology.

- Docker is gaining popularity and its usage is spreading like wildfire.
- The reason for Docker's growing popularity is the extent to which it can be used in an IT organization.
- Very few tools out there have the functionality to find itself useful to both developers and as well as system administrators.
- Docker is one such tool that truly lives up to its promise of **Build, Ship and Run**.
- In simple words, Docker is a software containerization platform, meaning you can build your application, package them along with their dependencies into a container and then these containers can be easily shipped to run on other machines.
- *For example:* Lets consider a linux based application which has been written both in Ruby and Python. This application requires a specific version of linux, Ruby and Python. In order to avoid any version conflicts on user's end, a linux docker container can be created with the required versions of Ruby and Python installed along with the application. Now the end users can use the application easily by running this container without worrying about the dependencies or any version conflicts.
- These containers uses Containerization which can be considered as an evolved version of Virtualization.
- The same task can also be achieved using Virtual Machines, however it is not very efficient.

Virtualization

- Virtualization is the technique of importing a Guest operating system on top of a Host operating system.
- This technique was a revelation at the beginning because it allowed developers to run multiple operating systems in different virtual machines all running on the same host.
- This eliminated the need for extra hardware resource. The advantages of Virtual Machines or Virtualization are:
 - Multiple operating systems can run on the same machine
 - Maintenance and Recovery were easy in case of failure conditions
 - Total cost of ownership was also less due to the reduced need for infrastructure



- In the diagram on the right, you can see there is a host operating system on which there are 3 guest operating systems running which is nothing but the virtual machines.

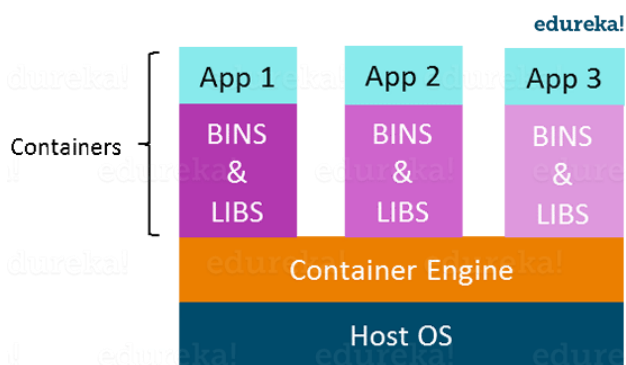
- As you know nothing is perfect, Virtualization also has some shortcomings. Running multiple Virtual Machines in the same host operating system leads to performance degradation. This is because of the guest OS running on top of the host OS, which will have its own kernel and set of libraries and dependencies. This takes up a large chunk of system resources, i.e. hard disk, processor and especially RAM.
- Another problem with Virtual Machines which uses virtualization is that it takes almost a minute to boot-up. This is very critical in case of real-time applications.
- Following are the disadvantages of Virtualization:
 - Running multiple Virtual Machines leads to unstable performance
 - Hypervisors are not as efficient as the host operating system
 - Boot up process is long and takes time
- These drawbacks led to the emergence of a new technique called Containerization. Now let me tell you about Containerization.

Containerization

Containerization is the technique of bringing virtualization to the operating system level. While Virtualization brings abstraction to the hardware, Containerization brings abstraction to the operating system. Do note that Containerization is also a type of Virtualization. Containerization is however more efficient because there is no guest OS here and utilizes a host's operating system, share relevant libraries & resources as and when needed unlike virtual machines. Application specific binaries and libraries of containers run on the host kernel, which makes processing and execution very fast. Even booting-up a container takes only a fraction of a second. Because all the containers share, host operating system and holds only the application related binaries & libraries. They are lightweight and faster than Virtual Machines.

Advantages of Containerization over Virtualization:

- Containers on the same OS kernel are lighter and smaller
- Better resource utilization compared to VMs
- Boot-up process is short and takes few seconds

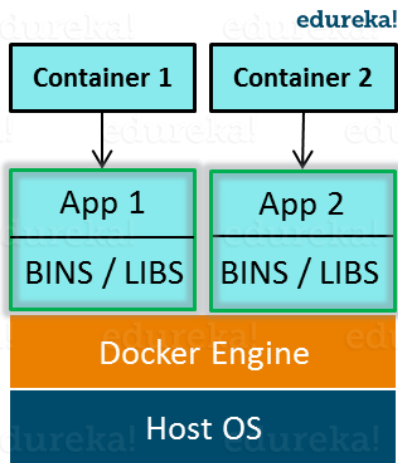


In the diagram on the right, you can see that there is a host operating system which is shared by all the containers. Containers only contain application specific libraries which are separate for each container and they are faster and do not waste any resources.

All these containers are handled by the containerization layer which is not native to the host operating system. Hence a software is needed, which can enable you to create & run containers on your host operating system.

Introduction To Docker

Docker is a containerization platform that packages your application and all its dependencies together in the form of Containers to ensure that your application works seamlessly in any environment.



As you can see in the diagram on the right, each application will run on a separate container and will have its own set of libraries and dependencies. This also ensures that there is process level isolation, meaning each application is independent of other applications, giving developers surety that they can build applications that will not interfere with one another.

As a developer, I can build a container which has different applications installed on it and give it to my QA team who will only need to run the container to replicate the developer environment.

Benefits of Docker

Now, the QA team need not install all the dependent software and applications to test the code and this helps them save lots of time and energy. This also ensures that the working environment is consistent across all the individuals involved in the process, starting from development to deployment. The number of systems can be scaled up easily and the code can be deployed on them effortlessly.

Features of Docker

- Docker has the ability to reduce the size of development by providing a smaller footprint of the operating system via containers.
- With containers, it becomes easier for teams across different units, such as development, QA and Operations to work seamlessly across applications.
- You can deploy Docker containers anywhere, on any physical and virtual machines and even on the cloud.
- Since Docker containers are pretty lightweight, they are very easily scalable.

Docker & Docker Container

Docker & it's need – Docker is a containerization platform that packages your application and all its dependencies together in the form of a docker container to ensure that your application works seamlessly in any environment.

Container– Docker Container is a standardized unit which can be created on the fly to deploy a particular application or environment. It could be an Ubuntu container, CentOS container, etc. to full-fill the requirement from an operating system point of view. Also, it could be an application oriented container like CakePHP container or a Tomcat-Ubuntu container etc.

Let's understand it with an example:

A company needs to develop a Java Application. In order to do so the developer will setup an environment with tomcat server installed in it. Once the application is developed, it needs to be tested by the tester. Now the tester will again set up tomcat environment from the scratch to test the application. Once the application testing is done, it will be deployed on the production server. Again the production needs an environment with tomcat

installed on it, so that it can host the Java application. If you see the same tomcat environment setup is done thrice. There are some issues that I have listed below with this approach:

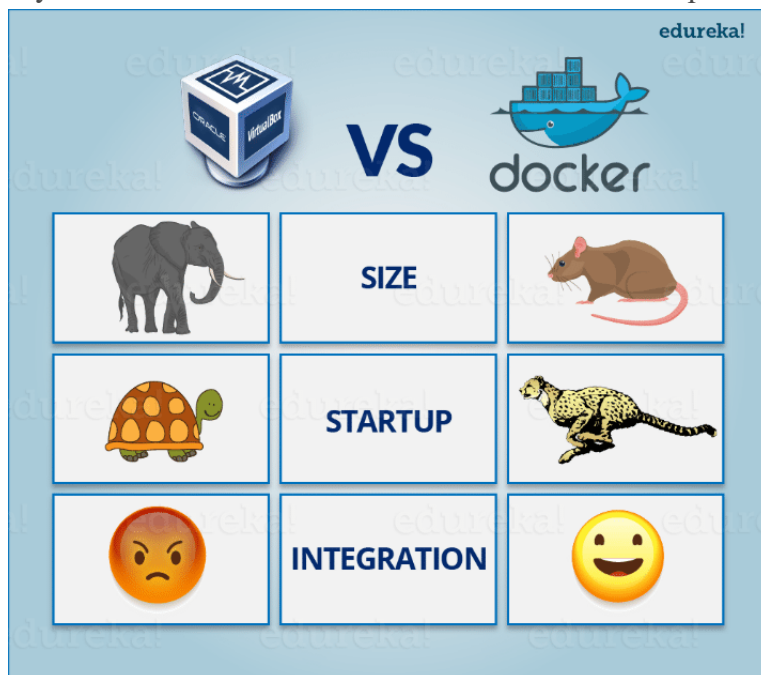
- 1) There is a loss of time and effort.
- 2) There could be a version mismatch in different setups i.e. the developer & tester may have installed tomcat 7, however the system admin installed tomcat 9 on the production server.

Now, I will show you how Docker container can be used to prevent this loss.

In this case, the developer will create a tomcat docker image (An Image is nothing but a blueprint to deploy multiple containers of the same configurations) using a base image like Ubuntu, which is already existing in Docker Hub (the Hub has some base images available for free) . Now this image can be used by the developer, the tester and the system admin to deploy the tomcat environment. This is how this container solves the problem.

I hope you are with me so far into the article . In case you have any further doubts, please feel to leave a comment, I will be glad to help you.

However, now you would think that this can be done using Virtual Machines as well. However, there is catch if you choose to use virtual machine. Let's see a comparison between the two to understand this better.



Let me take you through the above diagram. Virtual Machine and Docker Container are compared on the following three parameters:

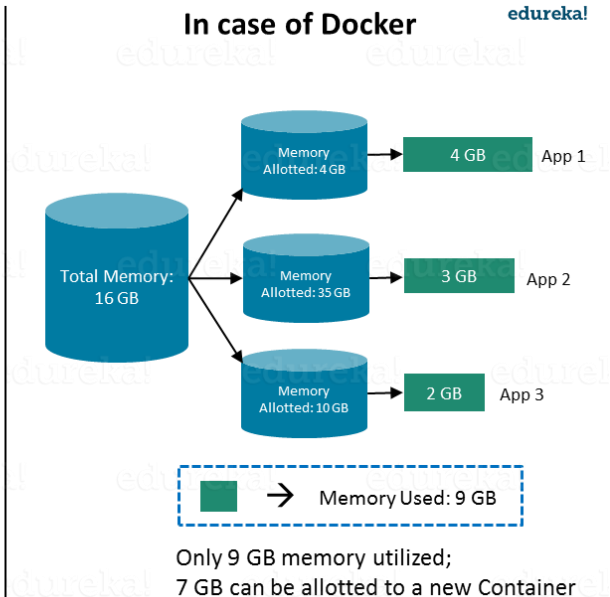
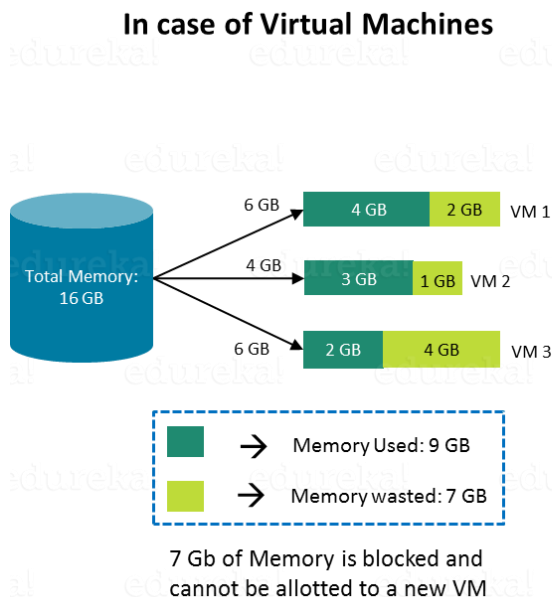
- Size – This parameter will compare Virtual Machine & Docker Container on their resource they utilize.
- Startup – This parameter will compare on the basis of their boot time.
- Integration – This parameter will compare on their ability to integrate with other tools with ease.

I will follow the above order in which parameters are listed. So first parameter would be “Size”.

Check out the following video to know more about Docker Containers & why it is used.

Size

The following image explains how Virtual Machine and Docker Container utilize the resources allocated to them.



Consider a situation depicted in the above image. I have a host system with 16 Gigabytes of RAM and I have to run 3 Virtual Machines on it. To run the Virtual Machines in parallel, I need to divide my RAM among the Virtual Machines. Suppose I allocate it in the following way:

- 6 GB of RAM to my first VM,
- 4 GB of RAM to my second VM, and
- 6 GB to my third VM.

In this case, I will not be left with anymore RAM even though the usage is:

- My first VM uses only **4 GB** of RAM – Allotted **6 GB** – **2 GB** Unused & Blocked
- My second VM uses only **3 GB** of RAM – Allotted **4 GB** – **1 GB** Unused & Blocked
- My third VM uses only **2 GB** of RAM – Allotted **6 GB** – **4 GB** Unused & Blocked

This is because once a chunk of memory is allocated to a Virtual Machine, then that memory is blocked and cannot be re-allocated. I will be wasting **7 GB (2 GB + 1 GB + 4 GB)** of RAM in total and thus cannot setup a new Virtual Machine. This is a major issue because RAM is a costly hardware.

So, how can I avoid this problem?

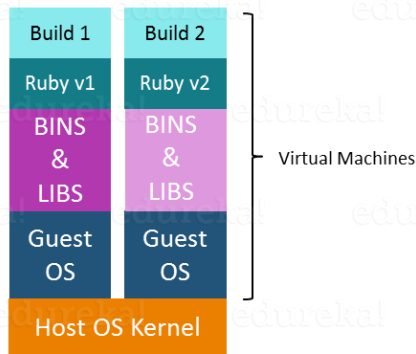
If I use Docker, my CPU will allocate exactly the amount of memory that is required by the Container.

- My first container will use only **4 GB** of RAM – Allotted **4 GB** – **0 GB** Unused & Blocked
- My second container will use only **3 GB** of RAM – Allotted **3 GB** – **0 GB** Unused & Blocked
- My third container will use only **2 GB** of RAM – Allotted **2 GB** – **0 GB** Unused & Blocked

Since there is no allocated memory (RAM) which is unused, I save **7 GB (16 – 4 – 3 – 2)** of RAM by using Docker Container. I can even create additional containers from the leftover RAM and increase my productivity.

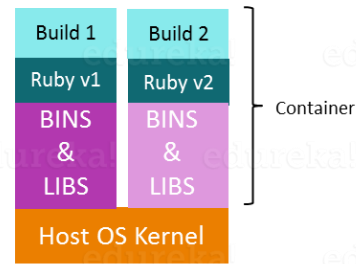
Start-Up

In case of Virtual Machines



New Builds → Multiple OS → Separate Libraries
→ Heavy → More Time

In case of Docker



New Builds → Same OS → Separate Libraries
→ Lightweight → Less Time

When it comes to start-up, Virtual Machine takes a lot of time to boot up because the guest operating system needs to start from scratch, which will then load all the binaries and libraries. This is time consuming and will prove very costly at times when quick startup of applications is needed. In case of Docker Container, since the container runs on your host OS, you can save precious boot-up time. This is a clear advantage over Virtual Machine.

Consider a situation where I want to install two different versions of Ruby on my system. If I use Virtual Machine, I will need to set up 2 different Virtual Machines to run the different versions. Each of these will have its own set of binaries and libraries while running on different guest operating systems. Whereas if I use Docker Container, even though I will be creating 2 different containers where each container will have its own set of binaries and libraries, I will be running them on my host operating system. Running them straight on my Host operating system makes my Docker Containers lightweight and faster.

Virtualization vs Containerization

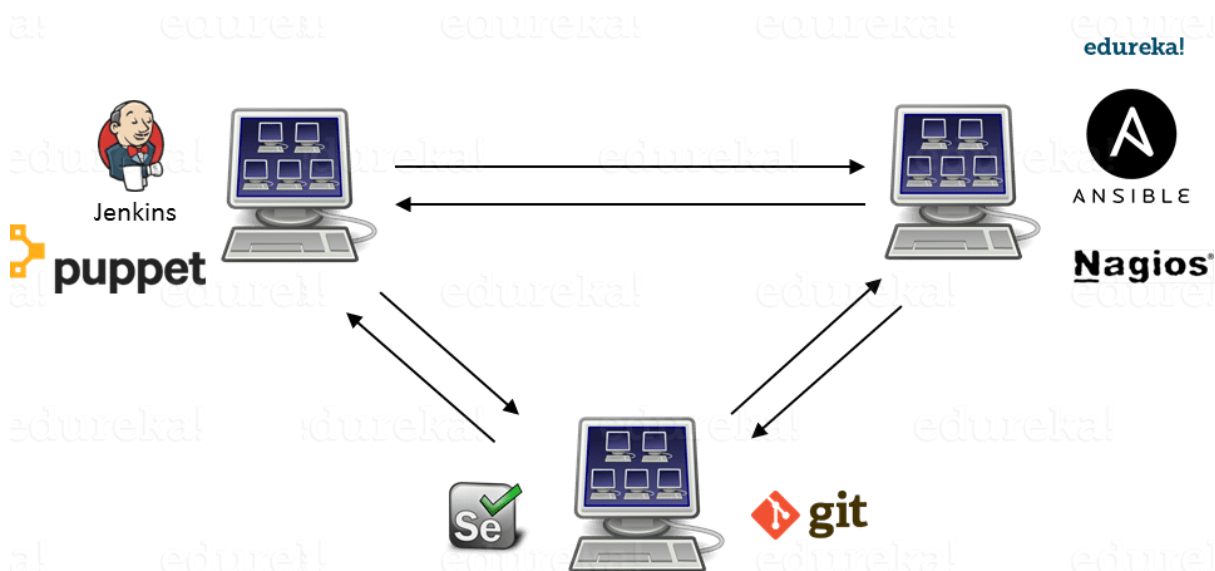
Virtualization and Containerization both let you run multiple operating systems inside a host machine.

Virtualization deals with creating many operating systems in a single host machine. Containerization on the other hand will create multiple containers for every type of application as required.

As we can see from the above image, the major difference is that there are multiple Guest Operating Systems in Virtualization which are absent in Containerization. The best part of Containerization is that it is very lightweight as compared to the heavy virtualization. So Docker Container clearly wins again from Virtual Machine based on Startup parameter.

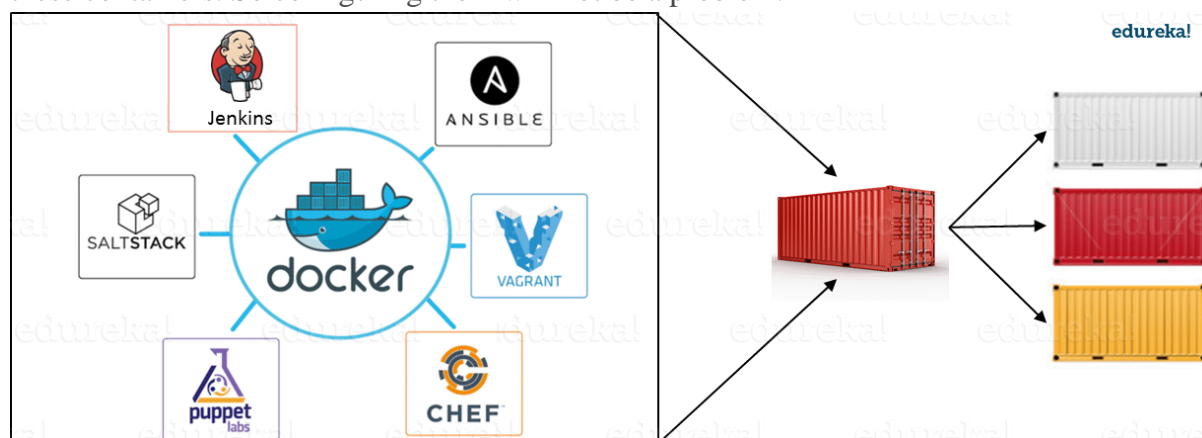
Integration of Tools

Integration of different tools using Virtual Machine maybe possible, but even that possibility comes with a lot of complications.



I can have only a limited number of DevOps tools running in a Virtual Machine. As you can see in the image above, If I want many instances of Jenkins and Puppet, then I would need to spin up many Virtual Machines because each can have only one running instance of these tools. Setting up each VM brings with it, infrastructure problems. I will have the same problem if I decide to setup multiple instances of Ansible, Nagios, Selenium and Git. It will also be a hectic task to configure these tools in every VM.

This is where Docker comes to the rescue. Using Docker Container, we can set up many instances of Jenkins, Puppet, and many more, all running in the same container or running in different containers which can interact with one another by just running a few commands. I can also easily scale up by creating multiple copies of these containers. So configuring them will not be a problem.



To sum up, it won't be an understatement to say that this is a more sensible option when compared to Virtual Machines.

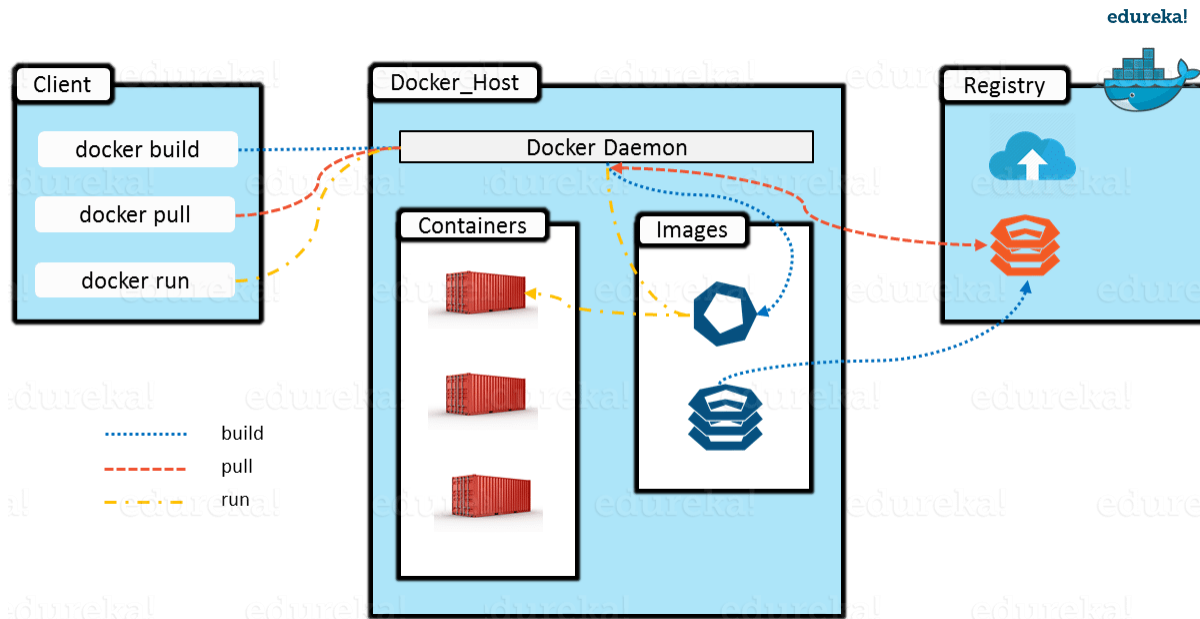
The former is designed to benefit both Developers and System Administrators, making it a part of many DevOps toolchains. Developers can write their code without worrying about the testing or the production environment and system administrators need not worry about infrastructure as Docker can easily scale up and scale down the number of systems for deploying on the servers.

Docker Architecture:

It consists of a Docker Engine which is a client-server application with three major components:

1. A server which is a type of long-running program called a daemon process (the docker command).
2. A REST API which specifies interfaces that programs can use to talk to the daemon and instruct it what to do.
3. A command line interface (CLI) client (the docker command).
4. The CLI uses the Docker REST API to control or interact with the Docker daemon through scripting or direct CLI commands. Many other Docker applications use the underlying API and CLI.

Docker Architecture includes a Docker client – used to trigger Docker commands, a Docker Host – running the Docker Daemon and a Docker Registry – storing Docker Images. The Docker Daemon running within Docker Host is responsible for the images and containers.



- To build a Docker Image, we can use the CLI (client) to issue a build command to the Docker Daemon (running on Docker_Host). The Daemon will then build an image based on our inputs and save it in the Registry, which can be either Docker hub or a local repository
- If we do not want to create an image, then we can just pull an image from the Docker hub, which would have been built by a different user
- Finally, if we have to create a running instance of my Docker image, we can issue a run command from the CLI, which will create a Container.

Docker Compose:

Docker Compose is basically used to run multiple Docker Containers as a single server. Let me give you an example:

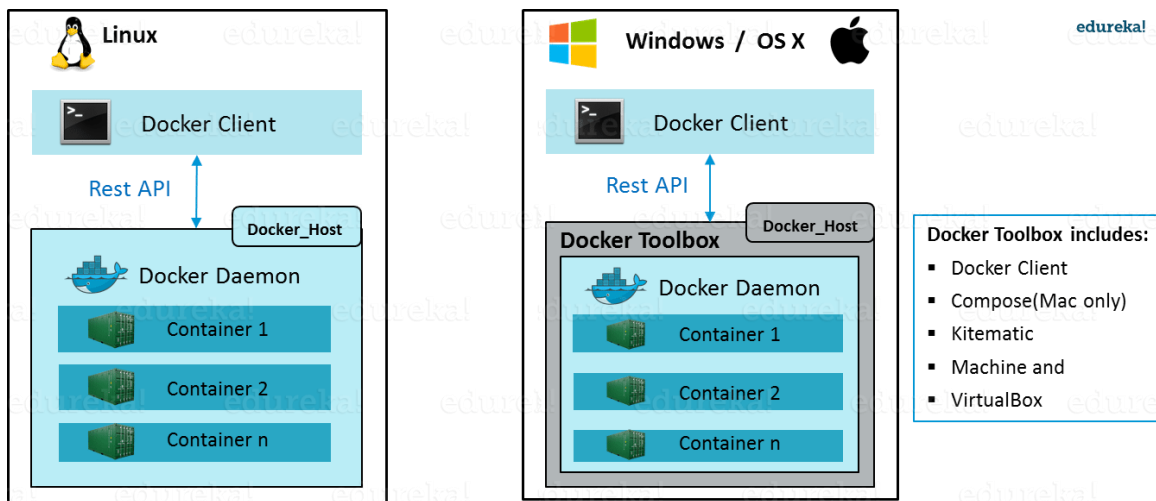
Suppose if I have an application which requires WordPress, Maria DB and PHP MyAdmin. I can create one file which would start both the containers as a service without the need to start each one separately. It is really useful especially if you have a microservice architecture.

Docker Engine

Now I will take you through Docker Engine which is the heart of the system.

Docker Engine is simply the application that is installed on your host machine. It works like a client-server application which uses:

- A **server** which is a type of long-running program called a daemon process
- A command line interface (CLI) **client**
- REST API is used for communication between the CLI client and Docker Daemon



As per the above image, in a Linux Operating system, there is a client which can be accessed from the terminal and a Host which runs the Daemon. We build our images and run containers by passing commands from the CLI client to the Daemon.

However, in case of Windows/Mac there is an additional Toolbox component inside the Docker host. This Docker Toolbox is an installer to quickly and easily install and setup a Docker environment on your Windows/iOS. This Toolbox installs Docker Client, Machine, Compose (Mac only), Kitematic and VirtualBox.

Let's now understand these important terms, i.e. **Docker File**, **Docker Images**, **Docker Containers** and **Docker Registry**.

Dockerfile, Images & Containers

The three most important aspects that you must know before you get your feet wet with Docker are:

1. Dockerfile
2. Docker Images
3. Docker Containers



In the above diagram you can see that when a Dockerfile is built, it gives you a Docker Image. Furthermore, when you execute the Docker Image then it finally gives you a Docker Container.

Let's now understand each of these in detail.

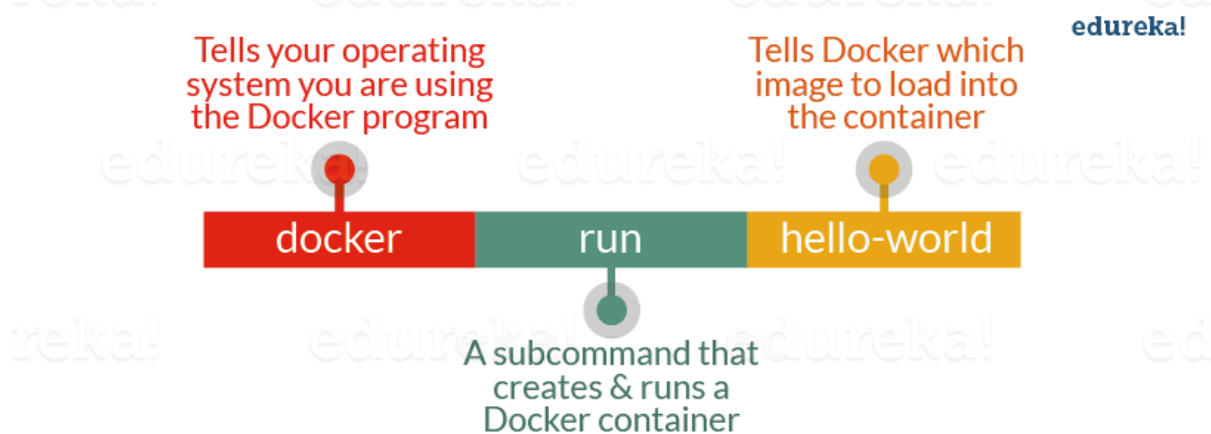
Dockerfile

A Docker Image is created by the sequence of commands written in a file called as Dockerfile. When this Dockerfile is executed using a docker command it results in a Docker Image with a name. When this Image is executed by "docker run" command it will by itself start whatever application or service it must start on its execution.

Docker Image

A Docker Image can be considered something similar to a template which is typically used to build Docker Containers. These Docker Images are created using the build command. These Read only templates are used for creating containers by using the run command. In other words, these read-only templates are nothing but the building blocks of a Docker Container. In order to execute an image and build a container you need to use the following docker commands.

The Docker Images that you create using this command are stored within the Docker Registry. It can be either a user's local repository or a public repository like a Docker Hub which allows multiple users to collaborate in building an application.

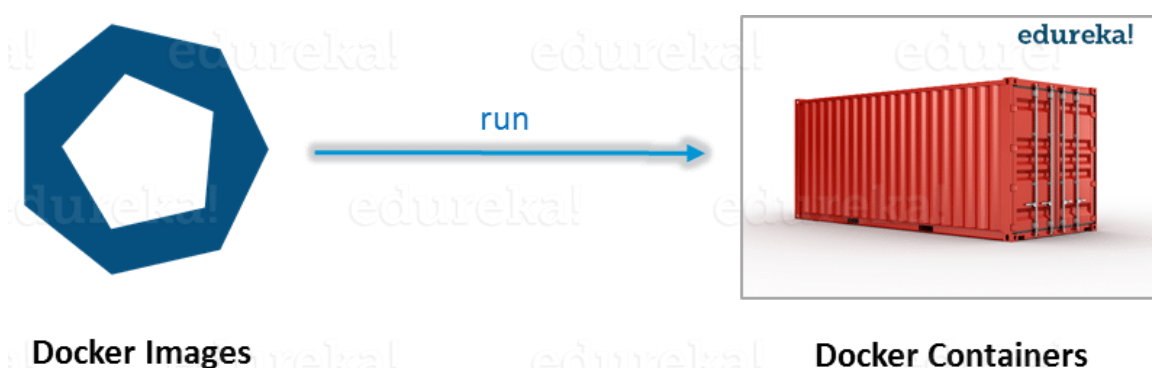


Docker lets people (or companies) create and share software through Docker images. Also, you don't have to worry about whether your computer can run the software in a Docker image — a Docker container *can always run it*.

I can either use a ready-made docker image from docker-hub or create a new image as per my requirement. In the Docker Commands blog we will see how to create your own image.

Docker Container

Docker Containers are the ready applications created from Docker Images. Or you can say they are running instances of the Images and they hold the entire package needed to run the application. This happens to be the ultimate utility of the technology.

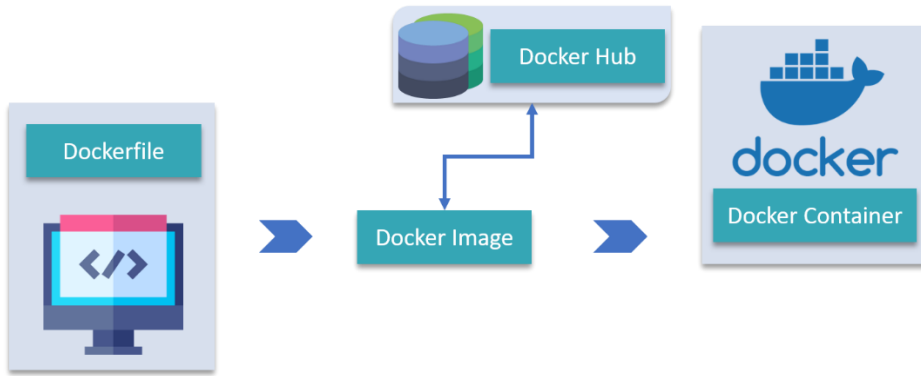


Docker Registry

Finally, Docker Registry is where the Docker Images are stored. The Registry can be either a user's local repository or a public repository like a Docker Hub allowing multiple users to collaborate in building an application. Even with multiple teams within the same organization can exchange or share containers by uploading them to the Docker Hub, which is a cloud repository similar to GitHub.

Docker Hub

Docker Hub is like GitHub for Docker Images. It is basically a cloud registry where you can find Docker Images uploaded by different communities, also you can develop your own image and upload on Docker Hub, but first, you need to create an account on DockerHub.



Now, let us install Docker.

Docker Installation:

I will be installing Docker on my Ubuntu 20.04.1 machine. You can also docker on windows and CentOS. You can follow the below videos to install Docker as per your desired environment.

Following are the steps to install Docker on Ubuntu:

1. Install required Packages
2. Setup Docker repository
3. Install Docker On Ubuntu

1. Install Required Packages:

There are certain packages you require in your system for installing Docker. Execute the below command to install those packages.

```
sudo apt-get install curl apt-transport-https ca-certificates software-properties-common
```

2. Setup Docker Repository:

Now, import Dockers official GPG key to verify packages signature before installing them with apt-get. Run the below command on terminal:

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add
```

```
edureka@edureka-VirtualBox:~$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add
OK
```

Now, add the Docker repository on your Ubuntu system which contains Docker packages including its dependencies, for that execute the below command:

```
sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable"
```

```
edureka@edureka-VirtualBox:~$ sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable"
```

3. Install Docker On Ubuntu:

Now you need to upgrade apt index and install Docker community edition, for that execute the below commands:

```
sudo apt-get update
```

```
sudo apt-get install docker-ce
```

```
edureka@edureka-VirtualBox:~$ sudo apt-get update
Hit:1 http://security.ubuntu.com/ubuntu artful-security InRelease
Hit:2 http://in.archive.ubuntu.com/ubuntu artful InRelease
Hit:3 http://in.archive.ubuntu.com/ubuntu artful-updates InRelease
Hit:4 http://in.archive.ubuntu.com/ubuntu artful-backports InRelease
Get:5 https://download.docker.com/linux/ubuntu artful InRelease [51.9 kB]
Get:6 https://download.docker.com/linux/ubuntu artful/stable amd64 Packages [2,509 B]
Fetched 54.4 kB in 1s (40.4 kB/s)
```

Congratulations! You have successfully installed Docker. Also, check out a few commonly used [Docker Commands](#).

DOCKER COMMANDS

1. docker --version

This command is used to get the currently installed version of docker

```
edureka@Manager-1: ~  
edureka@Manager-1:~$ docker --version  
Docker version 17.05.0-ce, build 89658be  
edureka@Manager-1:~$
```

2. docker pull

Usage: **docker pull** <image name>

This command is used to pull images from the **docker repository**(hub.docker.com)

```
edureka@Manager-1: ~  
edureka@Manager-1:~$ docker pull ubuntu  
Using default tag: latest  
latest: Pulling from library/ubuntu  
ae79f2514705: Pull complete  
c59d01a7e4ca: Pull complete  
41ba73a9054d: Pull complete  
f1bbfd495cc1: Pull complete  
0c346f7223e2: Pull complete  
Digest: sha256:6eb24585b1b2e7402600450d289ea0fd195cfb76893032bbbb3943e041ec8a65  
Status: Downloaded newer image for ubuntu:latest  
edureka@Manager-1:~$
```

3. docker run

Usage: **docker run -it -d** <image name>

This command is used to create a container from an image

```
edureka@Manager-1: ~  
edureka@Manager-1:~$ docker run -it -d ubuntu  
f49b58b66d1ebc7c2d9e42280c1e24019b3202fb3e69050c45e806e6f9b65f71  
edureka@Manager-1:~$
```

4. docker ps

This command is used to list the running containers

```
edureka@Manager-1: ~
edureka@Manager-1:~$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
f49b58b66d1e       ubuntu             "/bin/bash"        6 minutes ago      Up 6 minutes
angry_knuth
```

5. docker ps -a

This command is used to show all the running and exited containers

```
edureka@Manager-1: ~
edureka@Manager-1:~$ docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
fe6e370a1c9c       ubuntu             "/bin/bash"        12 seconds ago     Up 11 seconds
2b86a0703d4f       ubuntu             "/bin/bash"        About a minute ago Exited (0) 21 seconds ago
infallible_galileo
```

6. docker exec

Usage: `docker exec -it <container id> bash`

This command is used to access the running container

```
root@fe6e370a1c9c: /
edureka@Manager-1:~$ docker exec -it fe6e370a1c9c bash
root@fe6e370a1c9c:/#
```

7. docker stop

Usage: `docker stop <container id>`

This command stops a running container

```
edureka@Manager-1: ~
edureka@Manager-1:~$ docker stop fe6e370a1c9c
fe6e370a1c9c
edureka@Manager-1:~$
```

8. docker kill

Usage: `docker kill <container id>`

This command kills the container by stopping its execution immediately. The difference between ‘docker kill’ and ‘docker stop’ is that ‘docker stop’ gives the container time to shutdown gracefully, in situations when it is taking too much time for getting the container to stop, one can opt to kill it

```

edureka@Manager-1: ~
edureka@Manager-1:~$ docker kill d611cbc3789c
d611cbc3789c
edureka@Manager-1:~$ █

```

9. exit

Once you finish modifying the new container, exit out of it:

```
exit
```

Prompt the system to display a list of launched containers:

```
sudo docker ps -a
```

You will need the **CONTAINER ID** to save the changes you made to the existing image. Copy the ID value from the output.

```

root@deddd39fa163:/# exit
exit
sofiya@sofiya-VirtualBox:~$ sudo docker ps -a
CONTAINER ID        IMAGE               PORTS              COMMAND              NAMES              CREATED
STATUS              PORTS              NAMES              CREATED
deddd39fa163        cf0f3ca922e0       "bin/bash"         6 minutes ago
Exited (0) 43 seconds ago
befe9038ba17        cf0f3ca922e0       "/bin/bash"        6 minutes ago
Exited (0) 6 minutes ago
dreamy_goodall

```

10. docker commit

Usage: `docker commit <container id> <username/imagename>`

This command creates a new image of an edited container on the local system

```

edureka@Manager-1: ~
edureka@Manager-1:~$ docker commit fe6e370a1c9c hshar/ubuntuew
sha256:0678ee2e6b1e6a66ae7179c3be31610e5338d3004c52d25fc9f65fd2a63dc164
edureka@Manager-1:~$ █

```

11. docker login

This command is used to login to the docker hub repository

12. docker push

This command is used to push updated image to docker repository.

Usage: `docker push <username/imagename>`

13. docker search

We can use the command `docker search` to search for public images on the Docker hub. It will return information about the image name, description, stars, official and automated.

```
docker search MySQL
```


If you prefer a GUI-based search option, use the Docker Hub [website](#).

14. docker restart

Let's restart our stopped container by using the following command. We may want to use this after we reboot our machine.

```
docker restart f8c52bedeccc
```

15. docker rename

Now, let's change the container name from `compassionate_fermi` to `test_db`. We may want to change the name to keep track of our containers more easily.

```
docker rename compassionate_fermi test_db
```

16. docker logs

This command is helpful for debugging our Docker containers. It will fetch logs from a specified container.

```
docker logs test_db
```

17. docker rmi

Finally, if we want to free some disk space, we can use the `docker rmi` command with the image id to remove an image.

```
docker rmi eb0e825dc3cf
```

18. docker network

The following command in docker lists the details of all the network in the cluster.

```
$ docker network ls
```

NETWORK ID	NAME	DRIVER	SCOPE
85083e766f04	bridge	bridge	local
f51d1f3379e0	host	host	local
5e5d9a192c00	none	null	local

There are several other docker network commands.

```
$ docker network
```

Usage: docker network COMMAND

Manage networks

Commands:

connect	Connect a container to a network
create	Create a network
disconnect	Disconnect a container from a network
inspect	Display detailed information on one or more networks
ls	List networks
prune	Remove all unused networks
rm	Remove one or more networks

Run 'docker network COMMAND --help' for more information on a command.

19. Docker info

Get detailed information about docker installed on the system including the kernel version, number of containers and images, etc.

```
$ docker info
```

Containers: 3

Running: 1

Paused: 0

Stopped: 2

Images: 3

Server Version: 18.09.6

Storage Driver: overlay2

Backing Filesystem: extfs

Supports d_type: true

Native Overlay Diff: true

Logging Driver: json-file

Cgroup Driver: cgroupfs

Plugins:

Volume: local

Network: bridge host macvlan null overlay

Log: awslogs fluentd gcplogs gelf journald json-file local logentries splunk syslog

Swarm: inactive

Runtimes: runc

Default Runtime: runc

Init Binary: docker-init

containerd version: bb71b10fd8f58240ca47fbb579b9d1028eea7c84

runc version: 2b18fe1d885ee5083ef9f0838fee39b62d653e30

init version: fec3683

Security Options:

apparmor

seccomp

Profile: default

Kernel Version: 4.18.0-25-generic

Operating System: Ubuntu 18.10

OSType: linux

Architecture: x86_64

CPUs: 1

Total Memory: 4.982GiB

Name: geekflare

ID: RBCP:YGAP:QG6H:B6XH:JCT2:DTI5:AYJA:M44Z:ETRP:6TO6:OPAY:KLNJ

Docker Root Dir: /var/lib/docker

Debug Mode (client): false

Debug Mode (server): false

Username: geekflare

Registry: https://index.docker.io/v1/

Labels:

Experimental: false

Insecure Registries:

127.0.0.0/8

Live Restore Enabled: false

Product License: Community Engine

20. docker attach

Attach local standard input, output, and error streams to a running container

```
docker attach [CONTAINER]
```

```
root@samujsharma401ya:~# docker container ls
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS              NAMES
fca7f17dad2c       httpd              "httpd-foreground"  4 minutes ago       Up 17 seconds      80/tcp             blissful-hofstadter

root@samujsharma401ya:~# docker attach blissful_hofstadter
^C[Sat Nov 20 06:15:33.346490 2020] [age_wenttimeout] [pid 17114 139420481985280] AH00491: caught SIGTERM, shutting down
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS              NAMES
root@samujsharma401ya:~#
```

Node.js Node with Docker

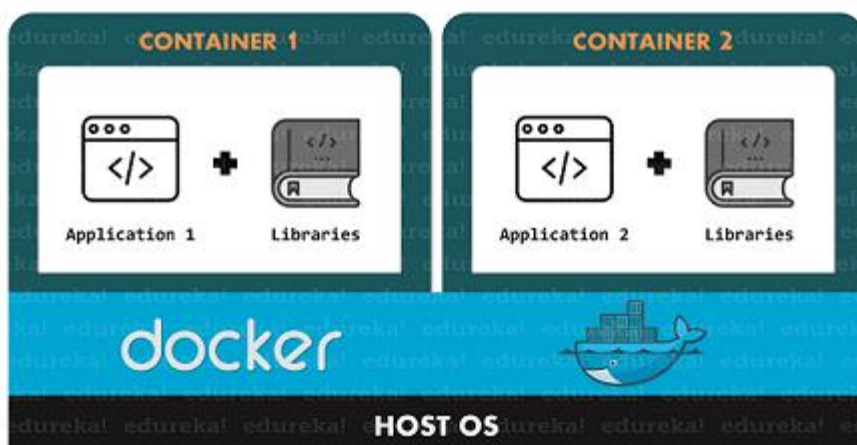
Every **Node.js developer** out there always puts in utmost efforts to make his application free any type of environment dependencies. But despite their measures, surprises occur all the time leading to the failure of the application. Well, this is where Docker comes to the rescue. In this Node.js Docker Tutorial, I will be giving you a complete walkthrough of how to Dockerize a Node.js application from scratch.

Docker

Docker is a containerization platform which is used for packaging an application and its dependencies together within a **Docker container**. This ensures the effortless and smooth functioning of our application irrespective of the changes in the environment.

Thus, you can think of Docker as a tool that is designed to make the creation, deployment, and execution of applications using the containers easier and efficient.

Talking about Docker Container it is nothing but a standardized unit that is used to deploy a particular application or environment and can be built dynamically. You can have any container such as Ubuntu, CentOS, etc. based on your requirement with respect to Operating Systems. Moreover, these containers aren't limited to just OS, you can have application-oriented OS as well. A few examples of such containers are CakePHP container, Tomcat-Ubuntu container, etc. To understand this better refer to the below diagram:



Now in this diagram, you can see that each and every application is running on a separate container along with its own set of dependencies & libraries. This ensures that each application is independent of others, enabling developers to build applications independently without any interference from other applications. Thus, being a developer you can simply build a container with different applications installed in it and hand it over to the QA team. The QA team then just needs to execute the container for replicating the developer's environment. Now, let me now throw some light on the nitty-gritty of Docker which will aid you in understanding the process of Dockerizing your Node.js application better.

Docker with Node.js

Below I have listed down a few of the most intriguing reasons to use Docker with your Node.js application:

- Docker helps in speeding up the application deployment process
- It makes the application portability across other machines easier
- It makes the version control easier and promotes component reuse
- Using Docker, one can easily share the Docker images and Dockerfiles
- Docker has a very lightweight footprint and thus puts minimal overhead on the applications
- It's easy and simple to use and maintain

I hope this gives you enough reasons to start using Docker right away. So, let's now dive deeper into this Node.js Docker Tutorial and see how exactly Docker can be used with [Node.js applications](#).

Demo: Node.js Docker

Before you start using Docker with Node.js, you need to make sure that Docker is already installed in your system and you have the right set of permissions to use it. Now that installation process is out of the way, let's now concentrate on Dockerizing a Node.js Application. In order to Dockerize a Node.js application, you need to go through the following steps:

1. Create Node.js Application
2. Create a Docker file
3. Build Docker Image
4. Execute

Create Node.js Application

In order to Dockerize a Node.js Application, the very initial thing you need is the Node.js Application. Once you are done developing the application, you need to make sure that the application is executing properly on the assigned port. In my case, I am using port 8080. If the application is working as expected, you can proceed to the next step.

Create a Dockerfile

In this step, we will be creating the Dockerfile which will enable us to recreate and scale our Node.js application as per our requirement. To complete this step, you need to create a new file in the root directory of the project and name it *Dockerfile*.

Here, I am using a lightweight alpine based image to build our Docker image on it. While creating a Dockerfile our main aim should be to keep the Docker image as small as possible in size all while availing everything that is required to run our application successfully.

Below I have written down the code that needs to add in your Dockerfile:

Dockerfile

```
1 FROM node:9-slim
2
3 # WORKDIR specifies the application directory
4 WORKDIR /app
5
6 # Copying package.json file to the app directory
7 COPY package.json /app
8
9 # Installing npm for DOCKER
9 RUN npm install
10
11 # Copying rest of the application to app directory
```

```
12 COPY ./app
13
14 # Starting the application using npm start
15 CMD ["npm","start"]
16
```

As you can see in the above code that I have used two distinct COPY commands to reduce the application rebuild time. As Docker can implicitly cache the result of each individual command, you don't need to execute all the commands from the beginning each time you try to create a Docker image.

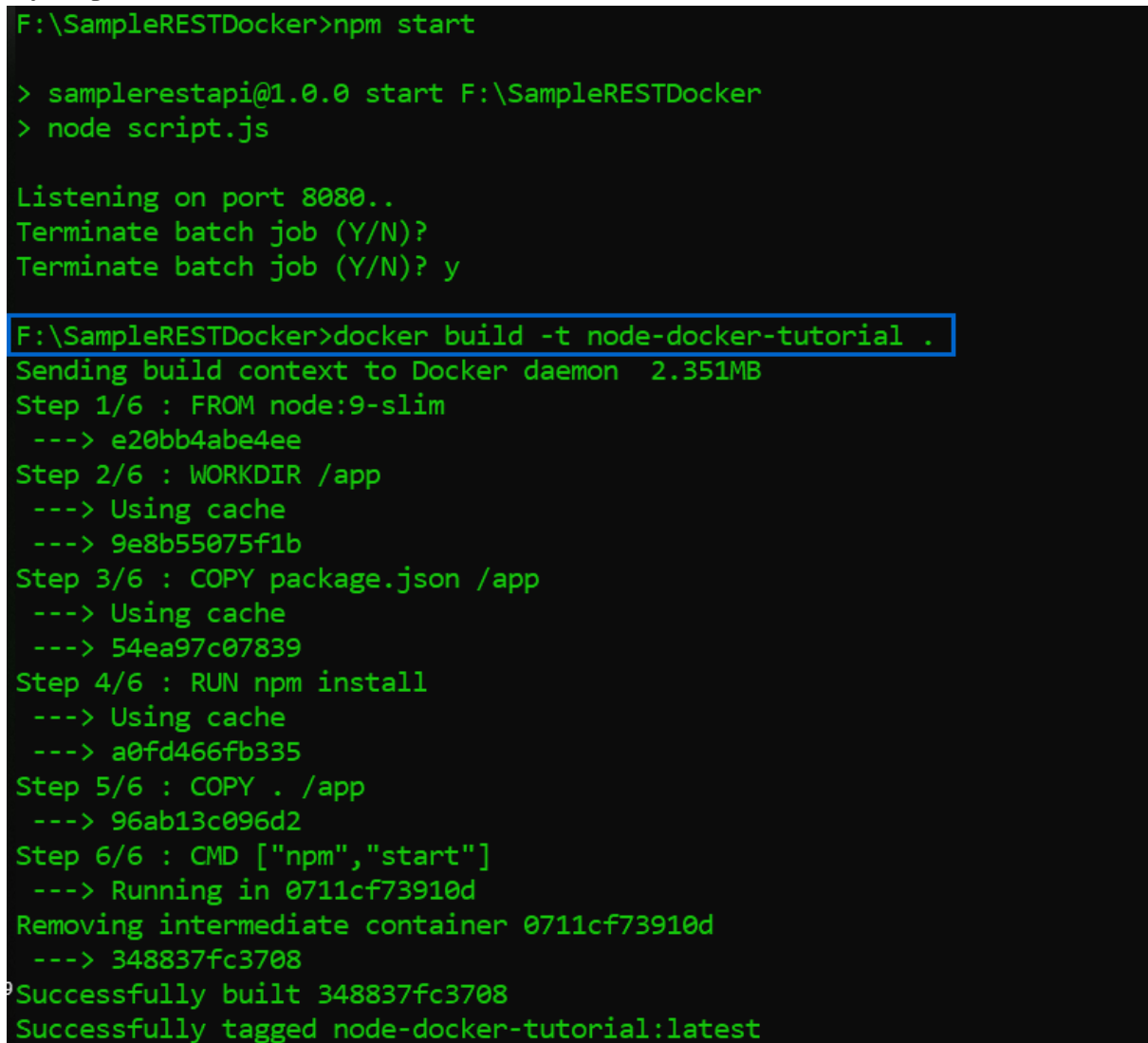
Now that you have successfully defined your Dockerfile, the next step is to Build a Docker Image. In the next section of this article, I will demonstrate how you can build your Docker image with ease.

Build Docker Image

Building a Docker image is rather easy and can be done using a simple command. Below I have written down the command that you need to type in your terminal and execute it:

```
1 docker build -t <docker-image-name> <file path>
```

Once you execute this command, you will see a 6 step output in your terminal. I have attached a screenshot to my output.



```
F:\SampleRESTDocker>npm start

> samplerestapi@1.0.0 start F:\SampleRESTDocker
> node script.js

Listening on port 8080..
Terminate batch job (Y/N)?
Terminate batch job (Y/N)? y

F:\SampleRESTDocker>docker build -t node-docker-tutorial .
Sending build context to Docker daemon 2.351MB
Step 1/6 : FROM node:9-slim
---> e20bb4abe4ee
Step 2/6 : WORKDIR /app
---> Using cache
---> 9e8b55075f1b
Step 3/6 : COPY package.json /app
---> Using cache
---> 54ea97c07839
Step 4/6 : RUN npm install
---> Using cache
---> a0fd466fb335
Step 5/6 : COPY ./app
---> 96ab13c096d2
Step 6/6 : CMD ["npm","start"]
---> Running in 0711cf73910d
Removing intermediate container 0711cf73910d
---> 348837fc3708
Successfully built 348837fc3708
Successfully tagged node-docker-tutorial:latest
```

If you are getting an output something similar to the above screenshot, then it means that your application is working fine and the docker image has been successfully created. In the next section of this Node.js Docker article, I will show you how to execute this Docker Image.

Executing the Docker Image

Since you have successfully created your Docker image, now you can run one or more Docker containers on this image using the below-given command:

```
1 docker run it -d -p <HOST PORT>:<DOCKER PORT> <docker-image-name>
```

This command will start your docker container based on your Docker image and expose it on the specified port in your machine. In the above command **-d flag** indicates that you want to execute your Docker container in a detached mode. In other words, this will enable your Docker container to run in the background of the host machine. While the **-p flag** specifies which host port will be connected to the docker port.

To check whether your application has been successfully Dockerized or not, you can try launching it on the port you have specified for the host in the above command.