# Basics of Docker

# 

# Learning Outcome:

# 

# After completion of this course, you will be able to:

* Describe basic building blocks of containers
* Describe need for containers and life cycle of containers
* Create containers using Docker and Docker compose
* Describe Docker networking model
* Attach Docker volumes

In order to deploy Applications, It takes infrastructure and architecture par excellence to manage applications with no friction between development, testing and release. Providing same environment accross development, testing and release teams becomes challenging.

Containers provide an excellent way of packaging our application with all of its dependencies in a package and delivering it irrespective of the target environment.

In this course, you are going through Docker, one of the container engines.

Docker is a container manager, based on Linux namespaces and cgroups. It provides a lightweight virtualization and has an immutable architecture.

# Basics of container and Docker

# What is a container?

Business requirements are ever-increasing. The demand for deploying application to meet those requirements are never-ending. All the application need necessarily require same set of tools and environment. It takes infrastructure and architecture par excellence to manage and deploy these divergent applications with no friction between development, testing and release.

Containers provide an excellent way of packaging our application with all of its dependencies in a package and delivering it irrespective of the target environment. This makes the deployment faster and consistent among all the environments (development, testing, production etc.,) maintaining the essence of the functionalities at the same time.

# What are Linux containers?

A Linux container is a single or a group of processes that are isolated from other processes running in the system. The Linux kernel performs this isolation using the kernel features such as namespaces, cgroups, etc.,

The files,libraries and dependencies necessary to run the containers are packaged and provided to the containers in the form of an image. This makes the containers easily portable and consistent among different environments involved in the software development cycle.

Linux Containers offer lightweight virtualisation since they share the operating system kernel of e host machine unlike Virtualisation which lets multiple operating systems run simultaneously. Hence, Linux containers are considered to be the best choice in case of limited resource availaility. Linux containers help us meet the most sought-after business requirement, which is faster deployment through its consistency across environments.

LXC (LinuX Containers) is an operating system-level virtualization method for running multiple isolated Linux systems (containers) on a single control host. It offers a simple command line interface that provides satisfying user-experience while working with containers.

# Features of Linux containers

Following are few irresistible features of Linux containers which are the major reasons why most of the organisations are moving towards containers now:

* It offers lightweight virtualization.
* A container can run one or more applications in it depending upon the requriement.
* Provisioning a container will not take more than few seconds/milliseconds.
* Near bare metal runtime performance.
* Faster deployment due to its consistency across various environments.
* Linux containers run on the operating system of the host machine , making it extremely lightweight and can be densely deployed.
* Provides agility.

# Working with Linux containers

Linux containers are considered as groups of isolated processes running in a host machine. Linux containers contain application packaged with all its dependencies and allow us to work on it. These isolated processes that are contained in the Linux system using several features built into the Linux kernel such as:

* Chroots
* Namespaces and
* Cgroups

Using these features, we can build a container in the Linux system without the use of Docker.

To build a Linux container, we need an OS image(Container image). The container image is an executable package of all the files and dependencies required to run  a container in the OS.

Before following the steps, make sure we are logged in as root user or have root user privileges.

1. [root@docker ~]$ whoami
2. root

# Step 1:  Making filesystem

Since a container image is a package of executable that contain all the files and dependencies required to run a container. For a CentOS image, we will now make a filesystem. Here, we have the dependencies archived in a tar file, centos-7-docker.tar.xz.  (If this tar file is not present, copy the basic necessary directories like /bin, /lib, etc., from the current root system "/", and paste it in a new directory. )

1. [root@docker ~]$ ls -l /var/tmp/Images/centos-7-docker.tar.xz
2. -rw-rw-rw-. 1 root root 42517348 Jun 16 09:09 /var/tmp/Images/centos-7-docker.tar.xz

This should be extracted into a directory. Make a directory 'rootfs' under /home/root

1. [root@docker ~]$ mkdir rootfs
2. [root@docker ~]$ pwd
3. /home/podxuser

Now, extract the centos-7-docker.tar.xz into the newly created directory.

1. [root@docker ~]$ tar xf /var/tmp/Images/centos-7-docker.tar.xz -C rootfs/

After extraction , the rootfs directory will look like a regular Linux system which has a "bin" directory with executables, an "etc" with system configuration, a "lib" with shared libraries, and so on.

1. [root@docker ~]$ ls rootfs/
2. anaconda-post.log bin dev etc home lib lib64 lost+found media mnt opt proc root run sbin srv sys tmp usr var

# Tools used by Linux Containers

The Linux containers are isolated processes that are contained in the Linux system using several features built into the Linux kernel, out of which we will be dealing with the following in this course:

* Chroot
* Namespaces and
* Cgroups

# Working in chroot environment

The first kernel feature that we will be dealing with is chroot. Chroot is a system call that acts like a thin wrapper around a process to restrict the process’ view of the filesystem. Chroot is used to create a separate environment for the container by creating an exclusive root directory for the container other than the actual system root (/). The newly created environment for the container applies for the child processes of the container as well.

In this case, we will restrict our process to the rootfs/ directory and then execute a bash shell.

1. [root@docker ~]$ sudo chroot rootfs /bin/bash
2. [root@rhel75 /]#
3. [root@rhel75 /]# pwd
4. /

To make sure that the processes are restricted to the rootfs/ directory, let us check the content of our new environment.

1. [root@rhel75 /]# ls
2. anaconda-post.log bin dev etc home lib lib64 lost+found media mnt opt proc root run sbin srv sys tmp usr var

The content of our new environment is the same as that of the rootfs/ directory.

Let us execute a simple command in our isolated environment.

1. [root@rhel75 /]# echo 'Hello from Container World!!!'
2. Hello from Container World!!!

Let us now check how effective and isolated is our chroot environment.

* Open a new Putty session and execute the top command on the host.

1. [root@docker ~]$ top

* In the chrooted shell, execute the following command.

1. [root@rhel75 /]# mount -t proc proc /proc
2. [root@rhel75 /]# ps aux | grep top
3. root 1423 0.1 0.2 161944 2196 ? S+ 08:33 0:00 top
4. root 1429 0.0 0.0 12460 968 ? R+ 08:33 0:00 grep --color=auto top

* Now try to kill the top command running in the host.

1. [root@rhel75 /]# pkill top

When we check the status of the top command now, we notice that it has been stopped. This is because the chrooted shell is running as root and it sees and kills the process running in the host with ease. This is definitely not expected out of containerized process. The Linux kernel offers another feature called namespaces which enables us to create restricted system views.

# Creation of namespaces

The Linux namespaces enable us to create restricted views of systems like the process tree, network interfaces, and mounts.

For example, by creating the namespaces for process tree of a container, we can let the container have a separate process tree, separate from the host system. This implies that the first process inside the container will have PID 1, irrespective of the number of processes already on the run in the host system.

Like “chroot”, “unshare” command is also a thin wrapper around syscall and it allows us to set the namespace manually.

We will now exit out of the chroot shell environment and create a PID namespace for the shell and then execute chroot like the previous demo. Execute the following command:

1. [root@docker ~]$ sudo unshare -p -f --mount-proc=$PWD/rootfs/proc chroot rootfs /bin/bash

Now we will check the list of all processes.

1. [root@rhel75 /]# ps aux
2. USER PID %CPU %MEM VSZ RSS TTY STAT START TIME COMMAND
3. root 1 0.0 0.1 15200 2004 ? S 08:47 0:00 /bin/bash
4. root 10 0.0 0.1 50868 1812 ? R+ 08:47 0:00 ps aux

From the snapshot above, we notice a process with PID 1 and we are not able to look at the processes running in the host system. To ensure this, open a new Putty session of the server. Run the top command to make sure that we are now not able to see the hosts's process tree. This best explains the significance of namespaces.

# Entering namespace with nsenter

One of the powerful property exhibited by namespace is its composability.

Composability is a system design principle that deals with the inter-relationships of components. A highly composable system provides components that can be selected and assembled in various combinations to satisfy specific user requirements.In this case, processes may choose to separate some namepsaces but share others.

We can enter a namespace using the nsenter command. It in turn uses the setns syscall to enter into the namespace.

In order to to do this, keep the chroot(with namespace) running from the previous demo. Open a new putty session for the host and find the shell running bash from previous demo. This can be done by executing the following command:

1. [root@docker ~]$ ps aux | grep /bin/bash | grep root | grep -v unshare
2. root 1440 0.0 0.1 15200 2004 pts/0 S+ 14:17 0:00 /bin/bash

The kernel exposes the namespace of a process under /proc/$PID/ns.

Here, the PID is 8136. So, /proc/8136/ns is the process namespace we are going to enter using nsenter.

1. [root@docker ~]$ export MYPID=1440
2. [root@docker ~]$ sudo ls -l /proc/$MYPID/ns
3. total 0
4. lrwxrwxrwx. 1 root root 0 Jun 16 14:38 ipc -> ipc:[4026531839]
5. lrwxrwxrwx. 1 root root 0 Jun 16 14:38 mnt -> mnt:[4026532119]
6. lrwxrwxrwx. 1 root root 0 Jun 16 14:38 net -> net:[4026531956]
7. lrwxrwxrwx. 1 root root 0 Jun 16 14:38 pid -> pid:[4026532129]
8. lrwxrwxrwx. 1 root root 0 Jun 16 14:38 user -> user:[4026531837]
9. lrwxrwxrwx. 1 root root 0 Jun 16 14:38 uts -> uts:[4026531838]

The nsenter command provides a wrapper around setns to enter the namespace. Execute the following command to enter into an already existing namespace, instead of creating one.

1. [root@docker ~]$ sudo nsenter --pid=/proc/$MYPID/ns/pid unshare -f --mount-proc=$PWD/rootfs/proc chroot rootfs /bin/bash
2. [root@rhel75 /]# ps aux
3. USER PID %CPU %MEM VSZ RSS TTY STAT START TIME COMMAND
4. root 1 0.0 0.1 15200 2004 ? S+ 08:47 0:00 /bin/bash
5. root 11 0.0 0.0 107936 592 ? S 09:14 0:00 unshare -f –mount-proc=/home/podxuser/rootfs/proc chroot rootfs /bin/bash
6. root 12 0.0 0.1 15200 2004 ? S 09:14 0:00 /bin/bash
7. root 21 0.0 0.1 50868 1816 ? R+ 09:14 0:00 ps aux

After entering the new namespace, we are able to see the process ID of the first shell(chroot shell) which was already running.

Along with this basic feature offered by namsepaces, there are many other features and capabilities carried by namespaces.

# Control groups

Cgroups(Control groups) is the kernel feature offered by Linux systems that allows the kernel to restrict the amount of resource each process uses like, processor, CPU, memory, disk, network etc., Such isolation of resources is required to prevent any malicious process or event in the background, which could drain RAM or other resources.

Like namespaces, the Linux kernel exposes the cgroups in the /sys/fs/cgroup directory.

We will see how to control CPU resources with the use of cgroup in the upcoming demo.

# Controlling CPU with cgroup

A cgroup can be created manually just by creating a directory under the directory of the corressponding resource we want to control. Since we are trying to control CPU with cgroups in this demo, we will create a directory under the CPU directory, /sys/fs/cgroup/cpu.

1. [root@rhel75 ~]$ ls /sys/fs/cgroup/
2. blkio cpu cpuacct cpu,cpuacct cpuset devices freezer hugetlb memory net\_cls net\_cls,net\_prio net\_prio perf\_event pids system

# Step 1: Create a directory

We will create a cgroup for "podxuser" which will control cpu.  For this, create a directory “podxuser” under /sys/fs/cgroup/cpu directory. Note: x in podxuser stands for any number. Here, pod1user is used.

1. [root@docker ~]$ sudo su
2. [root@rhel75 /]# export PODNUM=pod1user
3. [root@rhel75 /]# mkdir /sys/fs/cgroup/cpu/$PODNUM

Once we create the directory, the kernel auto populates the directory with the files needed to configure the cgroup. Let us check the content of the "linuxuser" directory.

1. [root@rhel75 /]# ls /sys/fs/cgroup/cpu/$PODNUM/
2. cgroup.clone\_children cgroup.procs cpuacct.usage cpu.cfs\_period\_us cpu.rt\_period\_us cpu.shares notify\_on\_release
3. cgroup.event\_control cpuacct.stat cpuacct.usage\_percpu cpu.cfs\_quota\_us cpu.rt\_runtime\_us cpu.stat tasks

# 

# Step 2: Configuring our cgroup

If the CPU resource used by the cgroup has to be limited to 2%, we write the corresponding value inside the cpu.cfs\_quota\_us file. The value is represented in microseconds(us). There are two kinds of files:

* Cpu.cfs\_period\_us: Represents the period how often the CPU resources should be reallocated to the cgroup in microseconds.
* Cpu.cfs\_quota\_us: Represents the time for how long the tasks in the cgroup can access the CPU during a single period.

In this demo, we will restrict the tasks in “podxuser” cgroup to use only 2% of the CPU using the Complete Fair Scheduler. To do so, make the following changes in  cpu.cfs\_quota\_us file.

Enter "2000" as the value in /sys/fs/cgroup/cpu/podxuser/cpu.cfs\_quota\_us file by entering the following command.

1. *# echo 2000 > /sys/fs/cgroup/cpu/$PODNUM/cpu.cfs\_quota\_us*

# Step 3: Adding the processes to the cgroup

The tasks file under the “podxuser” directory lets us add processes to our cgroup. We can add a process to the cgroup by just writing the corresponding PID of the process in the /sys/fs/cgroup/cpu/podxuser/tasks file.

Let us add the process ID of the current process to the tasks file.

1. [root@rhel75 /]# echo $$ > /sys/fs/cgroup/cpu/$PODNUM/tasks

The limitation of resources configured in any cgroup apply for all the processes, whose PID is present in the tasks file of the particular cgroup.

Let us now test whether the cgroup has been implemented properly. Run the following python one-liner which will try to choke the CPU and see what happens.

1. [root@rhel75 /]# python -c "while True: 42\*42"

Open another Putty session and check the CPU utilization of the particular process by running the top command.

1. [root@rhel75 /]# top

The kernel will allocate only 2% of the CPU as mentioned in the cpu.cfs\_quota\_us file, provided the setup is done correctly. Without this limit, it would gobble up the CPU.

The cgroup can be removed once we exit from all the shells using the following command.

1. [root@rhel75 /]# rmdir /sys/fs/cgroup/cpu/$PODNUM

# Controlling memory with cgroup

A cgroup can be created manually just by creating a directory under the directory of the corressponding resource we want to control. Since we are trying to control CPU with cgroups in this demo, we will create a directory under the CPU directory, /sys/fs/cgroup/cpu. Let us create a memory cgroup.

# Step 1: Create a directory

We will create a cgroup called “podxuser”, which will control memory.  For this, create a directory “podxuser” under /sys/fs/cgroup/memory.

1. [root@docker ~]$ sudo su
2. [root@rhel75 /]# export PODNUM=podxuser
3. [root@rhel75 /]# mkdir /sys/fs/cgroup/memory/$PODNUM

Once we create the directory, the kernel auto populates the directory with the files needed to configure the cgroup.

1. [root@rhel75 /]# ls /sys/fs/cgroup/memory/$PODNUM/
2. cgroup.clone\_children memory.kmem.slabinfo memory.memsw.failcnt
3. memory.soft\_limit\_in\_bytes
4. cgroup.event\_control memory.kmem.tcp.failcnt memory.memsw.limit\_in\_bytes memory.stat
5. cgroup.procs memory.kmem.tcp.limit\_in\_bytes memory.memsw.max\_usage\_in\_bytes memory.swappiness
6. memory.failcnt memory.kmem.tcp.max\_usage\_in\_bytes memory.memsw.usage\_in\_bytes
7. memory.usage\_in\_bytes
8. memory.force\_empty memory.kmem.tcp.usage\_in\_bytes memory.move\_charge\_at\_immigrate
9. memory.use\_hierarchy
10. memory.kmem.failcnt memory.kmem.usage\_in\_bytes memory.numa\_stat notify\_on\_release
11. memory.kmem.limit\_in\_bytes memory.limit\_in\_bytes memory.oom\_control tasks
12. memory.kmem.max\_usage\_in\_bytes memory.max\_usage\_in\_bytes memory.pressure\_level

# Step 2: Configuring our cgroup

Let us now limit the cgroup to 30MB of memory and turn swap off.

1. To limit the memory, make the following changes in the /sys/fs/cgroup/memory/$PODNUM/memory.limit\_in\_bytes file.

1. [root@rhel75 /]# echo "30000000" > /sys/fs/cgroup/memory/$PODNUM/memory.limit\_in\_bytes

2. To turn the swap off, enter "0" as the value in /sys/fs/cgroup/memory/linuxuser/memory.swappiness file

1. [root@rhel75 /]# echo "0" > /sys/fs/cgroup/memory/$PODNUM/memory.swappiness

# Step 3: Adding the processes to the cgroup

The tasks file under the “podxuser” directory lets us add processes to our cgroup. We can add a process to the cgroup by just writing the corresponding PID of the process in the /sys/fs/cgroup/memory/podxuser/tasks file.

Let us add the process ID of the current process to the tasks file.

1. [root@rhel75 /]# echo $$ > /sys/fs/cgroup/memory/$PODNUM/tasks

Let us now test whether the cgroup has been implemented properly. Create a python program which will continuously eat memory and then run it.

1. [root@rhel75 /]# cat eatmem.py
2. f = open("/dev/urandom", "r")
3. data = ""
4. i=0
5. while True:
6. data += f.read(10000000) *# 10mb*
7. i += 1
8. print "%dmb" % (i\*10,)

Run this program  now.

1. [root@rhel75 /]# python eatmem.py
2. 10mb
3. 20mb
4. 30mb
5. Killed

If the setup has done properly, this program will not crash the computer. The kernel will take care of controlling the memory and kill this rogue program. This can be checked in the /var/log/messages.

1. [root@rhel75|/root]# tail /var/log/messages
2. Jun 16 15:36:38 rhel75 kernel: Task in /podxuser killed as a result of limit of /podxuser
3. Jun 16 15:36:38 rhel75 kernel: memory: usage 48828kB, limit 48828kB, failcnt 34
4. Jun 16 15:36:38 rhel75 kernel: memory+swap: usage 48828kB, limit 9007199254740988kB, failcnt 0
5. Jun 16 15:36:38 rhel75 kernel: kmem: usage 0kB, limit 9007199254740988kB, failcnt 0
6. Jun 16 15:36:38 rhel75 kernel: Memory cgroup stats for /podxuser: cache:4KB rss:48824KB rss\_huge:28672KB mapped\_file:0KB swap:0KB
7. inactive\_anon:31712KB active\_anon:17112KB inactive\_file:4KB active\_file:0KB unevictable:0KB
8. Jun 16 15:36:38 rhel75 kernel: [ pid ] uid tgid total\_vm rss nr\_ptes swapents oom\_score\_adj name
9. Jun 16 15:36:38 rhel75 kernel: [ 9107] 0 9107 28859 532 14 0 0 bash
10. Jun 16 15:36:38 rhel75 kernel: [ 9127] 0 9127 43048 12645 41 0 0 python
11. Jun 16 15:36:38 rhel75 kernel: Memory cgroup out of memory: Kill process 9127 (python) score 1008 or sacrifice child
12. Jun 16 15:36:38 rhel75 kernel: Killed process 9127 (python) total-vm:172192kB, anon-rss:48580kB, file-rss:2000kB, shmem-rss:0kB

The cgroup can be removed once we exit from all the shells using the following command.

1. [root@rhel75|/root]# rmdir /sys/fs/cgroup/memory/$PODNUM

# Moving to docker

As we know, Linux containers existed since the beginning. But using these Linux containers to deploy application is something brand new and is being adapted by organizations widely. There are far too many Container management frameworks available which automates application deployment inside containers. Docker is one such open source project. It provides a Docker CLI to manage the containers throughout their lifecycle.

Docker is being discussed in the upcoming modules.

# Difference between container and virtual machine

Before learning the difference between a container and a virtual machine, we must understand the advent of them both over the years. We must learn the difference between a monolithic application and a microservice.

# Monolithic applications

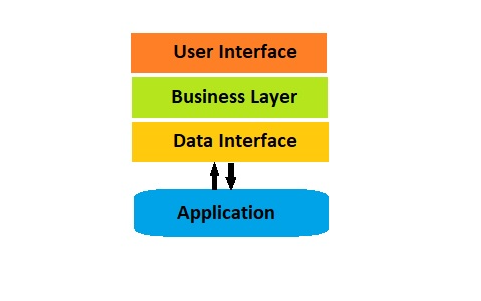
Software engineering offers continuous improvements in software deployment approaches in order to meet the substantial business requirements. Recent times have proved that nobody would prefer their application to have a downtime for even a minimum of one hour per week for maintenance. We want the changes to our applications to be done in no time so that the application is up all the 24 hours of a day.

Since the pattern of requirement changes from time to time as technology advances, we can survive this wave only by making sure that we can deploy the changes to an application faster than others.

One such traditional software deployment architecture is the monolithic application architecture.

* Monolithic application describes a single-tiered software application in which different components combined into a single program from a single platform.
* The monolithic application is built as a single instance. It often sees a larger team working on a single instance that has to be deployed. If there are changes to be imposed to the application, the developer has to build and deploy a new version of the application with the changes made.
* Usually, monolithic applications are built using a single development stack. A single development stack is used to build all the components of the particular application irrespective of the tool required by specific component.

In the case of a monolithic application, we commonly observe multiple layers of the application tightly coupled together. The user interface and the data access layer are grouped together. This is usually done via a large code repository, and both of these layers will probably depend on the other to be in place in order to run. Furthermore the user interface and data access layers typically are bundled into the same process at runtime.



# Downside of monolithic applications

Monolithic application has its own downside:

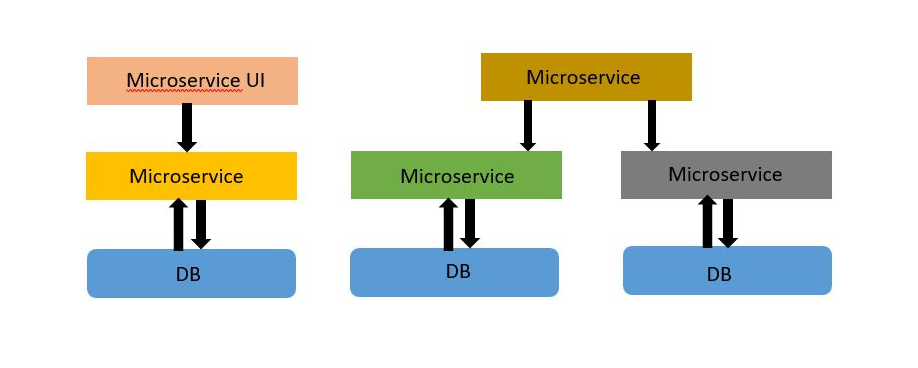
1. Since everybody as a team is working on a single instance, not much understanding about the application can be expected from any developer.
2. Monolithic applications are developed using a single development stack. This limits the usage of right tool for the right requirement.
3. The deployment of monolithic applications can never be agile.
4. Since the whole application is managed as a single instance, scaling can be quite challenging.
5. Complete redeployment is mandatory on every update.
6. Less reusability.
7. Less to no reliability. Any bug in a single module will bring down the whole application.

These disadvantages are sure an obstacle in achieving the effective way of software deployment which made organisations move towards the microservices architecture.

# Microservices architecture

The insufficient pace of system development offered by monolithic applications is best overcome by introducing the evolutionary microservices architecture. Microservice architecture, in concert with cloud deployment technologies, API management, and integration technologies, provides a different approach to software development.

It is a way to decompose large monolithic applications into loosely coupled components, each responsible for specific and independent services.



# Advantages of Microservices

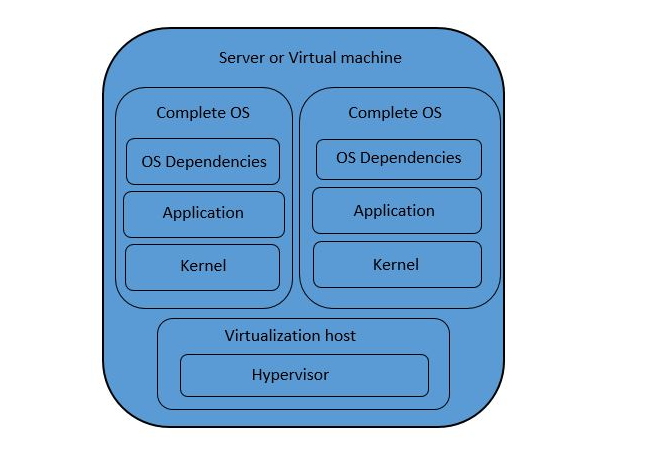
Microservice architecture has the following advantages making it capable of overcoming the drawbacks of monolithic applications.

1. The application is disassembled into separate independent services, each of which is developed by a small group of developers with expertise in the particular field.
2. Scaling can be done easily since services are independent of each other.
3. Since application is fragmented, right tool can be used for the right service unlike a single development stack.
4. Easy to deploy – architecture of low-weight and independent microservices enables frequent and fast deployment of new functionalities with no influence on other running applications.

# Virtualization

Virtualization is the creation of a virtual -- rather than actual -- version of something, such as an operating system, a server, a storage device or network resources.

* A virtual machine is an isolated environment which has an operating system of its own. It can contain either one or more applications inside it.
* A single physical host can run either one or more virtual machines.
* Virtual machines virtualize both OS as well as hardware of the real machine.
* Each VM is completely independent of each other.
* These multiple virtual machines can run several operating systems and applications on just a single physical server.



# Virtualization and Microservices

Microservices and virtualization technology can be combined together to enable multiple services to be developed and deployed irrespective of the location, the developer is present in and irrespective of the environment, the developer is working in.

* In Microservices architecture, the application is coupled down to independent services. These services have their own environment and tools with which the developer will work on it. In some cases, the resources are too costly to replicate the actual servers in different places of the developers which at the end, affects the productivity. Virtualisation overcomes this limitation and majorly responds to infrastructure limitation during software development.
* Microservice is coupled with virtualization to make the deployment process more effective. By virtualizing the environment associated with each microservice, the deployment process can be made faster and both cost and time effective.
* Time and cost overhead can be avoided by virtualizing the physical server rather than creating a physical copy of the actual server in places where the developers are present.

# Containerization

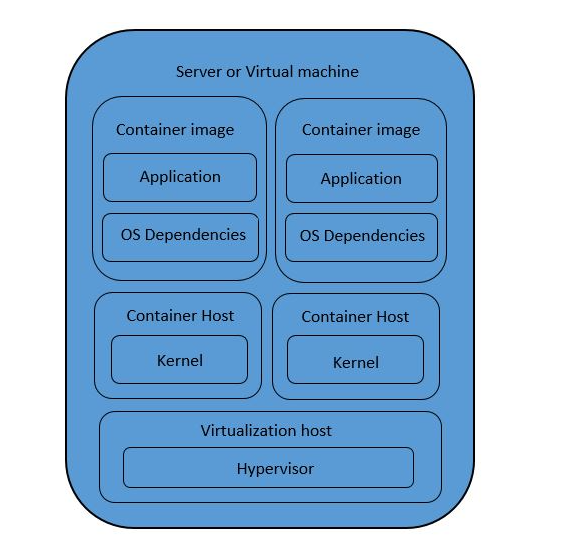
Containerization comes as a lightweight alternative to virtualization. The application with all its dependencies and shared libraries are packed and made into containers. These containers are then deployed to anywhere it is needed irrespective of the environment of the target machine. This makes it possible to run multiple containers in a single OS. This further reduces the overhead cost cause by virtualization.

* Containers have remained in Linux systems since way before.
* Containerization derives its foundation from LXC format.
* Unlike virtualization, containers provide a lightweight environment by virtualising just the OS and not the hardware.

# Immutability of Containers

Primarily, Containers offer an immutable infrastructure. In a mutable infrastructure, the current state of the system would be a result of continuous upgrades and changes and it is a tedious process to keep a record of what was changed when. Whereas in an immutable infrastructure offered by containers, the new change is updated in terms of a new container image. This implies that the old image is always there for the rescue if the update doesn’t go as planned.

Updating changes inside an existing container is also possible. But it negates the feature of immutability offered by Containers.



# Virtualization vs Containerization

Choosing whether to virtualize or containerize our microservices is a challenging task. Because both the process backs up different requirements that arise during software development process. Containerization has following upsides over virtualization.

1. Containerization reduces the overhead costs caused by virtualization, since only the OS of the physical server is virtualised.
2. Microservices are better isolated in case of containers. Multiple microservices requiring different environments cannot be packed in a single VM due to overlapping of environments. But multiple containers can be deployed in a single OS.
3. VMs take more space than containers. The same physical host can contain more containers than VMs.
4. Creation process is swift in case of containers when compared with VMs.
5. Versioning of applications is consistent in case of containers.
6. Rollback is easier in containers than virtual machines. Because, deploying a container hardly takes a few seconds.

Containers offer all the above advantages and few more which can be explored once we get our hands on containers.

# What is Docker?

The official definition of Docker goes like this:

"Docker is an open platform for developers and sysadmins to build, ship, and run distributed applications. Consisting of Docker Engine, a portable, lightweight runtime and packaging tool, and Docker Hub, a cloud service for sharing applications and automating workflows, Docker enables apps to be quickly assembled from components and eliminates the friction between development, QA, and production environments. As a result, IT can ship faster and run the same app, unchanged, on laptops, data center VMs, and any cloud”

–source: https://www.docker.com/whatisdocker/

Docker is a container manager that enables us to develop, deploy and run applications with containers. As mentioned earlier, containerization is not new. But deploying applications using containerization is.

Docker is a build system:

1. Images are built from sources.
2. Images are built using Docker file.

* It is a set of REST APIs:
  1. Engine API (controls the docker engine).
  2. Plugin API (extends the engine → network, storage, authorization).
  3. Registry API (publish/download images).

# 

# Features of Docker

Docker is being increasingly adopted by many organizations because of the following features:

• It offers lightweight virtualization.

• It adopts its foundation from the LXC format especially, the namespaces and cgroups of Linux containers.

• Docker offers immutable architecture.

• Immutable images - instance is ephemeral, persistent data is stored outside the container, on the host or volume-containers.

• Instant deployment is a key feature of Docker.

• Docker is more suitable for micro-services (one process, one container) - is designed to support a single application.

# Clustering

Along with all the essential features offered by Docker, one such feature that is not to be missed is "Clustering". Docker allows us to cluster containers across different physical servers in the network. This ensures effective management of containers by a manager node. Two broadly applicable options available for clustering are Docker swarm and Kubernetes. Combined with clustering, docker depicts the following functionalities:

* Provides maximum application density per machine (currently in the hundreds) than virtualization.
* Improved 'application namespace' isolation across your cluster.
* No longer need to ensure stacks / sub-stacks are rolled onto your cluster.
* Enables multiple versioning across a cluster easily.
* Docker combined with clustering can obviate traditional deployment problems.

# Downside of Traditional deployment

When we talk about containerization, we cannot omit discussing about legacy application deployment infrastructure. Since the time the industry shifted from monolithic applications to microservices, the deployment cycles has only become more and more complex. Now, the deployments have to be fast and iterative. Each application is developed with the right set of tools resulting in a good number of multiple environments at the end.

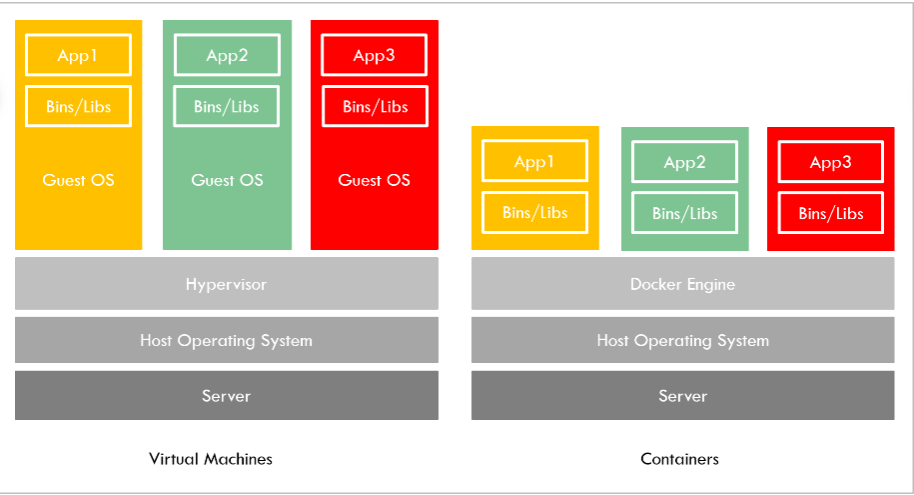
The traditional deployment model cannot cope up with these increasing requirements. Some of the reasons being:

* Many different stacks(languages, databases and frameworks).
* Many different targets. The application developed in multiple environments have to deployed to different targets such as pre-production, QA, testing etc.,

# Why docker?

Containerization using Docker deals with the problems faced with traditional application deployment as follows:

* Docker does not possess any dependency problem. If it builds in your system, it can build anywhere.
* Multiple environments for production, QA, testing etc., can be created in minutes.
* Implement reliable CI easily.
* Container images can be used as build artifacts.
* The container image is an executable which contains all the files and dependencies required to run the application. Hence, creating an environment can be as easy as running an executable file.
* Images are bigger, but they are broken down into layers.
* Changes are updated by just updating the layer which underwent change.
* It reduces the cost overhead by saving disk, network and memory usage.



# Loading Docker images

Docker launches a container by running an image. The container is nothing but a running instance of a container image. A container image is a packaged executable containing all the code, dependencies, shared libraries etc., required to run an application.

Apparently, to launch a container, we need the required container image. Since containers follow layering, the base layer of the container will be the container image.

Since we are working in a lab environment, we are going to preload all the Docker images needed for the lab by running a script, which will load the Docker images.

1. [root@docker ~]# cd /root/docker-images
2. [root@docker|/root/docker-images]# sh -x import-for-docker-lab.sh

This script will preload all the Docker images required for our current lab in Docker engine.(This step can be skipped if registry is used to get the Docker images.)

Executing the following command gives us the information about the docker images that has been imported:

1. [root@docker ~]# docker images
2. REPOSITORY TAG IMAGE ID CREATED SIZE
3. docker.io/centos latest 3fa822599e10 2 weeks ago 203.5 MB
4. docker.io/busybox latest 6ad733544a63 5 weeks ago 1.129 MB
5. docker.io/mhausenblas/simpleservice 0.5.0 601917f29430 7 months ago 682.6 MB
6. docker.io/jpetazzo/clock latest 12068b93616f 2 years ago 2.43 MB

# Life cycle of a container

# Life cycle of a container

A Docker container is creating by running a Docker image. The Docker image is pulled from the repository for this purpose. They can also be created on our own to satisfy our requirements. Once the container is created, it behaves as an isolated process in the OS running with its full environment. These containers go through a life cycle of changes when they are alive. The life cycle of a container defines the the state the container is currently in.

There are different states to a container as listed below:

Created: The state of the container when it gets created. In this state, the container has not yet started.

Running: The container is currently running in this state, which means it has started.

Paused: The container's processes have been paused. It is like temporarily stopping a container.

Restarting: The container is in the process of getting restarted.

Exited: The container is stopped either by the user or automatically due to a problem.

Dead: The container that has been removed. It no longer lives in the Docker host.

# 

# Container commands

The following is a list of commands that can be used to work with the containers throughout their life cycle.

1. To create a container:

1. $ docker run <image>

2. To run a container in the interactive mode:

1. $ docker run -it <image>

3. To run a container in the background:

1. $ docker run <image> -d

4. To attach to a container running in the background:

1. $ docker attach <containerID>

5. To get the statistics of a running container:

1. $ docker stat <containerID>

6. To see the processes withing a container:

1. $ docker top <containerID>

7. To stop a running container:

1. $ docker stop <containerID>

8. To kill the processes inside a container:

1. $ docker kill <containerID>

9. To remove a container:

1. $ docker rm <containerID>

10. To list all the containers in the Docker host:

1. $ docker ps -a

# Creating the first container

A container is a running instance of a docker image. Any number of containers can be created inside a Docker host. Any container can be created on any host. It has to be noted that anything that happens in the host machine or the container is exclusive to only themselves.

Docker engine launches a container by running a docker image.

Let us try to run our very first container. In this demo, we will run a simple "busybox" container in the interactive mode. The busybox image will be pulled from the repository.

Step 1: Execute the <docker run> command to create the container.

1. $ docker run -it busybox
2. Unable to find image 'busybox:latest' locally
3. latest: Pulling from library/busybox
4. 91f30d776fb2: Pull complete
5. Digest: sha256:9ddee63a712cea977267342e8750ecbc60d3aab25f04ceacfa795e6fce341793
6. Status: Downloaded newer image for busybox:latest
7. c4ef2e319cdf / *#*

The busybox image has been pulled from the repository."c4ef2e319cdf" is the ID of the container. The container ID is an alphanumeric string which is used to identify the container throughout its lifecycle.

Step 2: We have launched the container in interactive mode. Exit from the container and check the list of containers using following command.

1. c4ef2e319cdf / *# exit*
2. $ docker ps -a
3. CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
4. c4ef2e319cdf busybox "sh" About a minute ago Exited (0) 10 seconds ago upbeat\_yonath

The status of the container is exited which means the container was stopped.(We exited the container)

Step 3: Start the container using <docker start> command and execute <docker ps>.

1. $ docker start c4ef
2. c4ef
3. $ docker ps -a
4. CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
5. c4ef2e319cdf busybox "sh" 3 minutes ago Up 2 seconds upbeat\_yonath

The status of the container is now "Up" which means the container is running. But we have not connected to the terminal of the container by starting it. Execute the next step to do so.

Step 4: Attach to the terminal of the container using <docker attach> command.

1. $ docker attach c4ef
2. c4ef2e319cdf / *#*

Step 5: Exit the container. Now, the container is not running. We can now remove the container by executing <docker rm> command.

1. $ docker rm c4ef
2. c4ef
3. $ docker ps -a
4. CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

No container is running in the Docker host at the moment.

# Running multiple CentOS containers

As discussed, any number of containers can be run in a Docker host. But they are all isolated from each other and from the host. Whatever happens inside the containers stays inside the containers. The same holds good for the Docker host as well.

In this demo, we will launch multiple Centos containers in the interactive mode and keep them running.

Step 1: Run the first CentOS container.

1. $ docker run -it centos
2. Unable to find image 'centos:latest' locally
3. latest: Pulling from library/centos
4. 6910e5a164f7: Pull complete
5. Digest: sha256:4062bbdd1bb0801b0aa38e0f83dece70fb7a5e9bce223423a68de2d8b784b43b
6. Status: Downloaded newer image for centos:latest
7. [root@d6a35a07fd6a /]#

Step 2: Exit from this container to create a second CentOS container.

1. [root@d6a35a07fd6a /]# exit
2. exit
3. $ docker run -it centos
4. [root@146bd65f3a1c /]#

Step 3: Open a new terminal session and check the status of the containers.

1. $ docker ps -a
2. CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
3. 146bd65f3a1c centos "/bin/bash" 4 minutes ago Up 4 minutes hardcore\_liskov
4. d6a35a07fd6a centos "/bin/bash" 5 minutes ago Exited (0) 4 minutes ago frosty\_curran

We can see that the first CentOS container is in Exited state and the second container is running. In the next step, we will make the first Centos container run too.

Step 4: Start the first Centos container which is now in exited state.

1. $ docker start d6a35a07fd6a
2. d6a35a07fd6a

Step 5: Check the list of containers and their state.

1. $ docker ps -a
2. CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
3. 146bd65f3a1c centos "/bin/bash" 7 minutes ago Up 7 minutes hardcore\_liskov
4. d6a35a07fd6a centos "/bin/bash" 7 minutes ago Up 47 seconds frosty\_curran

Now, we can see that both the CentOS containers are up and running.

Step 6: Attach to the first CentOS container now and start working around.

1. $ docker attach d6a35a07fd6a
2. [root@d6a35a07fd6a /]#

Note: As soon as we exit the containers, the containers are moving to the stopped state. How can we exit from a container and still keep it running?

# Non-interactive containers

In the previous demo, we created two running CentOS containers using two different terminal sessions. Because, once a container is launched in the interactive mode, it occupies the terminal leading to the user not being able to do other tasks. Then the user either opens a new terminal session and launches the second container or exits from the first container and launches the second one in the same session. But in the second method, the first container stops running since we exit from it.

Now, to overcome this problem and to run multiple containers in a single session, Docker allows us to run containers in the background. In this demo, we will launch container that runs in the background. A jpetazzo/clock container is created in this demo. This container displays the current system time continuously thus occupying the terminal.

Step 1: Run the jpetazzo/clock container.

1. $ docker run jpetazzo/clock
2. Unable to find image 'jpetazzo/clock:latest' locally
3. latest: Pulling from jpetazzo/clock
4. 0f8c40e1270f: Pull complete
5. Digest: sha256:ace75dda37174abb563799a8b9b2043505619559fe1120a26a63363dc48bcd26
6. Status: Downloaded newer image for jpetazzo/clock:latest
7. Wed Jul 8 06:42:31 UTC 2020
8. Wed Jul 8 06:42:32 UTC 2020
9. .
10. .

The time gets printed on the terminal as long the container is running. If we want to launch another container in the same session, this container has to be stopped.

Step 2: Exit from the container using Ctrl+C.

1. Wed Jul 8 06:42:35 UTC 2020
2. Wed Jul 8 06:42:36 UTC 2020
3. ^C$ docker ps -a
4. CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
5. e4df7debb54a jpetazzo/clock "/bin/sh -c 'while d…" 2 minutes ago Exited (130) 2 minutes ago friendly\_shtern

We can see that this container has exited.

To overcome this problem, we are now going to run a container in the background.

Step 3: Run a new jpetazzo/clock container in the background.

1. $ docker run -d jpetazzo/clock
2. 98c34c8ebc4444fe664f431cf3be186ca831dfbdf6e088f6815e06f4bec8c81d

Now, instead of the output of the container, the container ID of the new container is displayed and the terminal is free for us to do any other task.

Step 4: Check the state of the container.

1. $ docker ps -a
2. CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
3. 98c34c8ebc44 jpetazzo/clock "/bin/sh -c 'while d…" 33 seconds ago Up 31 seconds serene\_fermi
4. e4df7debb54a jpetazzo/clock "/bin/sh -c 'while d…" 8 minutes ago Exited (130) 8 minutes ago friendly\_shtern

From the output, we can infer that the clock container is up and running (in the background). Any number of containers can be run in the background.

# Using docker ps command

<docker ps> command is used to list the containers in the Docker host and their details. In the previous demo, we have created a clock container that runs in the background. Start few more clock containers in the background to explore the usage of <docker ps> command.

Step 1: Start three more clock containers in the background

1. $ docker run -d jpetazzo/clock
2. 3fcbf015e73a514b06b6b0e76bc4d5a783ab0e9c354ae65919a02b7ad1195f33
3. $ docker run -d jpetazzo/clock
4. e37d1eb67ee5c9af2887e56332548be4f4e777bb6138ea63e0fa571edd3f9ff9
5. $ docker run -d jpetazzo/clock
6. cf14dbf927bc83ecf7b2137054c7da2f88033dfa4d6c72ebb9292303a0f5c4ae

Step 2: Based on the use case, execute the <docker ps> command to view the containers:

i) To list all the running containers:

1. $ docker ps
2. CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
3. cf14dbf927bc jpetazzo/clock "/bin/sh -c 'while d…" About a minute ago Up About a minute gifted\_hoover
4. e37d1eb67ee5 jpetazzo/clock "/bin/sh -c 'while d…" About a minute ago Up About a minute friendly\_boyd
5. 3fcbf015e73a jpetazzo/clock "/bin/sh -c 'while d…" About a minute ago Up About a minute upbeat\_visvesvaraya
6. 98c34c8ebc44 jpetazzo/clock "/bin/sh -c 'while d…" 5 minutes ago Up 5 minutes serene\_fermi

ii) To list all the containers irrespective of the state:

1. $ docker ps -a
2. CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
3. cf14dbf927bc jpetazzo/clock "/bin/sh -c 'while d…" 3 minutes ago Up 2 minutes gifted\_hoover
4. e37d1eb67ee5 jpetazzo/clock "/bin/sh -c 'while d…" 3 minutes ago Up 3 minutes friendly\_boyd
5. 3fcbf015e73a jpetazzo/clock "/bin/sh -c 'while d…" 3 minutes ago Up 3 minutes upbeat\_visvesvaraya
6. 98c34c8ebc44 jpetazzo/clock "/bin/sh -c 'while d…" 6 minutes ago Up 6 minutes serene\_fermi
7. e4df7debb54a jpetazzo/clock "/bin/sh -c 'while d…" 14 minutes ago Exited (130) 14 minutes ago friendly\_shtern

iii) To list the recently created container:

1. $ docker ps -l
2. CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
3. cf14dbf927bc jpetazzo/clock "/bin/sh -c 'while d…" 3 minutes ago Up 3 minutes gifted\_hoover

iv) To view only the IDs of the running containers:

1. $ docker ps -q
2. cf14dbf927bc
3. e37d1eb67ee5
4. 3fcbf015e73a
5. 98c34c8ebc44

v) To view the ID of the container that was started recently:

1. $ docker ps -lq
2. cf14dbf927bc

# Using docker logs command

Docker does not find any difference between running a container in the background or foreground. However, to fetch the logs of either a background/foreground container, docker offers the following utility.

Step 1:  Fetch the log of the recently started clock container by executing <docker logs> command. The container ID is given as the argument to <docker logs> command.

1. $ docker logs cf14dbf927bc
2. Wed Jul 8 06:53:45 UTC 2020
3. Wed Jul 8 06:53:46 UTC 2020
4. Wed Jul 8 06:53:47 UTC 2020
5. Wed Jul 8 06:53:48 UTC 2020
6. .
7. .

Step 2: Fetch the log of the same container dynamically by executing the following command.

1. $ docker logs --tail 5 --follow cf14dbf927bc
2. Wed Jul 8 07:04:07 UTC 2020
3. Wed Jul 8 07:04:08 UTC 2020
4. Wed Jul 8 07:04:09 UTC 2020
5. Wed Jul 8 07:04:10 UTC 2020
6. Wed Jul 8 07:04:11 UTC 2020
7. .
8. .

The above command dynamically prints the recent 5 logs of the container.

# Stopping a background container

Until now, we were using CTRL-C or CTRL-D to exit from the container that was launched with the interactive mode. Apparently, this technique doesn’t work with background containers.

There are two ways to stop a container running in the background.

* Using **docker kill** command: This command acts the same way like the Unix KILL signal. It terminates the container immediately.
* Using **docker stop** command: This command will send the Unix TERM signal to the container which will shut down the container gracefully. After 10 seconds, KILL signal is sent if the container was still running.

Let us try to stop containers with **docker kill** and **docker stop** command.

Step 1: List all the running containers.

1. $ docker ps
2. CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
3. cf14dbf927bc jpetazzo/clock "/bin/sh -c 'while d…" 28 minutes ago Up 28 minutes gifted\_hoover
4. e37d1eb67ee5 jpetazzo/clock "/bin/sh -c 'while d…" 28 minutes ago Up 28 minutes friendly\_boyd
5. 3fcbf015e73a jpetazzo/clock "/bin/sh -c 'while d…" 28 minutes ago Up 28 minutes upbeat\_visvesvaraya
6. 98c34c8ebc44 jpetazzo/clock "/bin/sh -c 'while d…" 31 minutes ago Up 31 minutes serene\_fermi

All of these containers are running in the background.

Step 2: Stop one of the containers using <docker stop> command.

1. $ docker stop cf14dbf927bc
2. cf14dbf927bc

Step 3: Check the list of running containers now.

1. $ docker ps
2. CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
3. e37d1eb67ee5 jpetazzo/clock "/bin/sh -c 'while d…" 30 minutes ago Up 30 minutes friendly\_boyd
4. 3fcbf015e73a jpetazzo/clock "/bin/sh -c 'while d…" 30 minutes ago Up 30 minutes upbeat\_visvesvaraya
5. 98c34c8ebc44 jpetazzo/clock "/bin/sh -c 'while d…" 33 minutes ago Up 33 minutes serene\_fermi

Step 4: Stop few containers using **<docker kill>** command.

1. $ docker kill e37d1eb67ee5
2. e37d1eb67ee5
3. $ docker kill 3fcbf015e73a
4. 3fcbf015e73a

Step 5: Check the list of running containers again.

1. $ docker ps
2. CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
3. 98c34c8ebc44 jpetazzo/clock "/bin/sh -c 'while d…" 36 minutes ago Up 36 minutes serene\_fermi

Only one container is running. We also noticed that stopping a container using **docker kill** command happened faster than while using **docker stop** command.

Step 6: List the stopped containers.

1. $ docker ps -a
2. CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
3. cf14dbf927bc jpetazzo/clock "/bin/sh -c 'while d…" 33 minutes ago Exited (137) 3 minutes ago gifted\_hoover
4. e37d1eb67ee5 jpetazzo/clock "/bin/sh -c 'while d…" 33 minutes ago Exited (137) About a minute ago friendly\_boyd
5. 3fcbf015e73a jpetazzo/clock "/bin/sh -c 'while d…" 34 minutes ago Exited (137) About a minute ago upbeat\_visvesvaraya
6. 98c34c8ebc44 jpetazzo/clock "/bin/sh -c 'while d…" 37 minutes ago Up 37 minutes serene\_fermi

Any container that is running in the background can be stopped either by executing <docker stop> or <docker kill> command.

# Detaching from a container

As mentioned earlier, it does not make any difference to docker whether a container runs in the background or foreground. It treats them all the same way. A container that is currently running in the interactive mode can be exited without actually stopping it. This is called detaching.

Detaching from a container running in the interactive mode is as easy as typing a key-sequence, which is CTRL-P + CTRL-Q.

In this demo, we will run a container in the interactive mode and  try to detach from it without stopping the container.

Step 1: Launch a CentOS container in the interactive mode.

1. $ docker run -it centos
2. [root@e62bdae9b7e5 /]#
3. [root@e62bdae9b7e5 /]#

Step 2: Now press the key sequence CTRL-P + CTRL-Q.

1. [root@e62bdae9b7e5 /]#
2. <<enter the Ctrl-P + Ctrl-Q key sequence>>
3. $

Step 3: We are now out from the container environment. But, let us now check the state of the container.

1. $ docker ps -a | grep e62bdae9b7e5

We can see that the container is still up and running. This behavior is apparently different from typing CTRL-C or CTRL-D to exit from the container, in which case, the container is terminated. The above steps can be followed to detach from a container running in the interactive mode.

# How do we detach from a container started without -it option?

In this case, the CTRL-P + CTRL-Q key sequence will not work. The other option of using CTRL-C or CTRL-D cannot be used, because they will terminate the container. These type of containers can be detached from by killing the docker client.

Step 1: Start a container without the –it option.

1. $ docker run jpetazzo/clock
2. Wed Jul 8 07:57:17 UTC 2020
3. Wed Jul 8 07:57:18 UTC 2020
4. .
5. .

Step 2: Open a new terminal and find the container ID and also find the Unix process ID of the docker client.

1. $ ps -ef | grep clock
2. root 73 1 0 08:29 pts/0 00:00:00 docker run jpetazzo/clock

Step 3: '73' is the process ID of the Docker client process. Kill this process using the <kill> command.

1. $ kill -9 73
2. $

Step 4: Go to the first terminal where the actual clock container is running and notice the output.

1. Wed Jul 8 08:33:05 UTC 2020
2. Wed Jul 8 08:33:06 UTC 2020
3. Wed Jul 8 08:33:07 UTC 2020
4. Killed

The terminal is now free. But we have just detached from the container, not stopped it.

Step 5: Check the logs of the clock container to verify that it is still running.

1. $ docker logs --tail 1 --follow de6c02be59f9
2. Wed Jul 8 08:37:15 UTC 2020
3. Wed Jul 8 08:37:16 UTC 2020
4. Wed Jul 8 08:37:17 UTC 2020
5. .
6. .
7. .

Thus, by killing the docker client, we can detach from containers started without the –it option. How do we connect to these detached containers that are still running?

# Detaching from a container

As mentioned earlier, it does not make any difference to docker whether a container runs in the background or foreground. It treats them all the same way. A container that is currently running in the interactive mode can be exited without actually stopping it. This is called detaching.

Detaching from a container running in the interactive mode is as easy as typing a key-sequence, which is CTRL-P + CTRL-Q.

In this demo, we will run a container in the interactive mode and  try to detach from it without stopping the container.

Step 1: Launch a CentOS container in the interactive mode.

1. $ docker run -it centos
2. [root@e62bdae9b7e5 /]#
3. [root@e62bdae9b7e5 /]#

Step 2: Now press the key sequence CTRL-P + CTRL-Q.

1. [root@e62bdae9b7e5 /]#
2. <<enter the Ctrl-P + Ctrl-Q key sequence>>
3. $

Step 3: We are now out from the container environment. But, let us now check the state of the container.

1. $ docker ps -a | grep e62bdae9b7e5

We can see that the container is still up and running. This behavior is apparently different from typing CTRL-C or CTRL-D to exit from the container, in which case, the container is terminated. The above steps can be followed to detach from a container running in the interactive mode.

# How do we detach from a container started without -it option?

In this case, the CTRL-P + CTRL-Q key sequence will not work. The other option of using CTRL-C or CTRL-D cannot be used, because they will terminate the container. These type of containers can be detached from by killing the docker client.

Step 1: Start a container without the –it option.

1. $ docker run jpetazzo/clock
2. Wed Jul 8 07:57:17 UTC 2020
3. Wed Jul 8 07:57:18 UTC 2020
4. .
5. .

Step 2: Open a new terminal and find the container ID and also find the Unix process ID of the docker client.

1. $ ps -ef | grep clock
2. root 73 1 0 08:29 pts/0 00:00:00 docker run jpetazzo/clock

Step 3: '73' is the process ID of the Docker client process. Kill this process using the <kill> command.

1. $ kill -9 73
2. $

Step 4: Go to the first terminal where the actual clock container is running and notice the output.

1. Wed Jul 8 08:33:05 UTC 2020
2. Wed Jul 8 08:33:06 UTC 2020
3. Wed Jul 8 08:33:07 UTC 2020
4. Killed

The terminal is now free. But we have just detached from the container, not stopped it.

Step 5: Check the logs of the clock container to verify that it is still running.

1. $ docker logs --tail 1 --follow de6c02be59f9
2. Wed Jul 8 08:37:15 UTC 2020
3. Wed Jul 8 08:37:16 UTC 2020
4. Wed Jul 8 08:37:17 UTC 2020
5. .
6. .
7. .

Thus, by killing the docker client, we can detach from containers started without the –it option. How do we connect to these detached containers that are still running?

# Attaching to a container

Any detached container will be a contained process inside the Docker host. We have to reattach to the container to interact with it or to give any inputs. Execute the following command to attach to a container which is in the running state.

In this demo, we will attach to the clock container which has been detached in the previous demo.

Step 1: Attach to the clock container using the <docker attach> command.

1. $ docker attach de6c02be59f9
2. Wed Jul 8 08:41:30 UTC 2020
3. Wed Jul 8 08:41:31 UTC 2020
4. Wed Jul 8 08:41:32 UTC 2020

Note: You cannot attach to a container that is not in the running state. Multiple attachments to a container can be made from different sessions.

Step 2: Stop the container by pressing Ctrl+C.

1. Wed Jul 8 08:43:36 UTC 2020
2. Wed Jul 8 08:43:37 UTC 2020
3. << press Ctrl+C>>
4. $

Step 3: Now, start the clock container which has been stopped. Any container which had been stopped, can be restarted by using the following command:

1. $ docker start de6c02be59f9
2. de6c02be59f9

The container is restarted with the same options we launched it with.

Step 4: Check the status of the container. The container must be in the running state.

1. $ docker ps -a
2. CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
3. de6c02be59f9 jpetazzo/clock "/bin/sh -c 'while d…" 17 minutes ago Up About a minute romantic\_villani

When we list the containers using “docker ps” command, along with the container id, image and other information, the name of the container also gets displayed. How are these names set? Can we give a name of our choice to the container?  We will deal with this aspect in the upcoming demo.

# Naming a container

Naming a container can come in handy in times when there are more number of the same container running. Eg., 15 CentOS containers. In this case, it is strenuous to remember the container ID of all the containers. By default, docker names the container with a random name in the adjective\_name form.

Observe the snippet below to look at some of the names given by docker to the containers.

1. $ docker ps -a
2. CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
3. cf14dbf927bc jpetazzo/clock "/bin/sh -c 'while d…" 33 minutes ago Exited (137) 3 minutes ago gifted\_hoover
4. e37d1eb67ee5 jpetazzo/clock "/bin/sh -c 'while d…" 33 minutes ago Exited (137) About a minute ago friendly\_boyd
5. 3fcbf015e73a jpetazzo/clock "/bin/sh -c 'while d…" 34 minutes ago Exited (137) About a minute ago upbeat\_visvesvaraya
6. 98c34c8ebc44 jpetazzo/clock "/bin/sh -c 'while d…" 37 minutes ago Up 37 minutes serene\_fermi

Docker also allows us to name containers with names of our choice.

In this demo, we will launch a container with a name of our choice.

Step 1: Use the --name option with <docker run> command to provide the name we want to give our container. Run a jpetazzo/clock container with the name "timeclock".

1. $ docker run -d --name timeclock jpetazzo/clock
2. 0a1e7eebb32a1d79b3a4162220713f0f0a8e27514ab5d7d6d73bd52a3848f328

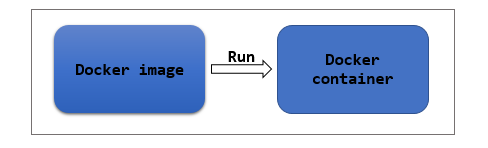
Step 2: Check the list of containers to verify the name.

1. $ docker ps| grep timeclock
2. 0a1e7eebb32a jpetazzo/clock "/bin/sh -c 'while d…" About a minute ago Up About a minute timeclock

The container names can also be used to refer to the container, apart from container IDs.

# Docker images

Docker images are building blocks of containers. They contain the set of instructions necessary for creating the containers. Containers are born when the docker images are run. Essentially, a container is a running instance of a docker image.



Imagine, we want to create an Nginx container. To create a container, we need the necessary Docker image. This image can be accessed from a Docker registry(either Docker Hub or a self-hosted registry). For this purpose, we create a running instance of the Docker image by executing the following command.

1. *# docker run nginx*

In the above command, nginx is the name of the image used to create the Nginx container. So it is rightly said that docker images are building blocks of docker containers.

In essence, a Docker image is a bundle of files and metadata required to create a fully functional container environment which might include the filesystem, installation code, applications and dependencies, etc.,

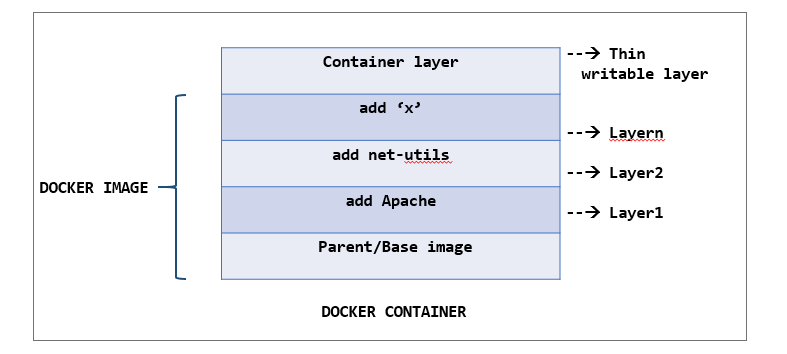
# What are Docker images made up of?

Docker images are made up of layers, one stacked on the other conceptually. Each layer refers to certain instructions that was given while creating the Docker image (through a Docker file).

In fact, when we built our first container, we noticed the images being pulled from the registry in the form of layers.

Docker images are mostly built from a parent image (maybe a Debian, CentOS, Ubuntu etc.,) of our requirement. Over the parent image, the changes are made in terms of layers. The instructions mentioned in the Docker file are executed one by one on the parent image in the form of layers in order to form the final Docker image.

Each newly stacked layer references the layer beneath it.



Docker images can also be built using a base image, meaning you build your image from the scratch. This way, you can completely control the content of your image. But, building a Docker image from scratch is considered to be more time-consuming and complex for beginners.

Note: However, the terms 'parent image' and 'base image' are being used interchangeably.

# Immutability of Docker images

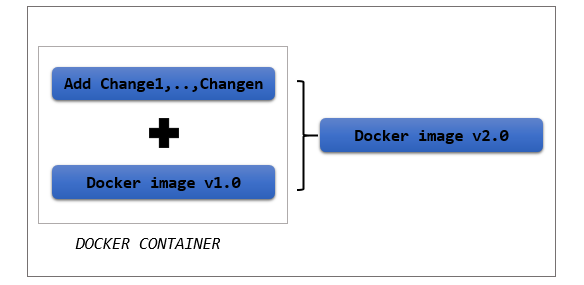
A Docker image once built, it immutable because it is only a read-only filesystem. So, how can we make changes to the existing image?

When a Docker container is instantiated by running a Docker image, Docker adds a thin writable layer which stores all the changes made to the container during the runtime. This is called as the container layer. This container layer is difference between the original Docker image used to build the container and the live container.

Since Docker images are immutable, any number of containers can be built from the same image while those containers can maintain their individual states without any change.

In essence, we don’t make changes to the existing image. Instead,

* We build a container out of the image.
* Make the necessary changes.
* Once the changes are complete, we convert it into a new layer.
* A new image is now created by adding this new layer to the existing image.



# Types of Docker registries

By default, DockerHub is the registry used when installing a Docker engine. But there are other forms as well in which we can  Docker registry.

1. Docker hub- It is the official image resource owned Docker. It contains enormous number of container images. These are the images that might have been shared by Docker’s user community, vendors or any open source projects. The Docker hub can also be used to host and manage your own Docker images. Docker hub allows us to use both Public/Private registry based on our requirement. Access https://hub.docker.com/ to dive into the official image resource of Docker.
2. Self-hosted registry- Use a self-hosted registry if you want to manage your Docker images within your organization’s infrastructure. Use a self-hosted registry when you want to  This can be done by deploying a registry server.
3. Third party registry- Apart from DockerHub, third-party registries like AWS EC2 Container registry, Google registry , etc., can also be used.

# Managing Docker images

The Docker images once created can be managed and made available to other users (either public or private) through Docker registries. Docker registries are used for versioning, managing and sharing of Docker images through a single image resource. Docker registries are a preferred mechanism of managing images by enterprises for the following reasons:

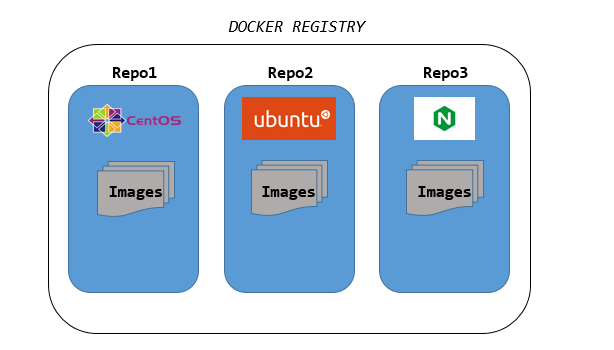
1. Better collaboration with team.

2. Security of the images is ensured by defining who can access the image and who cannot since enterprise data is highly sensitive and needs to be secured all the time.

3. Distribution of Docker images is made easy by Docker registry. Operations as simple as push and pull is sufficient to upload and download images from the registry.

4. Easier to share your image with the whole world.

Inside the registry, remains the repository which is the actual physical location where the images are stored. A Docker repository is the namespace used to store your Docker image. It usually contains the different versions of the same tags. So, when you host a Docker image in the Docker registry, it actually gets stored in the respective repository where we are pushing it to.



Here, Repo1 contains all the CentOS images with different versions/tags.

# Namespaces of Docker images

The Docker images can be pushed/pulled from/to a Docker registry using the Docker client. There are three kinds of namespaces used while referring to a Docker image.

1. Official images- Name of the image is used.

Eg., CentOS, Ubuntu, etc.,

2. Images inside a repository- <repository\_name>/<image\_name>

Eg., jpetazzo/clock, weaveworks/scope

3. Images from a self-hosted registry - <path\_of\_the \_registry>:<port>/<repository\_name>/<image\_name>

Eg., registry.mylab.com:5000/my-private/image

# Working with Docker images

Docker client deals with the Docker images. Docker client pulls an image from the registry, pushes an image to the registry, tags an image, delete/remove, etc.,  A docker image when created will be given an ID, which can be used to refer to the image in future.

The docker command is used with the following . Using it with appropriate arguments and command line options lets us work with docker images seamlessly.

1. To list all the images present in the Docker engine:

1. $ docker images

2. To search for a particular image from the Docker registry (By default, DockerHub is searched):

1. $ docker search nginx

3. To download an image from the Docker registry:

1. $ docker pull nginx

4. To push a Docker image into the Docker registry:

1. $ docker push newubuntu

# Working with Docker images

As discussed, it is very clear that a container is the running instance of a docker image. Let us assume that we want to run a CentOS container.  There are two cases:

i) The image required is present: In this case, the container can be simply run using the "docker run" command.

ii) The image required is not present: In this case, the container can be run,

1. By either pulling the image first and then running the container using "docker run" command or
2. By executing the "docker run" command directly. The docker run command will search for the image locally. If not found, it automatically pulls the image from the registry and then runs the container

In this demo, let us assume that the image required is not present.

Step 1: List all the images available in the Docker host to check whether CentOS is present.

1. $ docker images
2. REPOSITORY TAG IMAGE ID CREATED SIZE
3. ubuntu latest 16508e5c265d 22 months ago 84.1MB
4. redis latest 4e8db158f18d 23 months ago 83.4MB
5. weaveworks/scope 1.9.1 4b07159e407b 23 months ago 68MB
6. alpine latest 11cd0b38bc3c 24 months ago 4.41MB

Step 2: Search for all the images in the Docker registry with the keyword "centos" in their names. Docker Hub also provides the description of each image present.

1. $ docker search centos
2. NAME DESCRIPTION STARS OFFICIAL AUTOMATED
3. centos The official build of CentOS. 6064 [OK]
4. ansible/centos7-ansible Ansible on Centos7 130 [OK]
5. consol/centos-xfce-vnc Centos container with "headless" VNC session… 116 [OK]
6. .
7. .
8. .
9. .
10. pivotaldata/centos7-dev CentosOS 7 image for GPDB development 0
11. pivotaldata/centos6.8-dev CentosOS 6.8 image for GPDB development 0
12. smartentry/centos centos with smartentry 0 [OK]

Step 3: Choose the appropriate image and pull it from the registry.

If no tags are mentioned, by default, the image with "latest" tag gets pulled. More on images and tags, later.

1. $ docker pull centos [OK]
2. Using default tag: latest [OK]
3. latest: Pulling from library/centos [OK]
4. 6910e5a164f7: Pull complete [OK]
5. Digest: sha256:4062bbdd1bb0801b0aa38e0f83dece70fb7a5e9bce223423a68de2d8b784b43b
6. Status: Downloaded newer image for centos:latest

Step 4: List the images to check whether "centos" image is now present in the Docker host.

1. $ docker images
2. REPOSITORY TAG IMAGE ID CREATED SIZE
3. centos latest 831691599b88 12 days ago 215MB [OK]
4. ubuntu latest 16508e5c265d 22 months ago 84.1MB
5. redis latest 4e8db158f18d 23 months ago 83.4MB [OK]
6. weaveworks/scope 1.9.1 4b07159e407b 23 months ago 68MB [OK]
7. alpine latest 11cd0b38bc3c 24 months ago 4.41MB

Step 5: Now that we have the image in our Docker host, we can run the "centos" container.

1. $ docker run -it centos
2. [root@a9e73acbecd8 /]#

The "centos" container is successfully created.

Now, Let us create a "debian" container. From the list of images displayed in "Step 4", it is clear that, there is no "debian" image present in the Docker host. Instead of pulling the image first, let us run the container directly.

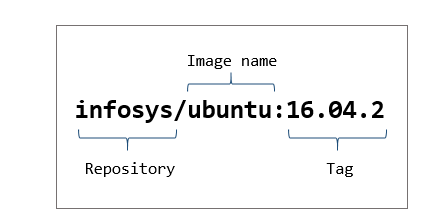
1. $ docker run -it debian
2. Unable to find image 'debian:latest' locally
3. latest: Pulling from library/debiane9afc4f90ab0: Pull complete
4. Digest: sha256:46d659005ca1151087efa997f1039ae45a7bf7a2cbbe2d17d3dcbda632a3ee9a
5. Status: Downloaded newer image for debian:latest
6. root@ba308272c106:/#

You can notice the image being pulled layer by layer. Once the pull was complete, a new container was started. List the containers to see a "debian" container present.

1. $ docker ps -a
2. CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
3. ba308272c106 debian "bash" 2 minutes ago Exited (0) About a minute ago awesome\_kowalevski
4. a9e73acbecd8 centos "/bin/bash" 12 minutes ago Exited (0) 12 minutes ago priceless\_fermat

# Images and tags

In the last demo, when we pulled a "centos" image from the registry, it automatically pulls the image with tag "latest". "Latest" is the default tag assumed by Docker engine when no tag is mentioned while pulling an image. But the "latest" tag doesn't really reflect the same meaning as it looks like. More on the "latest" tag later.



Docker images are referred using their unique image IDs or names. A sample image ID might look like "16508e5c265d". But if you want to access an image with a more relevant name rather than either a broad name like "ubuntu" or a complex image ID like, "16508e5c265d", Docker tags will help you do it.

Using the docker tag command gives us the following scope:

1. We are able to create an alias name for an image.
2. We are able to perform version/variant controlling over docker images.

For example, Look at the list of images present in the Docker host currently.

1. $ docker images
2. REPOSITORY TAG IMAGE ID CREATED SIZE
3. ubuntu latest 16508e5c265d 22 months ago 84.1MB
4. redis latest 4e8db158f18d 23 months ago 83.4MB
5. weaveworks/scope 1.9.1 4b07159e407b 23 months ago 68MB
6. alpine latest 11cd0b38bc3c 24 months ago 4.41MB

Suppose I am using the weaveworks/scope image for my "Application1" , I might want to name it something like "imageforapp1". This is it is more relevant and also easier to remember.

Once the tagging is done, we will see that an alias name was given to the image weaveworks/scope.

1. $ docker images
2. REPOSITORY TAG IMAGE ID CREATED SIZE
3. ubuntu latest 16508e5c265d 22 months ago 84.1MB
4. redis latest 4e8db158f18d 23 months ago 83.4MB
5. imageforapp1 latest 4b07159e407b 23 months ago 68MB
6. weaveworks/scope 1.9.1 4b07159e407b 23 months ago 68MB
7. alpine latest 11cd0b38bc3c 24 months ago 4.41MB

As we can see, "imageforapp1" alias was created and notice the "latest" tag given to it. If we do not mention a tag explicitly for an image, "latest" is the default tag assigned to it by the Docker engine.

Note that the "latest" tag does not always refer to the "latest" version of the particular image. It is the image that was named without any tag.

However tagging an image makes more sense, when we have a newer version of the image with some updates and we want the users to be aware of what they are getting into before downloading it.

Note: Notice that the image ID for weaveworks/scope and imageforapp1 have the same image ID, implying that both are same images with different names.

# Tag an image

"docker tag" command is used to tag docker images. Pull the "centos:6.8" image from the registry. Let us tag the "centos:6.8" image in our docker host. Assume that we are going to add few changes to the "centos:6.8" image step by step and tag it with the appropriate name.

Step 1: List the available images.

1. $ docker images
2. REPOSITORY TAG IMAGE ID CREATED SIZE
3. centos 6.8 82f3b5f3c58f 15 months ago 195MB
4. ubuntu latest 16508e5c265d 22 months ago 84.1MB
5. redis latest 4e8db158f18d 23 months ago 83.4MB
6. weaveworks/scope 1.9.1 4b07159e407b 23 months ago 68MB
7. alpine latest 11cd0b38bc3c 24 months ago 4.41MB

Step 2: First tag the centos:6.8 image as "labimage1" since we are going to use it for the lab.

1. $ docker tag centos:6.8 centos:labimage1
2. $ docker images
3. REPOSITORY TAG IMAGE ID CREATED SIZE
4. centos 6.8 82f3b5f3c58f 15 months ago 195MB
5. centos labimage1 82f3b5f3c58f 15 months ago 195MB
6. redis latest 4760dc956b2d 2 years ago 107MB
7. ubuntu latest f975c5035748 2 years ago 112MB
8. alpine latest 3fd9065eaf02 2 years ago 4.14MB

Step 3: Assume we have created a second version of lab image and now, we want to tag it as "labimage2".

1. $ docker tag centos:6.8 centos:labimage2
2. $ docker images
3. REPOSITORY TAG IMAGE ID CREATED SIZE
4. centos 6.8 82f3b5f3c58f 15 months ago 195MB
5. centos labimage1 82f3b5f3c58f 15 months ago 195MB
6. centos labimage2 82f3b5f3c58f 15 months ago 195MB
7. .
8. .
9. .

What if we don't specify the tag?

Step 4: The third set of changes has been made to centos:6.8 image. This time let us not specify any tag.

1. $ docker tag centos:6.8 centos
2. $ docker images
3. REPOSITORY TAG IMAGE ID CREATED SIZE
4. centos 6.8 82f3b5f3c58f 15 months ago 195MB
5. centos labimage1 82f3b5f3c58f 15 months ago 195MB
6. centos labimage2 82f3b5f3c58f 15 months ago 195MB
7. centos latest 82f3b5f3c58f 15 months ago 195MB
8. .
9. .

We could see that, when no tag has been specified, the image will be by default tagged as "latest" whereas it need not necessarily be the latest version of the image. For example, let us create the most recent version of centos:6.8 and tag it as centos:labimage4

1. $ docker tag centos:6.8 centos:labimage4
2. $ docker images
3. REPOSITORY TAG IMAGE ID CREATED SIZE
4. centos 6.8 82f3b5f3c58f 15 months ago 195MB
5. centos labimage1 82f3b5f3c58f 15 months ago 195MB
6. centos labimage2 82f3b5f3c58f 15 months ago 195MB
7. centos labimage4 82f3b5f3c58f 15 months ago 195MB
8. centos latest 82f3b5f3c58f 15 months ago 195MB
9. .
10. .

Evidently, centos:labimage4 is the latest version of the "centos:6.8" image.

Let us try to run a "centos" container with the latest version of the "centos" image. Any user who is not familiar with how the versions of "centos" images has been named would try to run something as below.

1. $ docker run -it centos:latest
2. [root@910c826a2cfc /]# exit
3. exit
4. $ docker ps -a
5. CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
6. 910c826a2cfc centos:latest "/bin/bash" 25 seconds ago Exited (0) 15 seconds ago infallible\_bardeen

The Docker engine chose the "centos:latest" image to run the container. But the user would have expected that the actual latest version of the "centos" image might be chosen.

This is why tagging is important. Through tagging, the users are sure what version of the image they are getting themselves. Let us run the container with the required image by specifying the appropriate tag name.

1. $ docker run -it centos:labimage4
2. [root@8c6daa016ac5 /]# exit
3. exit
4. $ docker ps -a
5. CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
6. 8c6daa016ac5 centos:labimage4 "/bin/bash" About a minute ago Exited (0) 59 seconds ago suspicious\_villani
7. 910c826a2cfc centos:latest "/bin/bash" 4 minutes ago Exited (0) 4 minutes ago infallible\_bardeen

Note: It is always advisable to use the "latest" tag only to the latest version of the image considering effective versioning and ease of usage among the users in the organisation.

# Untag an image

Removing the tag of a docker image is as simple as assigning one. When you untag a docker image, the tag gets removed. The image will now be present only with its original tag.

Step 1: List the docker image.

1. $ docker images
2. REPOSITORY TAG IMAGE ID CREATED SIZE
3. centos latest 82f3b5f3c58f 15 months ago 195MB
4. redis latest 4760dc956b2d 2 years ago 107MB
5. ubuntu latest f975c5035748 2 years ago 112MB
6. alpine latest 3fd9065eaf02 2 years ago 4.14MB

Step 2: Assign a new tag for centos:latest image.

1. $ docker tag centos:latest centos:new
2. $ docker images
3. REPOSITORY TAG IMAGE ID CREATED SIZE
4. centos latest 82f3b5f3c58f 15 months ago 195MB
5. centos new 82f3b5f3c58f 15 months ago 195MB
6. redis latest 4760dc956b2d 2 years ago 107MB
7. ubuntu latest f975c5035748 2 years ago 112MB
8. alpine latest 3fd9065eaf02 2 years ago 4.14MB

Step 3: Remove the tag given to the image.

1. $ docker rmi centos:new
2. Untagged: centos:new

Step 4: List the images now to check the "centos" image being present with its original tag.

1. $ docker images
2. REPOSITORY TAG IMAGE ID CREATED SIZE
3. centos latest 82f3b5f3c58f 15 months ago 195MB
4. redis latest 4760dc956b2d 2 years ago 107MB
5. ubuntu latest f975c5035748 2 years ago 112MB
6. alpine latest 3fd9065eaf02 2 years ago 4.14MB

# Building a docker image

Till the last demo, we were using the readily available docker images present in the Docker Hub. Some of those images are official images. Others are images created by Docker users for specific purposes. Whenever we are posed with requirements that are new, we cater to it by building a new solution. Same way, when we want a Docker image which can readily be used for a new purpose, we can easily build it.

Docker images can be built in two ways:

i) Interactive way( Create an image from existing container):

In this case, we start a container with an existing image. Doing this will create a thin writable layer over the container. Now, the changes can be customized inside the container, which will be written in the writable layer. From this, we can build a new Docker image.

ii) Using a Docker file:

A Docker file contains the instructions required to build the Docker image. There are two ways in which you can create Docker image using Docker file.

1. We can specify a parent image and specify the changes we want to make to it. These changes will be made in the form of layers. Finally, we can build our own Docker image.
2. We can create a fresh Docker image from scratch using the Docker file.

# Docker file

A Dockerfile is a simple configuration file following a specific syntax. The Dockerfile contains the step by step instructions that are necessary to build a docker image. These instructions are available as commands that need to be run to assemble the final Docker image. Each of the instructions are added in the form of layers and each layer is the delta of the previous layer.

We can create a Dockerfile by easily creating a text file and editing it using a text editor of our choice. "Dockerfile" is the default name for the file used widely. But it can be named anything of our choice. A sample Docker file where a docker image is built from a parent image is shown below:

1. FROM ubuntu:18.04
2. COPY . /app
3. RUN make /app
4. CMD python /app/app.py

Each of the above instructions create a new layer. Let us break down the Dockerfile line by line.

FROM ubuntu:18.04

FROM is a keyword used to specify the parent image on which we are going to make changes to build our new Docker image. Here, an Ubuntu image of version 18.04 is used as the parent image.

COPY . /app

COPY is used to copy files/ directories from a source to target. This is used to put the required files in the Docker image from out host.

RUN make /app

RUN keyword lets us run different commands inside our Docker image. Multiple commands can be broken down into different RUN statements in order to improve readability of the Docker file.

CMD python /app/app.py

CMD keywork allows us to specify a default command to run once the container is run using this image. Multiple CMD statements can be specified but only the last one will take effect.

The "docker build" command is used to build an image from the Dockerfile.

# Building your own Docker image

Imagine we are setting up a lab environment for a set of trainees to practice on how to configure an Apache webserver. There is no proper details about the Linux distribution the trainees have in their laptops. So we have decided to share a docker image for this purpose, which when run automatically spins up a CentOS container, appropriate for practicing web server configuration.

In such scenario, choosing to build an image from scratch is going to be unnecessarily additional work since we have numerous CentOS images available in the registry. So, ruling out this choice, we have two more choices left in hands:

1. Launch a centos container and make the necessary changes to it, which will be written in the writable layer of the container and build a new docker image out of it.
2. Use a Dockerfile by specifying a parent image, add the instructions and then build the image.

Before diving into writing our own Dockerfile, we will try to build a Docker image interactively.

# Build an image interactively

Let us build our Docker image that has to be shared with the trainees, in order for them to be able to practice webserver configuration with ease. Follow the below steps to do the same.

Step 1: Make sure you have the parent image in the Docker host. In this case, it is CentOS 8. If not, pull one from the registry.

1. $ docker pull centos:8
2. 8: Pulling from library/centos
3. 6910e5a164f7: Pull complete
4. Digest: sha256:4062bbdd1bb0801b0aa38e0f83dece70fb7a5e9bce223423a68de2d8b784b43b
5. Status: Downloaded newer image for centos:8
6. $ docker images
7. REPOSITORY TAG IMAGE ID CREATED SIZE
8. centos 8 831691599b88 13 days ago 215MB
9. redis latest 4760dc956b2d 2 years ago 107MB
10. ubuntu latest f975c5035748 2 years ago 112MB
11. alpine latest 3fd9065eaf02 2 years ago 4.14MB

Step 2: Launch a CentOS:8 container.

1. $ docker run -it centos:8
2. [root@4914a0cea58c /]#

Step 3: As matter of fact, we want the Apache web server package to be readily installed for the trainees to practice. Check whether "httpd" package is installed.

1. [root@4914a0cea58c /]# rpm -q httpd
2. package httpd is not installed

Step 4: Install the "httpd" package in the container using the yum repository. (Make sure the yum repository is configured.)

1. [root@4914a0cea58c /]# yum install -y httpd
2. .
3. <snip>
4. .
5. Installed:
6. apr-1.6.3-9.el8.x86\_64 apr-util-1.6.1-6.el8.x86\_64
7. apr-util-bdb-1.6.1-6.el8.x86\_64 apr-util-openssl-1.6.1-6.el8.x86\_64
8. brotli-1.0.6-1.el8.x86\_64 centos-logos-httpd-80.5-2.el8.noarch
9. httpd-2.4.37-21.module\_el8.2.0+382+15b0afa8.x86\_64 httpd-filesystem-2.4.37-21.module\_el8.2.0+382+15b0afa8.noarch
10. httpd-tools-2.4.37-21.module\_el8.2.0+382+15b0afa8.x86\_64 mailcap-2.1.48-3.el8.noarch
11. mod\_http2-1.11.3-3.module\_el8.2.0+307+4d18d695.x86\_64
12. Complete!

Step 5: Check for the installed package.

1. [root@4914a0cea58c /]# rpm -q httpd 10/11
2. httpd-2.4.37-21.module\_el8.2.0+382+15b0afa8.x86\_64

The httpd package is successfully installed.

This change done inside the container is written on the writable layer of the container.

Step 6: To inspect the changes made to the filesystem of the container, execute the "docker diff <container\_id>". This gives the difference between the container's current filesystem and the original image. Exit from the container before executing the command.

1. [root@4914a0cea58c /]# exit
2. exit
3. $ docker diff 4914a0cea58c 2/11
4. C /etc 3/11
5. C /etc/gshadow 4/11
6. .
7. .
8. A /var/log/dnf.librepo.log
9. A /var/log/dnf.log
10. A /var/log/httpd

We can see a list of files/directories prefixed with either C or A.

C - indicates files/directories that have been changed.

A - indicates files/directories that have been added.

Step 7: To add our changes as a new layer and to create a new image with this added layer, execute "docker commit" command. The output of this command is the image ID of our new Docker image.

1. $ docker commit 4914a0cea58c
2. sha256:9ac3092f21147e8ff78ba2dfa38ccdd0b179ba96bdacf0f718510177be2ed2bb

We have successfully created our own first docker image. The big alphanumeric output is the ID of the new image.

Step 8: List out the available image to see whether the new image has been added to the lot.

1. $ docker images
2. REPOSITORY TAG IMAGE ID CREATED SIZE
3. <none> <none> 9ac3092f2114 2 minutes ago 254MB
4. centos 8 831691599b88 13 days ago 215MB
5. redis latest 4760dc956b2d 2 years ago 107MB
6. ubuntu latest f975c5035748 2 years ago 112MB
7. alpine latest 3fd9065eaf02 2 years ago 4.14MB

Step 9: Name the newly created docker image with an appropriate tag.

1. $ docker tag 9ac3092f2114 centoswithhttpd:1
2. $ docker images
3. REPOSITORY TAG IMAGE ID CREATED SIZE
4. centoswithhttpd 1 9ac3092f2114 5 minutes ago 254MB
5. centos 8 831691599b88 13 days ago 215MB
6. redis latest 4760dc956b2d 2 years ago 107MB
7. ubuntu latest f975c5035748 2 years ago 112MB
8. alpine latest 3fd9065eaf02 2 years ago 4.14MB

Step 10: Launch a container with the new image to see the changes.

1. $ docker run -it centoswithhttpd:1
2. [root@cae890485c74 /]# rpm -q httpd
3. httpd-2.4.37-21.module\_el8.2.0+382+15b0afa8.x86\_64

As we can see, the new centos image has "httpd" pre-installed in it unlike the base image.

The image is now ready to be shared with trainees for practice. Instead of sharing it individually, this image can be uploaded to a private registry where the users with access, can use this image. More on this later.

# Build an image using Dockerfile

The docker image created in the previous demo involved human interaction with the container. In case there are loads of complex changes to be incorporated into the container, it can get really hard to do so without any errors. In order to automate the process of creating a docker image, Dockerfile can be used. The FROM keyword in the Dockerfile decides whether we are creating a new image based on a parent image or we are creating a brand new docker image from scratch.

FROM centos:8 => indicates that we are going to build the new docker image with the base image as "centos:8".

FROM scratch => indicates that we are going to build an image from scratch.

Our Dockerfile is going to contain a series of instructions that will build our final docker image.

Step 1: The Dockerfile must be created in a new empty directory. Create a directory "centoswithhttpd". Under it, create a text file with the name, "Dockerfile".

1. $ mkdir centoswithhttpd
2. $ cd centoswithhttpd
3. $ touch Dockerfile
4. $ ls
5. Dockerfile

Step 2: Add the content to the Dockerfile using the VI editor(or any editor of your choice) as shown below.

1. $ vi Dockerfile
2. $ cat Dockerfile
3. FROM centos:8
4. RUN yum install -y httpd

Step 3: Now that we have instructed the steps to create the image, let us go ahead and build the image.

1. $ docker build -t centoswithhttpd:1 .
2. Sending build context to Docker daemon 2.048kB
3. Step 1/2 : FROM centos:8
4. ---> 831691599b88
5. Step 2/2 : RUN yum install -y httpd
6. ---> Running in 62163df9c424
7. CentOS-8 - AppStream 9.8 MB/s | 5.8 MB 00:00
8. CentOS-8 - Base 5.5 MB/s | 2.2 MB 00:00
9. CentOS-8 - Extras 22 kB/s | 6.7 kB 00:00
10. Last metadata expiration check: 0:00:01 ago on Tue Jun 30 11:30:21 2020.
11. Dependencies resolved.
12. ================================================================================
13. Package Arch Version Repo Size
14. ================================================================================
15. Installing:
16. httpd x86\_64 2.4.37-21.module\_el8.2.0+382+15b0afa8 AppStream 1.7 M
17. Installing dependencies:
18. apr x86\_64 1.6.3-9.el8 AppStream 125 k
19. apr-util x86\_64 1.6.1-6.el8 AppStream 105 k
20. brotli x86\_64 1.0.6-1.el8 BaseOS 323 k
21. centos-logos-httpd
22. noarch 80.5-2.el8 BaseOS 24 k
23. httpd-filesystem noarch 2.4.37-21.module\_el8.2.0+382+15b0afa8 AppStream 36 k
24. httpd-tools x86\_64 2.4.37-21.module\_el8.2.0+382+15b0afa8 AppStream 103 k
25. mailcap noarch 2.1.48-3.el8 BaseOS 39 k
26. mod\_http2 x86\_64 1.11.3-3.module\_el8.2.0+307+4d18d695 AppStream 157 k
27. Installing weak dependencies:
28. apr-util-bdb x86\_64 1.6.1-6.el8 AppStream 25 k
29. apr-util-openssl x86\_64 1.6.1-6.el8 AppStream 27 k
30. Enabling module streams:
31. httpd 2.4
32. Transaction Summary
33. ================================================================================
34. Install 11 Packages
35. Total download size: 2.6 M
36. Installed size: 7.5 M
37. Downloading Packages:
38. (1/11): apr-util-bdb-1.6.1-6.el8.x86\_64.rpm 456 kB/s | 25 kB 00:00
39. (2/11): apr-1.6.3-9.el8.x86\_64.rpm 1.5 MB/s | 125 kB 00:00
40. (3/11): apr-util-1.6.1-6.el8.x86\_64.rpm 975 kB/s | 105 kB 00:00
41. (4/11): apr-util-openssl-1.6.1-6.el8.x86\_64.rpm 424 kB/s | 27 kB 00:00
42. (5/11): httpd-filesystem-2.4.37-21.module\_el8.2 528 kB/s | 36 kB 00:00
43. (6/11): httpd-tools-2.4.37-21.module\_el8.2.0+38 1.0 MB/s | 103 kB 00:00
44. (7/11): mod\_http2-1.11.3-3.module\_el8.2.0+307+4 1.9 MB/s | 157 kB 00:00
45. (8/11): centos-logos-httpd-80.5-2.el8.noarch.rp 825 kB/s | 24 kB 00:00
46. (9/11): mailcap-2.1.48-3.el8.noarch.rpm 637 kB/s | 39 kB 00:00
47. (10/11): httpd-2.4.37-21.module\_el8.2.0+382+15b 5.1 MB/s | 1.7 MB 00:00
48. (11/11): brotli-1.0.6-1.el8.x86\_64.rpm 1.7 MB/s | 323 kB 00:00
49. --------------------------------------------------------------------------------
50. Total 2.6 MB/s | 2.6 MB 00:01
51. warning: /var/cache/dnf/AppStream-02e86d1c976ab532/packages/apr-1.6.3-9.el8.x86\_64.rpm: Header V3 RSA/SHA256 Signature, key ID 8483c65d: NOKEY
52. CentOS-8 - AppStream 1.6 MB/s | 1.6 kB 00:00
53. Importing GPG key 0x8483C65D:
54. Userid : "CentOS (CentOS Official Signing Key) <security@centos.org>"
55. Fingerprint: 99DB 70FA E1D7 CE22 7FB6 4882 05B5 55B3 8483 C65D
56. From : /etc/pki/rpm-gpg/RPM-GPG-KEY-centosofficial
57. Key imported successfully
58. Running transaction check
59. Transaction check succeeded.
60. Running transaction test
61. Transaction test succeeded.
62. Running transaction
63. Preparing : 1/1
64. Installing : apr-1.6.3-9.el8.x86\_64 1/11
65. Running scriptlet: apr-1.6.3-9.el8.x86\_64 1/11
66. Installing : apr-util-bdb-1.6.1-6.el8.x86\_64 2/11
67. Installing : apr-util-openssl-1.6.1-6.el8.x86\_64 3/11
68. Installing : apr-util-1.6.1-6.el8.x86\_64 4/11
69. Running scriptlet: apr-util-1.6.1-6.el8.x86\_64 4/11
70. Installing : httpd-tools-2.4.37-21.module\_el8.2.0+382+15b0afa8. 5/11
71. Installing : mailcap-2.1.48-3.el8.noarch 6/11
72. Installing : centos-logos-httpd-80.5-2.el8.noarch 7/11
73. Installing : brotli-1.0.6-1.el8.x86\_64 8/11
74. Running scriptlet: httpd-filesystem-2.4.37-21.module\_el8.2.0+382+15b0 9/11
75. Installing : httpd-filesystem-2.4.37-21.module\_el8.2.0+382+15b0 9/11
76. Installing : mod\_http2-1.11.3-3.module\_el8.2.0+307+4d18d695.x86 10/11
77. Installing : httpd-2.4.37-21.module\_el8.2.0+382+15b0afa8.x86\_64 11/11
78. Running scriptlet: httpd-2.4.37-21.module\_el8.2.0+382+15b0afa8.x86\_64 11/11
79. Verifying : apr-1.6.3-9.el8.x86\_64 1/11
80. Verifying : apr-util-1.6.1-6.el8.x86\_64 2/11
81. Verifying : apr-util-bdb-1.6.1-6.el8.x86\_64 3/11
82. Verifying : apr-util-openssl-1.6.1-6.el8.x86\_64 4/11
83. Verifying : httpd-2.4.37-21.module\_el8.2.0+382+15b0afa8.x86\_64 5/11
84. Verifying : httpd-filesystem-2.4.37-21.module\_el8.2.0+382+15b0 6/11
85. Verifying : httpd-tools-2.4.37-21.module\_el8.2.0+382+15b0afa8. 7/11
86. Verifying : mod\_http2-1.11.3-3.module\_el8.2.0+307+4d18d695.x86 8/11
87. Verifying : brotli-1.0.6-1.el8.x86\_64 9/11
88. Verifying : centos-logos-httpd-80.5-2.el8.noarch 10/11
89. Verifying : mailcap-2.1.48-3.el8.noarch 11/11
90. Installed:
91. apr-1.6.3-9.el8.x86\_64
92. apr-util-1.6.1-6.el8.x86\_64
93. apr-util-bdb-1.6.1-6.el8.x86\_64
94. apr-util-openssl-1.6.1-6.el8.x86\_64
95. brotli-1.0.6-1.el8.x86\_64
96. centos-logos-httpd-80.5-2.el8.noarch
97. httpd-2.4.37-21.module\_el8.2.0+382+15b0afa8.x86\_64
98. httpd-filesystem-2.4.37-21.module\_el8.2.0+382+15b0afa8.noarch
99. httpd-tools-2.4.37-21.module\_el8.2.0+382+15b0afa8.x86\_64
100. mailcap-2.1.48-3.el8.noarch
101. mod\_http2-1.11.3-3.module\_el8.2.0+307+4d18d695.x86\_64
102. Complete!
103. Removing intermediate container 62163df9c424
104. ---> 2477edfffca1
105. Successfully built 2477edfffca1
106. Successfully tagged centoswithhttpd:1

The -t stands for tagging the newly created image with an appropriate name.

Step 4: Check the image list to see our newly created image.

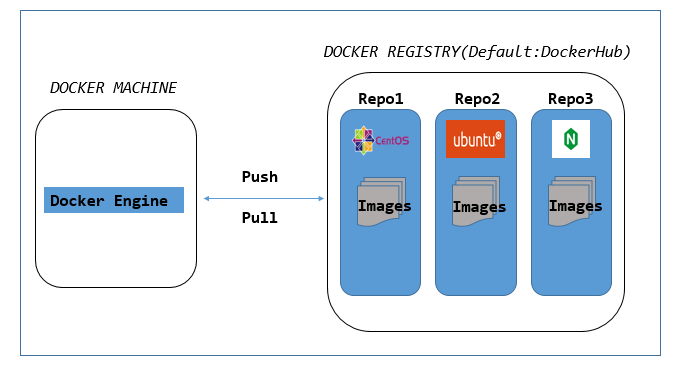
1. $ docker images
2. REPOSITORY TAG IMAGE ID CREATED SIZE
3. centoswithhttpd 1 2477edfffca1 4 minutes ago 254MB
4. centos 8 831691599b88 13 days ago 215MB
5. ubuntu latest 16508e5c265d 22 months ago 84.1MB
6. redis latest 4e8db158f18d 23 months ago 83.4MB
7. weaveworks/scope 1.9.1 4b07159e407b 23 months ago 68MB
8. alpine latest 11cd0b38bc3c 24 months ago 4.41MB

Step 5: Spin a new container with the new docker image.

1. $ docker run -it centoswithhttpd:1
2. [root@a69aec27a7b3 /]#
3. [root@a69aec27a7b3 /]# rpm -q httpd
4. httpd-2.4.37-21.module\_el8.2.0+382+15b0afa8.x86\_64

# Pushing your image to Docker registry

We have successfully created the Docker image that has to be shared with the trainees. During the course, we have pulled a docker image multiple times from Docker registry. By default, the Docker engine tries to pull the image from DockerHub. (For this, the Docker host must be connected to the internet). Docker also allows us to run a private Docker registry accessible only to our organisation.

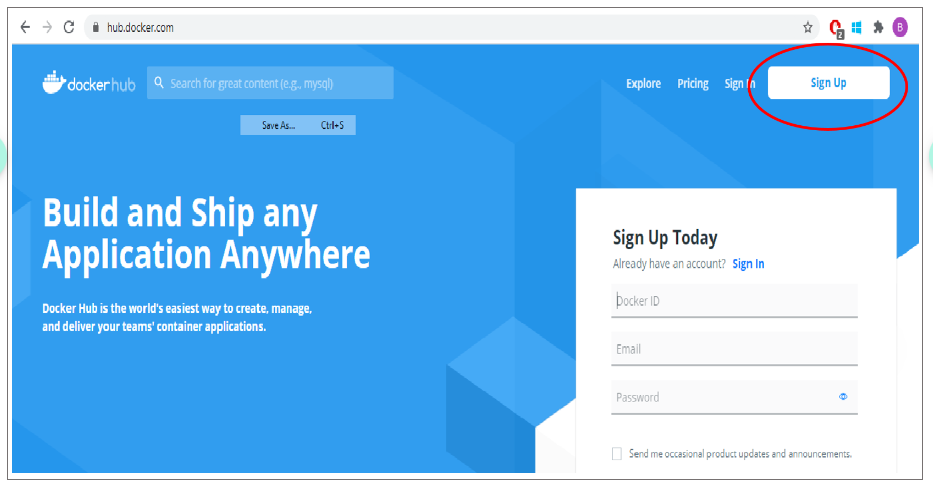


Apart from just pulling the image, what if we want to publish our own Docker image so that it is available for other users in the world?

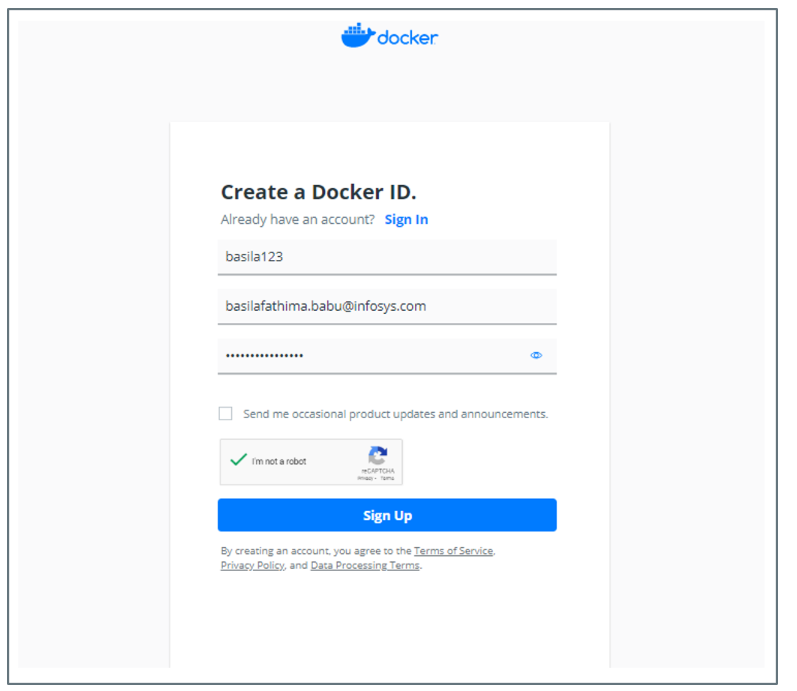
Using Docker registry, we can store and distribute these images effectively. By doing so, you can share your Docker image with the world or choose to share to a chosen audience. Before you can publish you own image in Docker Hub, you should have a Docker ID of your own. In this demo, we will create a new Docker ID and push our Docker image.

Note: To perform this in your system, make sure your Docker Host is connected to the internet.

Step 1: Access hub.docker.com and click Sign Up.

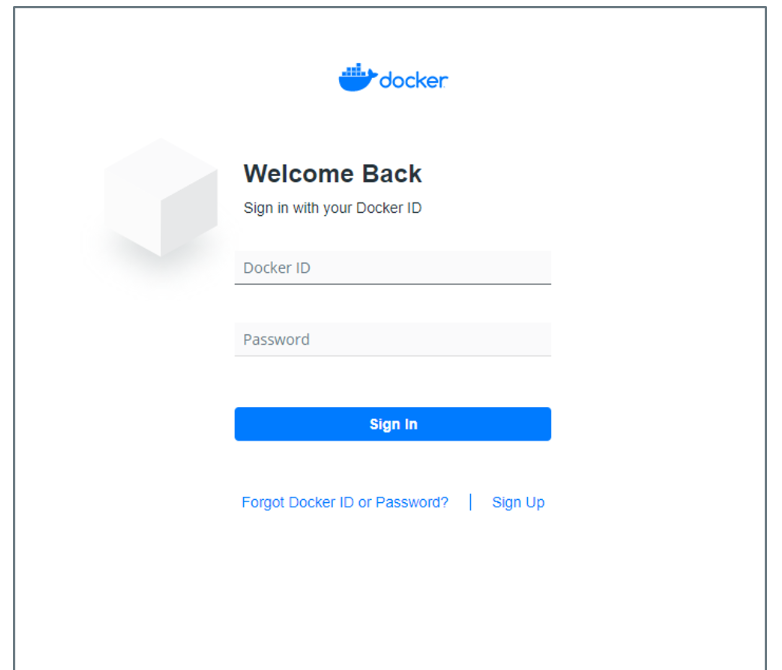


Step 2: Provide your username, email ID and a password for your DockerHub account. Click on Sign Up.



Step 3: Once you have signed up, Docker send a verification mail to your registered mail account. Verify the account. Without verifiying, we will not be able to create repository or work with Docker Hub.

Step 4: Once verified, Log in to Docker Hub with your credentials.



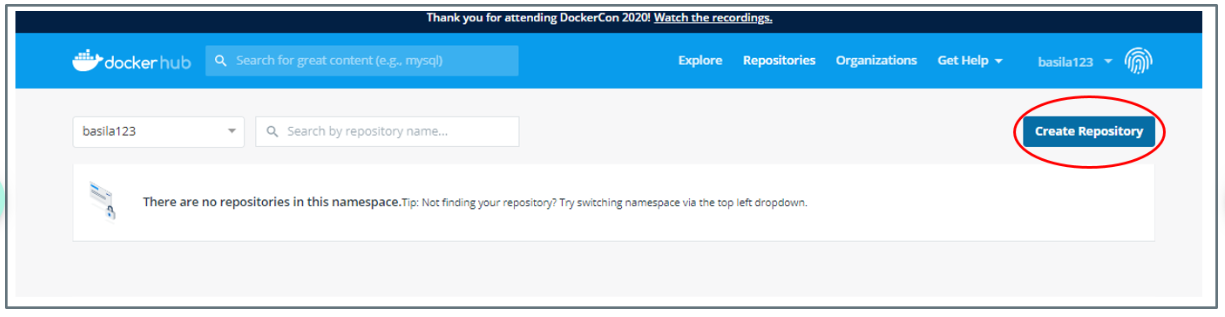
# Creating Public/Private repositories in DockerHub

After the successful creation of a DockerID for the user, the next step is to play around the DockerHub by creating repositories to store and manage your Docker images. DockerHub allows us to create types of repositories: Public and Private. A Public repository can be accessed by all the Docker users in the world whereas a private repository is only available to the owner. Usually, this can be mentioned while creating the repository. However, DockerHub allows us to change the visibility of existing repositories as well.

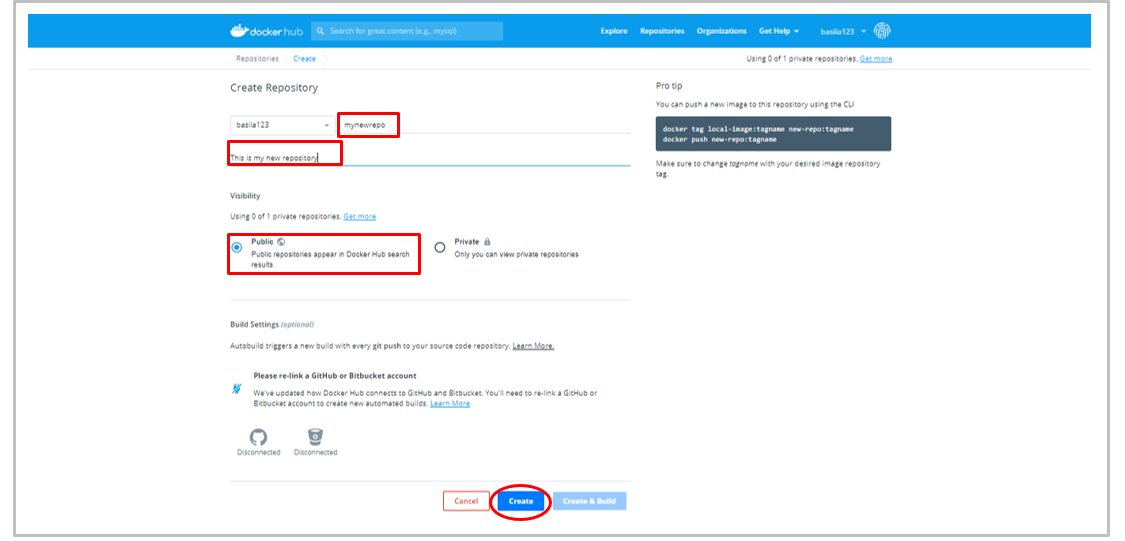
A repository should be created under a namespace. By default, there is one namespace present with every DockerID, which is the DockerID itself. Repositories can be created under this namespace using the following steps.

Step 1: Sign in to Docker Hub and Click on Repositories.

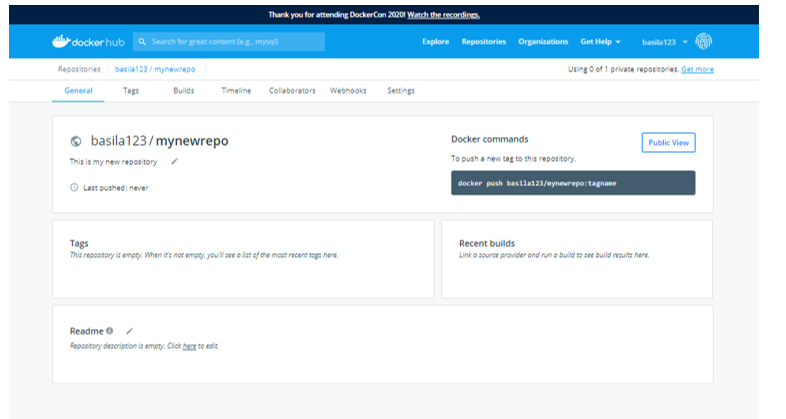
Step 2: Click on Create Repositories.



Step 3: Enter a unique name for the repository and enter the description. Also, choose the visibility to be Public. Click on create.



Once the repository is created, the docker images can be pushed into it.



# Pushing Docker image to Docker hub

Let us push the "centoswithhttpd:1" image, we created in the previous demo to our repository. Follow the below steps to push any number of local Docker image to your DockerHub repository.

Note: To push a local image to DockerHub repository, your Docker machine needs to connected to the internet.

Step 1: Before pushing a Docker image to a Dockerhub repository, the image must be named appropriately. The following convention must be followed while naming the image.

<dockerid>/<repo\_name>:<tag>

Using this convention, multiple images can be pushed inside a repository by naming the images with different tags. If the tag is not mentioned, the default tag "latest" is assumed. Images can be named as follows:

i) Naming the image while building the image:

1. $ docker build -t <dockerid>/<repo\_name>:<tag> Dockerfile

ii) Naming an existing image:

1. $ docker tag <existing\_image> <dockerid>/<repo\_name>:<tag>

iii) Naming the image while committing changes:

1. $ docker commit <existing\_container> <dockerid>/<repo\_name>:<tag>

In this demo, we are renaming an existing image, centoswithhttps:1.

1. $ docker tag centoswithhttpd:1 basila123/mynewrepo:centoswithhttpd
2. $ docker images
3. REPOSITORY TAG IMAGE ID CREATED SIZE
4. basila123/mynewrepo centoswithhttpd c3c247ab0923 52 seconds ago 254MB
5. centoswithhttpd 1 c3c247ab0923 52 seconds ago 254MB

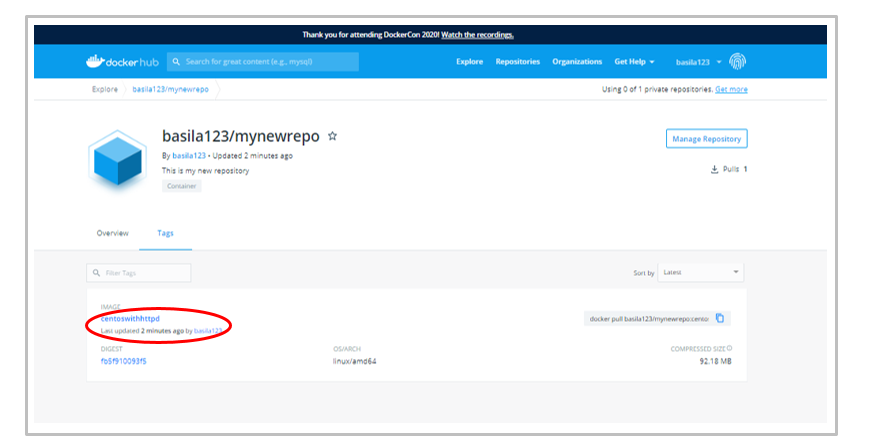
Step 2: Now the renamed image can be pushed to the repository. To do this, we have to login to the DockerHub account from the command line using "docker login" command.

1. $ docker login
2. Login with your Docker ID to push and pull images from Docker Hub. If you don't have a Docker ID, head over to https://hub.docker.com to create one.
3. Username: basila123
4. Password:
5. Login Succeeded

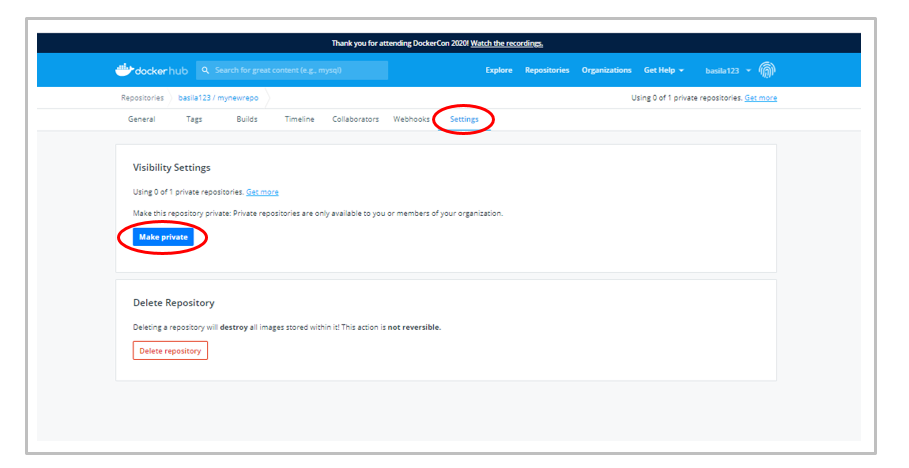
Step 3: Push the docker image to the repository now using "docker push" command.

1. $ docker push basila123/mynewrepo:centoswithhttpd
2. The push refers to repository [docker.io/basila123/mynewrepo]
3. 80b77f3d336f: Pushed
4. eb29745b8228: Mounted from library/centos
5. centoswithhttpd: digest: sha256:fb5f910093f5dcfe529667e6d9551a7bc5d459184f92fa11bb98006fc17221fc
6. size: 741

Now, this image is available to the Docker community for use. Click on the "tags" column of you repository to view the image being present in the DockerHub.



Note: To create a Private DockerHub repository, select "Private" as the visibility. The visibility of existing repositories can be changed in the Settings tab of the repository.

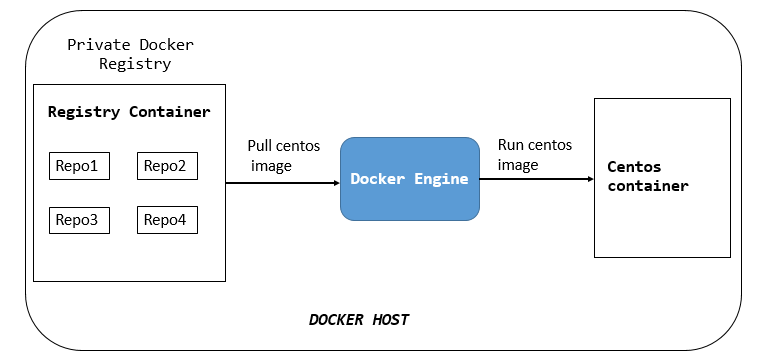


# Private Docker registry

Hosting your Docker images on Docker Hub, as simple as it sounds, is not always chosen by organisations dealing with critical information. The data dealt within the organisation should be protected off breach at all times. That being said, hosting a Docker image created for an organisation level purpose on Docker Hub sounds ridiculous. Here is where Private registries come into picture.

Docker allows us to have a Private Docker registry of our own hosted inside our organisation. This Private Docker registry can then be used to store and manage images that are ought to be kept within the walls of organisation.

Docker provides us a "registry" container which can be used to host a private docker registry. This registry will remain as the source of our Docker images within the organisation.



# Create a Private Docker registry

Follow the below steps to create local private docker registry.

Step 1: Run a registry container in your Docker host. We are going to expose a port of the container to the port of our host machine.

1. $ docker run -d -p 5000:5000 --name registry registry:2
2. Unable to find image 'registry:2' locally
3. 2: Pulling from library/registry
4. cbdbe7a5bc2a: Pull complete
5. 47112e65547d: Pull complete
6. 46bcb632e506: Pull complete
7. c1cc712bcecd: Pull complete
8. 3db6272dcbfa: Pull complete
9. Digest: sha256:8be26f81ffea54106bae012c6f349df70f4d5e7e2ec01b143c46e2c03b9e551d
10. Status: Downloaded newer image for registry:2
11. 23b142cfd4daedd01ad51e01227e4fcce157a460259a5b9e82e1b417dcb6c4ed

-p - to specify the ports.

registry - name of the container.

registry:2 - tagging the image with this name.

Step 2: Check the list of docker images.

1. $ docker images
2. REPOSITORY TAG IMAGE ID CREATED SIZE
3. registry 2 2d4f4b5309b1 12 days ago 26.2MB
4. ubuntu latest 16508e5c265d 22 months ago 84.1MB
5. redis latest 4e8db158f18d 23 months ago 83.4MB
6. weaveworks/scope 1.9.1 4b07159e407b 23 months ago 68MB
7. alpine latest 11cd0b38bc3c 24 months ago 4.41MB

We have ubuntu:latest image in our Docker host. We will try to upload this to our private docker registry. Before that, the image has to be tagged appropriately.

Step 3: Tag the ubuntu image as follows.

1. $ docker tag ubuntu localhost:5000/ubuntu
2. $ docker images
3. REPOSITORY TAG IMAGE ID CREATED SIZE
4. registry 2 2d4f4b5309b1 12 days ago 26.2MB
5. ubuntu latest 16508e5c265d 22 months ago 84.1MB
6. localhost:5000/ubuntu latest 16508e5c265d 22 months ago 84.1MB
7. redis latest 4e8db158f18d 23 months ago 83.4MB
8. weaveworks/scope 1.9.1 4b07159e407b 23 months ago 68MB
9. alpine latest 11cd0b38bc3c 24 months ago 4.41MB

localhost:5000 - name of the private docker registry

ubuntu - name of the repository where the image is going to be pushed.

Specify a tag if needed.

Step 4: Push the newly tagged image to our private docker registry.

1. $ docker push localhost:5000/ubuntu
2. The push refers to repository [localhost:5000/ubuntu]
3. ec8257ff6a7a: Pushed
4. 7422efa72a14: Pushed
5. b6a02001ba33: Pushed
6. a26724645421: Pushed
7. a30b835850bf: Pushed
8. latest: digest: sha256:ac533e4ead4110211a4d67cbf44ed8b7d1aca2b8e6f15d1e8768eadaf433dd31 size: 1357

Step 5: Delete both the ubuntu images present in our Docker host now.

1. $ docker rmi ubuntu:latest
2. Untagged: ubuntu:latest
3. Untagged: ubuntu@sha256:72f832c6184b55569be1cd9043e4a80055d55873417ea792d989441f207dd2c7
4. $ docker rmi localhost:5000/ubuntu
5. Untagged: localhost:5000/ubuntu:latest
6. Untagged: localhost:5000/ubuntu@sha256:ac533e4ead4110211a4d67cbf44ed8b7d1aca2b8e6f15d1e8768eadaf433dd31
7. Deleted: sha256:16508e5c265dcb5c05017a2a8a8228ae12b7b56b2cda0197ed5411bda200a961
8. Deleted: sha256:9c86133a1c6d4396a658294f4327a5cf7bcd7572c272bf5f75200f29326e8b6c
9. Deleted: sha256:33eb1e02dcc9a707a475bf5aa739bd9f6d607b355beae5544b6576d1534fe949
10. Deleted: sha256:e54feb0ab7d782d04296e5d56bbf7efc843233497f0a16aab21bf8995f3385a7
11. Deleted: sha256:c2f76372c07b398c959f25ac97af0c4080e323d082e809de4316c1c728d5c096
12. Deleted: sha256:a30b835850bfd4c7e9495edf7085cedfad918219227c7157ff71e8afe2661f63

List the docker images to make sure there are  no ubuntu images currently.

1. $ docker images
2. REPOSITORY TAG IMAGE ID CREATED SIZE
3. registry 2 2d4f4b5309b1 12 days ago 26.2MB
4. redis latest 4e8db158f18d 23 months ago 83.4MB
5. weaveworks/scope 1.9.1 4b07159e407b 23 months ago 68MB
6. alpine latest 11cd0b38bc3c 24 months ago 4.41MB

Step 6: Pull the ubuntu image from our private docker registry.

1. $ docker pull localhost:5000/ubuntu
2. Using default tag: latest
3. latest: Pulling from ubuntu
4. 124c757242f8: Pull complete
5. 2ebc019eb4e2: Pull complete
6. dac0825f7ffb: Pull complete
7. 82b0bb65d1bf: Pull complete
8. ef3b655c7f88: Pull complete
9. Digest: sha256:ac533e4ead4110211a4d67cbf44ed8b7d1aca2b8e6f15d1e8768eadaf433dd31
10. Status: Downloaded newer image for localhost:5000/ubuntu:latest

Check the image list to find the ubuntu image.

1. $ docker images
2. REPOSITORY TAG IMAGE ID CREATED SIZE
3. registry 2 2d4f4b5309b1 12 days ago 26.2MB
4. localhost:5000/ubuntu latest 16508e5c265d 22 months ago 84.1MB
5. redis latest 4e8db158f18d 23 months ago 83.4MB
6. weaveworks/scope 1.9.1 4b07159e407b 23 months ago 68MB
7. alpine latest 11cd0b38bc3c 24 months ago 4.41MB

Likewise, we will be able to push and pull Docker images within our organisation using the private Docker registry as long as the registry container is running in the environment.

# Build an Apache web server application

If you have followed every resource of this course till here, You would now able to launch any containerized application on your own, test it and expose it to the outside world. In a way of consolidation what you have learnt till now, Let us go ahead and create an Apache web server application in Docker.

In this demo, The Apache web server will be hosted on a CentOS machine. We will start this demo right from creating a Docker Image for the application,

Step 1: We will build the image for our application with CentOS as the parent image. Make sure the CentOS image is present in the Docker Host.

1. $ docker pull centos
2. Using default tag: latest
3. latest: Pulling from library/centos
4. 6910e5a164f7: Pull complete
5. Digest: sha256:4062bbdd1bb0801b0aa38e0f83dece70fb7a5e9bce223423a68de2d8b784b43b
6. Status: Downloaded newer image for centos:latest

Step 2: We have the parent image now. After this, the process can be carried out in two ways. We can either build the web server image using a Dockerfile or in the interactive way. Since interactive way involves a lot of manual commands, we will choose the Docker file method over it. Let us create a Dockerfile first. Create a new directory "webserver" and create a new text file, "Dockerfile".

1. $ mkdir webserver
2. $ cd webserver
3. $ touch Dockerfile
4. $ ls
5. Dockerfile

Step 3: In the webserver directory, create an index.html file for the webserver. This file will be copied from the host to the Docker image when we build it.

1. $ vim index.html
2. $ cat index.html
3. <html>
4. <body>
5. <h1>Welcome to Webserver!<h1>
6. <p>You have successfully hosted a web application</p>
7. </body>
8. </html>

Step 4: Edit the Docker file as per our requirement using appropriate Dockerfile commands.

1. FROM centos:latest
2. MAINTAINER IMSAcademy
3. RUN yum -y install httpd
4. COPY index.html /var/www/html/
5. CMD [“/usr/sbin/httpd”, “-D”, “FOREGROUND”]
6. EXPOSE 80

FROM - specifies the parent image, centos.

MAINTAINER - owner of this Dockerfile.

RUN - commands to be run.

COPY - file that has to be copied from host to the image.

CMD - the commands to be run as soon as the container is launched.

* /usr/bin/httpd - starts the webservice.
* -D FOREGROUND - argument which is used to run the webserver in the background.

EXPOSE - specifies the port number where the webserver is running. The default webserver port number 80 is used.

Step 5: Build the docker image. Also, add a tag to the image as below.

1. $ docker build -t webserver:1 ./webserver
2. Sending build context to Docker daemon 3.072kB
3. Step 1/6 : FROM centos:latest
4. ---> 831691599b88
5. Step 2/6 : MAINTAINER IMSAcademy
6. ---> Running in 324b65e19c92
7. Removing intermediate container 324b65e19c92
8. ---> 38e3b61373be
9. Step 3/6 : RUN yum -y install httpd
10. ---> Running in 07f1c494f32c
11. CentOS-8 - AppStream 13 MB/s | 5.8 MB 00:00
12. CentOS-8 - Base 3.8 MB/s | 2.2 MB 00:00
13. CentOS-8 - Extras 17 kB/s | 6.7 kB 00:00
14. Dependencies resolved.
15. ================================================================================
16. Package Arch Version Repo Size
17. ================================================================================
18. Installing:
19. httpd x86\_64 2.4.37-21.module\_el8.2.0+382+15b0afa8 AppStream 1.7 M
20. Installing dependencies:
21. apr x86\_64 1.6.3-9.el8 AppStream 125 k
22. apr-util x86\_64 1.6.1-6.el8 AppStream 105 k
23. brotli x86\_64 1.0.6-1.el8 BaseOS 323 k
24. centos-logos-httpd
25. noarch 80.5-2.el8 BaseOS 24 k
26. httpd-filesystem noarch 2.4.37-21.module\_el8.2.0+382+15b0afa8 AppStream 36 k
27. httpd-tools x86\_64 2.4.37-21.module\_el8.2.0+382+15b0afa8 AppStream 103 k
28. mailcap noarch 2.1.48-3.el8 BaseOS 39 k
29. mod\_http2 x86\_64 1.11.3-3.module\_el8.2.0+307+4d18d695 AppStream 157 k
30. Installing weak dependencies:
31. apr-util-bdb x86\_64 1.6.1-6.el8 AppStream 25 k
32. apr-util-openssl x86\_64 1.6.1-6.el8 AppStream 27 k
33. Enabling module streams:
34. httpd 2.4
35. Transaction Summary
36. ================================================================================
37. Install 11 Packages
38. Total download size: 2.6 M
39. Installed size: 7.5 M
40. Downloading Packages:
41. (1/11): apr-util-bdb-1.6.1-6.el8.x86\_64.rpm 537 kB/s | 25 kB 00:00
42. (2/11): apr-util-openssl-1.6.1-6.el8.x86\_64.rpm 1.2 MB/s | 27 kB 00:00
43. (3/11): apr-1.6.3-9.el8.x86\_64.rpm 1.6 MB/s | 125 kB 00:00
44. (4/11): apr-util-1.6.1-6.el8.x86\_64.rpm 1.2 MB/s | 105 kB 00:00
45. (5/11): httpd-filesystem-2.4.37-21.module\_el8.2 2.6 MB/s | 36 kB 00:00
46. (6/11): httpd-tools-2.4.37-21.module\_el8.2.0+38 3.4 MB/s | 103 kB 00:00
47. (7/11): mod\_http2-1.11.3-3.module\_el8.2.0+307+4 3.0 MB/s | 157 kB 00:00
48. (8/11): centos-logos-httpd-80.5-2.el8.noarch.rp 243 kB/s | 24 kB 00:00
49. (9/11): httpd-2.4.37-21.module\_el8.2.0+382+15b0 7.6 MB/s | 1.7 MB 00:00
50. (10/11): mailcap-2.1.48-3.el8.noarch.rpm 450 kB/s | 39 kB 00:00
51. (11/11): brotli-1.0.6-1.el8.x86\_64.rpm 978 kB/s | 323 kB 00:00
52. --------------------------------------------------------------------------------
53. Total 3.6 MB/s | 2.6 MB 00:00
54. warning: /var/cache/dnf/AppStream-02e86d1c976ab532/packages/apr-1.6.3-9.el8.x86\_64.rpm: Header V3 RSA/SHA256 Signature, key ID 8483c65d: NOKEY
55. CentOS-8 - AppStream 1.6 MB/s | 1.6 kB 00:00
56. Importing GPG key 0x8483C65D:
57. Userid : "CentOS (CentOS Official Signing Key) <security@centos.org>"
58. Fingerprint: 99DB 70FA E1D7 CE22 7FB6 4882 05B5 55B3 8483 C65D
59. From : /etc/pki/rpm-gpg/RPM-GPG-KEY-centosofficial
60. Key imported successfully
61. Running transaction check
62. Transaction check succeeded.
63. Running transaction test
64. Transaction test succeeded.
65. Running transaction
66. Preparing : 1/1
67. Installing : apr-1.6.3-9.el8.x86\_64 1/11
68. Running scriptlet: apr-1.6.3-9.el8.x86\_64 1/11
69. Installing : apr-util-bdb-1.6.1-6.el8.x86\_64 2/11
70. Installing : apr-util-openssl-1.6.1-6.el8.x86\_64 3/11
71. Installing : apr-util-1.6.1-6.el8.x86\_64 4/11
72. Running scriptlet: apr-util-1.6.1-6.el8.x86\_64 4/11
73. Installing : httpd-tools-2.4.37-21.module\_el8.2.0+382+15b0afa8. 5/11
74. Installing : mailcap-2.1.48-3.el8.noarch 6/11
75. Installing : centos-logos-httpd-80.5-2.el8.noarch 7/11
76. Installing : brotli-1.0.6-1.el8.x86\_64 8/11
77. Running scriptlet: httpd-filesystem-2.4.37-21.module\_el8.2.0+382+15b0 9/11
78. Installing : httpd-filesystem-2.4.37-21.module\_el8.2.0+382+15b0 9/11
79. Installing : mod\_http2-1.11.3-3.module\_el8.2.0+307+4d18d695.x86 10/11
80. Installing : httpd-2.4.37-21.module\_el8.2.0+382+15b0afa8.x86\_64 11/11
81. Running scriptlet: httpd-2.4.37-21.module\_el8.2.0+382+15b0afa8.x86\_64 11/11
82. Verifying : apr-1.6.3-9.el8.x86\_64 1/11
83. Verifying : apr-util-1.6.1-6.el8.x86\_64 2/11
84. Verifying : apr-util-bdb-1.6.1-6.el8.x86\_64 3/11
85. Verifying : apr-util-openssl-1.6.1-6.el8.x86\_64 4/11
86. Verifying : httpd-2.4.37-21.module\_el8.2.0+382+15b0afa8.x86\_64 5/11
87. Verifying : httpd-filesystem-2.4.37-21.module\_el8.2.0+382+15b0 6/11
88. Verifying : httpd-tools-2.4.37-21.module\_el8.2.0+382+15b0afa8. 7/11
89. Verifying : mod\_http2-1.11.3-3.module\_el8.2.0+307+4d18d695.x86 8/11
90. Verifying : brotli-1.0.6-1.el8.x86\_64 9/11
91. Verifying : centos-logos-httpd-80.5-2.el8.noarch 10/11
92. Verifying : mailcap-2.1.48-3.el8.noarch 11/11
93. Installed:
94. apr-1.6.3-9.el8.x86\_64
95. apr-util-1.6.1-6.el8.x86\_64
96. apr-util-bdb-1.6.1-6.el8.x86\_64
97. apr-util-openssl-1.6.1-6.el8.x86\_64
98. brotli-1.0.6-1.el8.x86\_64
99. centos-logos-httpd-80.5-2.el8.noarch
100. httpd-2.4.37-21.module\_el8.2.0+382+15b0afa8.x86\_64
101. httpd-filesystem-2.4.37-21.module\_el8.2.0+382+15b0afa8.noarch
102. httpd-tools-2.4.37-21.module\_el8.2.0+382+15b0afa8.x86\_64
103. mailcap-2.1.48-3.el8.noarch
104. mod\_http2-1.11.3-3.module\_el8.2.0+307+4d18d695.x86\_64
105. Complete!
106. Removing intermediate container 07f1c494f32c
107. ---> 6bff8362fee0
108. Step 4/6 : COPY index.html /var/www/html/
109. ---> 9e947d72dc7f
110. Step 5/6 : CMD ["/usr/sbin/httpd", "-D", "FOREGROUND"]
111. ---> Running in 066f114ff05a
112. Removing intermediate container 066f114ff05a
113. ---> dc994f9dc0ac
114. Step 6/6 : EXPOSE 80
115. ---> Running in a0220eb967c1
116. Removing intermediate container a0220eb967c1
117. ---> aabd34dad820
118. Successfully built aabd34dad820
119. Successfully tagged webserver:1

-t sepcifies the tag of the new image.

./webserver is the directory where the Dockerfile resides.

Step 6: List the Docker images to see the newly created image.

1. $ docker images
2. REPOSITORY TAG IMAGE ID CREATED SIZE
3. webserver 1 aabd34dad820 11 seconds ago 254MB
4. centos latest 831691599b88 2 weeks ago 215MB
5. ubuntu latest 16508e5c265d 22 months ago 84.1MB
6. redis latest 4e8db158f18d 23 months ago 83.4MB
7. weaveworks/scope 1.9.1 4b07159e407b 23 months ago 68MB
8. alpine latest 11cd0b38bc3c 24 months ago 4.41MB

Step 7: We are now good to run a container that is going to host a web application, as we wanted. Expose the port 80 of the container to port 80 of the host so that the web service is available to the outside world too.

1. $ docker run -dit -p 80:80 webserver:1
2. 5a78ab508974139be4c652f16ee17738dff07d8dc9ca45f37889904b51a5f556

-d runs the container in the background

80:80 exposes the container's port no.80 to the Docker host's port no.80.

Step 8: Let us test the web application we just deployed by accessing it in a web browser. You can also use the command below.

1. elinks http:*//192.168.10.10:80*

192.168.10.10 is the IP address of the Docker host.

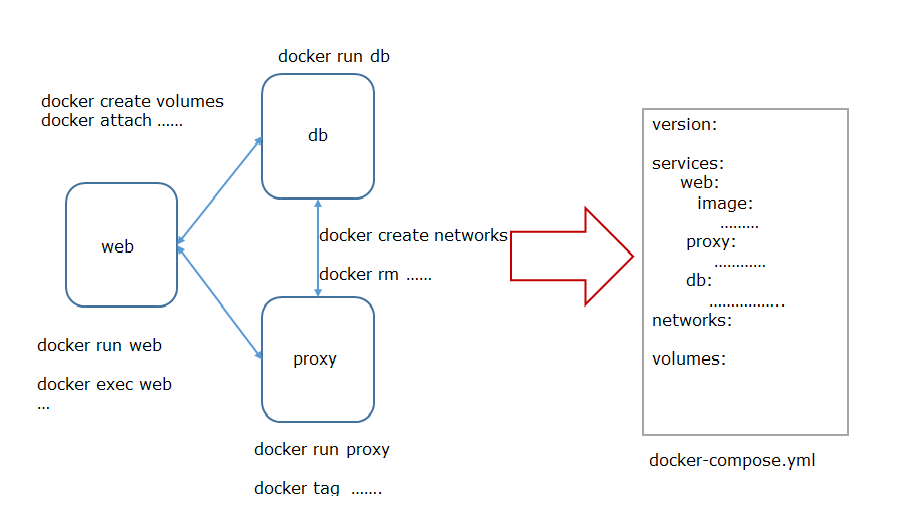
You can see the web page you created on your web browser now. We have successfully deployed a web application to the world using Docker. Docker, as flexible as we know it is, allows us to even deploy multi-tier applications without any uncertainty in maintaining the containers.

# Docker compose

# Introduction to Docker compose

Most project architectures require applications that include more than one container to be built and run at the same time. For example, building a web application requires two or more separate containers such as a web server and a database and a proxy to be up and running, in such scenarios building, running and connecting the containers using individual docker CLI commands on each container is time-consuming, cumbersome.

Docker compose solves this problem by allowing the user to define and run these multi-container applications using a single docker compose command.



Thus docker compose can be defined as “a tool for defining and running multi-container docker applications.”

 Compose uses YAML file format for defining containers and their configurations.   The “docker-compose.yml” is the default YAML file under which the containers configurations should be written. The user can define as many number of container instances the application requires inside the docker-compose file.

Executing a single docker-compose command will pull, build and run all containers defined inside the docker-compose.yml file.

# Docker Compose Features

The following features of Docker Compose make it a powerful and effective tool.

# 1. Multiple isolated environments on a single host

Compose uses the project name (the directory name under which the docker-compose file is created) to isolate application environments from each other inside a single host machine. This helps to build as many isolated applications required within a single host.

# 2. Only recreate containers that have changed

Compose caches the configuration used to create a container, so when the service or application is restarted, compose re-uses the existing containers and recreate only the containers whose configurations have been changed.

# 3. Preserve volume data when containers are created

Compose preserves and copies the data volumes from old containers to new containers, when the existing containers are recreated.

# 4.  Variables and moving a composition between environments

Compose supports variables to be declared in compose file, thereby enabling user to create more customized application with user defined environmental variables.

# Common use cases

Docker Compose can be used in all environments i.e., production, development, staging, testing, etc., However, the most common use cases of Docker Compose are listed below:

# 1. Development environments

Developing software requires applications to run in an isolated environments and interact with its service dependencies such as caches, queues, database, API, etc. The docker compose tool can be used to create such environments where the applications and dependencies can interact.

# 2. Automated testing environments

Automated end-to-end testing requires an isolated environment to run tests. Compose provides a convenient way to create and destroy isolated testing environments for the test suite

# 3. Single host deployments

Compose can be used to deploy a remote Docker Engine. The Docker Engine may be a single instance provisioned with Docker Machine or an entire Docker Swarm cluster.

# Working with Docker Compose

# Working with Docker Compose

Docker Compose can be run on Mac, Windows, Windows Server 2016 and Linux systems. Follow the below steps to install Docker Compose on your system.

# Prerequisites

1. Before installing Docker Compose, make sure Docker Engine is installed in your machine since Docker compose depends on Docker Engine.
2. In case you are using Docker Desktop for Mac or  Windows, Docker Compose comes preinstalled with the Desktop installation.
3. Docker compose can run by root user. To run Docker Compose using a non-root user, make sure that the user has sudo privilege.

# Installation

The below instructions can be followed if you want to install Docker Compose on a Linux system.

Step 1: In order to fulfill Prerequisite no.3, log in to the machine where the Docker daemon is running as a root user ( or a user with root privilege).

1. $ whoami
2. root

Step 2: Download the  the binaries present the GitHub compose repository page, with the help of curl utility. We are downloading the latest stable version of Docker compose (v1.26.1).

1. $ curl -L "https://github.com/docker/compose/releases/download/1.26.1/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
2. % Total % Received % Xferd Average Speed Time Time Time Current
3. Dload Upload Total Spent Left Speed
4. 100 638 100 638 0 0 1722 0 --:--:-- --:--:-- --:--:-- 1724
5. 100 11.6M 100 11.6M 0 0 5671k 0 0:00:02 0:00:02 --:--:-- 9.8M

Step 3: Apply executable permissions for the docker-compose Binary files.

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

Step 4: Verify the installation by checking the version of docker-compose.

1. $ docker-compose version
2. docker-compose version 1.26.1, build f216ddbf
3. docker-py version: 4.2.2
4. CPython version: 3.7.7
5. OpenSSL version: OpenSSL 1.1.0l 10 Sep 2019

To uninstall Docker Compose if you have installed using the above steps:

1. $ rm /usr/local/bin/docker-compose

# Working with Docker Compose

Docker Compose can be run on Mac, Windows, Windows Server 2016 and Linux systems. Follow the below steps to install Docker Compose on your system.

# Prerequisites

1. Before installing Docker Compose, make sure Docker Engine is installed in your machine since Docker compose depends on Docker Engine.
2. In case you are using Docker Desktop for Mac or  Windows, Docker Compose comes preinstalled with the Desktop installation.
3. Docker compose can run by root user. To run Docker Compose using a non-root user, make sure that the user has sudo privilege.

# Installation

The below instructions can be followed if you want to install Docker Compose on a Linux system.

Step 1: In order to fulfill Prerequisite no.3, log in to the machine where the Docker daemon is running as a root user ( or a user with root privilege).

1. $ whoami
2. root

Step 2: Download the  the binaries present the GitHub compose repository page, with the help of curl utility. We are downloading the latest stable version of Docker compose (v1.26.1).

1. $ curl -L "https://github.com/docker/compose/releases/download/1.26.1/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
2. % Total % Received % Xferd Average Speed Time Time Time Current
3. Dload Upload Total Spent Left Speed
4. 100 638 100 638 0 0 1722 0 --:--:-- --:--:-- --:--:-- 1724
5. 100 11.6M 100 11.6M 0 0 5671k 0 0:00:02 0:00:02 --:--:-- 9.8M

Step 3: Apply executable permissions for the docker-compose Binary files.

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

Step 4: Verify the installation by checking the version of docker-compose.

1. $ docker-compose version
2. docker-compose version 1.26.1, build f216ddbf
3. docker-py version: 4.2.2
4. CPython version: 3.7.7
5. OpenSSL version: OpenSSL 1.1.0l 10 Sep 2019

To uninstall Docker Compose if you have installed using the above steps:

1. $ rm /usr/local/bin/docker-compose

# Running Docker Compose as non-root user

As mentioned earlier, Docker Compose usually run by root user. If you want to run it as any non-root user, follow the below steps:

Step 1: Create an user "compose" and set the password.

1. $ useradd compose
2. $ passwd compose
3. Enter new UNIX password:
4. Retype new UNIX password:
5. passwd: password updated successfully

Step 2: Create a group name "docker" and add the user to the "docker" group.

1. $ groupadd docker
2. $ usermod -aG docker compose

Step 3: Run the following command to activate the changes to the Docker group.

1. $ newgrp docker

Step 4: Login as "compose" user  and verify the docker-compose version.

1. $ su - compose
2. $ docker-compose version
3. docker-compose version 1.26.1, build f216ddbf
4. docker-py version: 4.2.2
5. CPython version: 3.7.7
6. OpenSSL version: OpenSSL 1.1.0l 10 Sep 2019

Following the above steps enable us to run docker-compose as a non-root user without escalating the privilege.

# Compose file structure

To start creating containerized application using Compose, the user should create a .yml extension  file called docker-compose.yml in the present working directory. When docker-compose commands are executed in the terminal, by default Compose searches for the docker-compose.yml file in the current working directory to build and start the containerized applications. The User can create as many number of default docker-compose.yml file, but each one should be under a different directory/project name.

Thus, the default path for compose file is   ./docker-compose.yml.Compose file structure and syntax

The contents inside docker-compose file is organized under 2 levels:

1. Top level keys
2. Service level keys

# Top level key

These are the basic building blocks of a compose file. There are 4 primarily used top level keys.

Version: - specifies the compose file syntax version. There are several versions of compose file formats, starting with 1.x,2.x, 3.x. The latest release is 3.7

Services: - defines the configurations and dependencies of the containers that as be started as a part of application stack. Each service should have a unique name inside the compose file. The minimum information needed to be specified inside each service name to start the container is the image name.

Volumes: - used to specify named volumes that would mount a linked path present in the host machine with the file or directory inside container to store container data outside the container so that the data is persistent even if container crashes, also used to share volumes between two or more services.

Networks: - It is used to specify and configure networks needed for the services. It allows user to change the settings of the default network or connect to an external network or to define application-specific networks.

# Service level keys

These keys are used under the top-level keys for specifying the dependencies, configurations and environmental variables, etc. required to start the services. The mainly used service level keys are:

Image: used to specify the image from which containers are created.

Port: Using service level key called ports, a container port number can be mapped with host machine port number in the following manner “<host port number>: <container port number>”

Build:  used instead of image key to Specify the location of Dockerfile that will be used to build this container

Restart: Docker containers exit by default if no process is running in them. This restart key tells the container to restart if the container exits.

Environment:  used to specify the environment variables for a service

Networks:  used to define an application specific networks and configurations.

Volumes: used to map the volume in the host machine with the container’s volume in the following format.  < host machine volume path: container’s volume>

Container\_name: used to give user defined container name.

Depends\_on: used to specify the service names on which the current service depends.

# Getting started with Docker Compose

In this demo, we will build a simple web application running on Docker Compose. The application uses a tomcat web server and nginx proxy server. Before proceeding, make sure you have Docker engine and Docker compose installed in your machine.

Step 1: Create the setup

Create a working directory for this application. And make it as the current working directory.

1. $ mkdir compose
2. $ cd compose

Step 2: Create a Dockerfile

Create a Dockerfile that will build a tomcat image with all the dependencies of the application. In the "tomcat" directory, create a file "Dockerfile" and add the following content using any editor of your choice.

1. $ cat Dockerfile
2. FROM tomcat

This tells Docker to:

Build the image using tomcat as the parent image.

Step 3: Create a Compose file

The two services have to be defined in a docker-compose file. Create a docker-compose.yml file in the current directory of our application and add the following content.

1. $ cat docker-compose.yml
2. version : '2'
3. services:
4. web\_server:
5. build: .
6. proxy\_server:
7. image: nginx

Version of the docker-compose.yml file is mentioned as 2. If we are working with clustered applications, version 3 is mandatory. But for simple docker-compose applications, version 2 is enough.

Web service:

web\_server is built from the custom Dockerfile present in the current working directory, since service level build key is specified. The new image is tagged as <directory\_name>\_<service\_name> and is used to run the container.

Nginx service:

proxy\_server is built using nginx image locally or from the Docker registry, since service level image key is specified.

Step 4: Validate the compose file. Execute the following command to do the same.

1. $ docker-compose config
2. services:
3. proxy\_server:
4. image: nginx
5. web\_server:
6. build:
7. context: /compose
8. version: '2.0'

Note: Execute the above command with -q, if you want to silently validate.

Step 5: Build and run the application with Docker Compose.

This command immediately spins up two containers as mentioned in the file.

1. $ docker-compose up
2. Creating network "compose\_default" with the default driver
3. Building web\_server
4. Step 1/1 : FROM tomcat
5. latest: Pulling from library/tomcat
6. e9afc4f90ab0: Pull complete
7. 989e6b19a265: Pull complete
8. af14b6c2f878: Pull complete
9. 5573c4b30949: Pull complete
10. fb1a405f128d: Pull complete
11. 612a9f566fdc: Pull complete
12. cf63ebed1142: Pull complete
13. fbb20561cd50: Pull complete
14. 76c915a2cfb7: Pull complete
15. a2c2864c3363: Pull complete
16. Digest: sha256:11f247df062558074169fb92a54033ab2eb6563bda9765b3a9e53106db3c2f4a
17. Status: Downloaded newer image for tomcat:latest
18. ---> 6055d4d564e1
19. Successfully built 6055d4d564e1
20. Successfully tagged compose\_web\_server:latest
21. WARNING: Image for service web\_server was built because it did not already exist. To rebuild this image you must use `docker-compose build` or `docker-compose up --build`.
22. Pulling proxy\_server (nginx:)...
23. latest: Pulling from library/nginx
24. 8559a31e96f4: Pull complete
25. 8d69e59170f7: Pull complete
26. 3f9f1ec1d262: Pull complete
27. d1f5ff4f210d: Pull complete
28. 1e22bfa8652e: Pull complete
29. Digest: sha256:21f32f6c08406306d822a0e6e8b7dc81f53f336570e852e25fbe1e3e3d0d0133
30. Status: Downloaded newer image for nginx:latest
31. Creating compose\_proxy\_server\_1 ... done
32. Creating compose\_web\_server\_1 ... done
33. Attaching to compose\_web\_server\_1, compose\_proxy\_server\_1
34. web\_server\_1 | NOTE: Picked up JDK\_JAVA\_OPTIONS: --add-opens=java.base/java.lang=ALL-UNNAMED --add-opens=java.base/java.io=ALL-UNNAMED --add-opens=java.rmi/sun.rmi.transport=ALL-UNNAMED
35. proxy\_server\_1 | /docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will attempt to perform configuration
36. proxy\_server\_1 | /docker-entrypoint.sh: Looking for shell scripts in /docker-entrypoint.d/
37. proxy\_server\_1 | /docker-entrypoint.sh: Launching /docker-entrypoint.d/10-listen-on-ipv6-by-default.sh
38. proxy\_server\_1 | 10-listen-on-ipv6-by-default.sh: Getting the checksum of /etc/nginx/conf.d/default.conf
39. proxy\_server\_1 | 10-listen-on-ipv6-by-default.sh: Enabled listen on IPv6 in /etc/nginx/conf.d/default.conf
40. proxy\_server\_1 | /docker-entrypoint.sh: Launching /docker-entrypoint.d/20-envsubst-on-templates.sh
41. proxy\_server\_1 | /docker-entrypoint.sh: Configuration complete; ready for start up
42. web\_server\_1 | 07-Jul-2020 11:05:10.373 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log Server version name: Apache Tomcat/9.0.37
43. web\_server\_1 | 07-Jul-2020 11:05:10.393 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log Server built: Jun 30 2020 20:09:49 UTC
44. web\_server\_1 | 07-Jul-2020 11:05:10.394 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log Server version number: 9.0.37.0
45. web\_server\_1 | 07-Jul-2020 11:05:10.395 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log OS Name: Linux
46. web\_server\_1 | 07-Jul-2020 11:05:10.396 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log OS Version: 4.14.67-1-lts
47. web\_server\_1 | 07-Jul-2020 11:05:10.397 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log Architecture: amd64
48. web\_server\_1 | 07-Jul-2020 11:05:10.397 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log Java Home: /usr/local/openjdk-11
49. web\_server\_1 | 07-Jul-2020 11:05:10.398 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log JVM Version: 11.0.7+10
50. web\_server\_1 | 07-Jul-2020 11:05:10.398 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log JVM Vendor: Oracle Corporation
51. web\_server\_1 | 07-Jul-2020 11:05:10.404 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log CATALINA\_BASE: /usr/local/tomcat
52. web\_server\_1 | 07-Jul-2020 11:05:10.406 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log CATALINA\_HOME: /usr/local/tomcat
53. web\_server\_1 | 07-Jul-2020 11:05:10.479 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log Command line argument: --add-opens=java.base/java.lang=ALL-UNNAMED
54. web\_server\_1 | 07-Jul-2020 11:05:10.481 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log Command line argument: --add-opens=java.base/java.io=ALL-UNNAMED
55. web\_server\_1 | 07-Jul-2020 11:05:10.482 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log Command line argument: --add-opens=java.rmi/sun.rmi.transport=ALL-UNNAMED
56. web\_server\_1 | 07-Jul-2020 11:05:10.483 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log Command line argument: -Djava.util.logging.config.file=/usr/local/tomcat/conf/logging.properties
57. web\_server\_1 | 07-Jul-2020 11:05:10.496 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log Command line argument: -Djava.util.logging.manager=org.apache.juli.ClassLoaderLogManager
58. web\_server\_1 | 07-Jul-2020 11:05:10.498 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log Command line argument: -Djdk.tls.ephemeralDHKeySize=2048
59. web\_server\_1 | 07-Jul-2020 11:05:10.500 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log Command line argument: -Djava.protocol.handler.pkgs=org.apache.catalina.webresources
60. web\_server\_1 | 07-Jul-2020 11:05:10.501 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log Command line argument: -Dorg.apache.catalina.security.SecurityListener.UMASK=0027
61. web\_server\_1 | 07-Jul-2020 11:05:10.501 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log Command line argument: -Dignore.endorsed.dirs=
62. web\_server\_1 | 07-Jul-2020 11:05:10.502 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log Command line argument: -Dcatalina.base=/usr/local/tomcat
63. web\_server\_1 | 07-Jul-2020 11:05:10.502 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log Command line argument: -Dcatalina.home=/usr/local/tomcat
64. web\_server\_1 | 07-Jul-2020 11:05:10.520 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log Command line argument: -Djava.io.tmpdir=/usr/local/tomcat/temp
65. web\_server\_1 | 07-Jul-2020 11:05:10.521 INFO [main] org.apache.catalina.core.AprLifecycleListener.lifecycleEvent Loaded Apache Tomcat Native library [1.2.24] using APR version [1.6.5].
66. web\_server\_1 | 07-Jul-2020 11:05:10.521 INFO [main] org.apache.catalina.core.AprLifecycleListener.lifecycleEvent APR capabilities: IPv6 [true], sendfile [true], accept filters [false], random [true].
67. web\_server\_1 | 07-Jul-2020 11:05:10.522 INFO [main] org.apache.catalina.core.AprLifecycleListener.lifecycleEvent APR/OpenSSL configuration: useAprConnector [false], useOpenSSL [true]
68. web\_server\_1 | 07-Jul-2020 11:05:10.532 INFO [main] org.apache.catalina.core.AprLifecycleListener.initializeSSL OpenSSL successfully initialized [OpenSSL 1.1.1d 10 Sep 2019]
69. web\_server\_1 | 07-Jul-2020 11:05:12.032 INFO [main] org.apache.coyote.AbstractProtocol.init Initializing ProtocolHandler ["http-nio-8080"]
70. web\_server\_1 | 07-Jul-2020 11:05:12.216 INFO [main] org.apache.catalina.startup.Catalina.load Server initialization in [2701] milliseconds
71. web\_server\_1 | 07-Jul-2020 11:05:12.438 INFO [main] org.apache.catalina.core.StandardService.startInternal Starting service [Catalina]
72. web\_server\_1 | 07-Jul-2020 11:05:12.441 INFO [main] org.apache.catalina.core.StandardEngine.startInternal Starting Servlet engine: [Apache Tomcat/9.0.37]
73. web\_server\_1 | 07-Jul-2020 11:05:12.477 INFO [main] org.apache.coyote.AbstractProtocol.start Starting ProtocolHandler ["http-nio-8080"]
74. web\_server\_1 | 07-Jul-2020 11:05:12.523 INFO [main] org.apache.catalina.startup.Catalina.start Server startup in [305] milliseconds
75. ^CGracefully stopping... (press Ctrl+C again to force)
76. Stopping compose\_web\_server\_1 ... done
77. Stopping compose\_proxy\_server\_1 ... done

As you can see, the terminal is occupied. To come out of it, press Ctrl+C. But doing this also stops our containers. Execute 'docker ps' to check

1. $ docker ps
2. CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

Always, compose up the containers in the background.

1. $ docker-compose up -d
2. Starting compose\_proxy\_server\_1 ... done
3. Starting compose\_web\_server\_1 ... done

Now, check the list of containers.

1. $ docker ps
2. CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
3. 40d2b2ae3c77 compose\_web\_server "catalina.sh run" 3 minutes ago Up About a minute 8080/tcp compose\_web\_server\_1
4. e6a7df2dba55 nginx "/docker-entrypoint.…" 3 minutes ago Up About a minute 80/tcp compose\_proxy\_server\_1

The names of the container is being named as <directory\_name>\_<service\_name>.

Step 6: To bring down the containers:

1. $ docker-compose down
2. Stopping compose\_web\_server\_1 ... done
3. Stopping compose\_proxy\_server\_1 ... done
4. Removing compose\_web\_server\_1 ... done
5. Removing compose\_proxy\_server\_1 ... done
6. Removing network compose\_default

Check the list of containers running.

1. $ docker ps
2. CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

docker-compose down command has stopped and deleted both the containers.

Note: If the default way of naming the containers as <directory\_name>\_<service\_name> has to be overridden, use the -p option while building up the containers. -p option is used to add the project name as part of the containers' names.

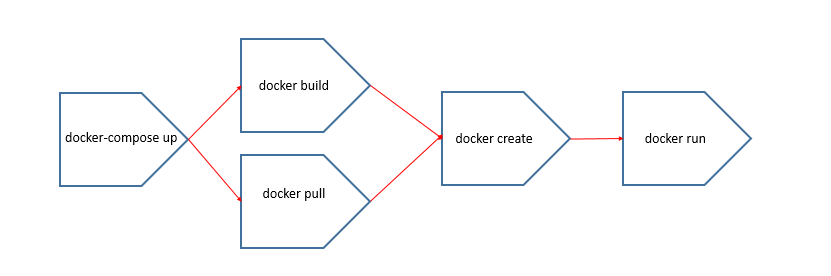
1. $ docker-compose -p project1 up -d
2. Creating network "project1\_default" with the default driver
3. Building web\_server
4. Step 1/1 : FROM tomcat
5. ---> 6055d4d564e1
6. Successfully built 6055d4d564e1
7. Successfully tagged project1\_web\_server:latest
8. WARNING: Image for service web\_server was built because it did not already exist. To rebuild this image you must use `docker-compose build` or `docker-compose up --build`.
9. Creating project1\_proxy\_server\_1 ... done
10. Creating project1\_web\_server\_1 ... done

Check the names of the containers now.

1. $ docker ps
2. CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
3. a5519a63e611 project1\_web\_server "catalina.sh run" 25 seconds ago Up 23 seconds 8080/tcp project1\_web\_server\_1
4. f5d4926cb109 nginx "/docker-entrypoint.…" 25 seconds ago Up 22 seconds 80/tcp project1\_proxy\_server\_1

The names of the containers are now <project\_name>\_<service\_name>.

As we have seen in the demo, a simple docker-compose command is enough to perform the operations of docker pull or docker build, docker create and docker run internally.



==========================================================

# Container network drivers

Docker containers have their own network namespaces. This means that their network resources stand isolated from the host or other containers. Communication outside the container is not possible usually. But the fact that Docker containers can be connected together and even connected with non-Docker instances is what makes them so powerful.

Container network drives come into picture here. These network drivers offer the containers, network functionality and communication with the outside environment up to a certain degree.

The Docker Engine supports many different network drivers. The built-in drivers include:

* bridge (default)
* none
* host
* container

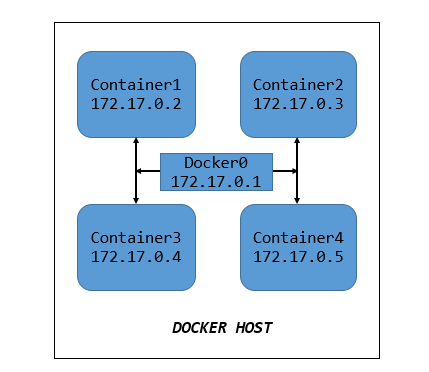
The network driver can be selected with the use of the following command.

1. *# docker run --net none*

We will discuss these network drivers in the upcoming modules.

# Bridge network driver

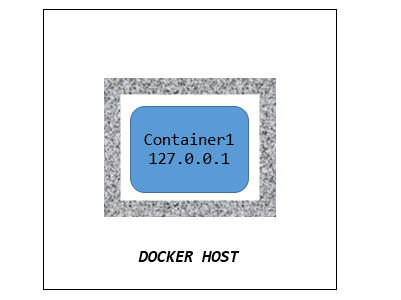
Bridge network driver is the default network driver a container gets. This is the type of network that gets created, if no driver is specified. This implies that, by default, the container gets a virtual eth0 interface. (In addition to its own private lo loopback interface.)



* The interface is provided by a veth pair.
* An isolated network is created for the container and it is connected to the Docker bridge to connect it to the host’s network.
* The interface is named docker0 by default.
* The container’s addresses are allocated on a private, internal subnet. By default, Docker uses 172.17.0.0/16 subnet.
* Outbound traffic goes through an iptables MASQUERADE rule and Inbound traffic goes through an iptables DNAT rule.
* The container can have its own routes, iptables rules, etc.

# None network driver

The none network driver is used to disable all networking.



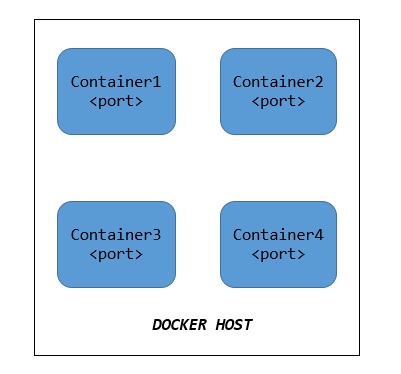
* To create this, the container is started as follows:

1. *# docker run --net none*

* When none network driver is created, the container gets only the loopback address, not eth0.
* The container cannot send or receive network traffic.
* This is used when we are planning to use an isolated/untrusted network driver.

# Host network driver

Host network drivers are used by standalone containers. The network isolation between the Docker host and the container is removed and the container uses the host’s network interfaces.This implies that, multiple containers on the same host will communicate like how processes on the host communicates instead of using a private/isolated network for communication.



To use the host network driver, the container is started as follows:

1. *# docker run --net host*

* ​​​It can bind any address, any port.
* Network traffic doesn't have to go through NAT, bridge, or veth.
* Used by performance sensitive applications such as VOIP, gaming, streaming etc.,

# Container network driver

* Container network driver is used when we want to re-use the network stack of another container.
* To use the container network driver, the container is started as follows:

1. *# docker run --net container*

* The container will share with this other container, the same interfaces, IP address(es), routes, iptables rules, etc.
* These containers communicate with each other over their lo interface. (i.e. one can bind to 127.0.0.1 and the others can connect to it.)

# Container Network Model

* The CNM was introduced in Engine 1.9.0 (November 2015).
* The Container Network Model standardizes the process of using multiple network drivers for providing network to the containers.
* The CNM has two interfaces for IPAM plugins and network plugins.
* IPAM plugin APIs - to allocate/deallocate IP addresses to containers and to create/ delete address pools.
* Network plugin APIs – to create/delete networks and to add/remove containers in the network.
* The CNM adds the notion of a network, and a new top-level command to manipulate and see those networks.

1. *# docker network ls*

# Components of a container network

* Conceptually, a container network is a virtual switch. A CNM has 5 main objects:
  + Network controller
  + Driver
  + Network
  + Endpoint
  + Sandbox
* The container’s network can be local (to a single Engine) or global (spanning multiple hosts) with a subnet associated to it.
* The container’s network is managed by a driver.
* Docker will allocate IP addresses to the containers connected to a network.
* Containers can be connected to multiple networks and given network specific names and aliases.
* Docker engine resolves the names and aliases using the DNS resolver.
* A new multi-host driver, overlay, is available out of the box. More drivers can be provided by
* plugins (OVS, VLAN...)
* The IPAM and network plugins are used by the CNM to allocate IP addresses to containers, to add containers to a network and for other networking purposes.

# Creating and working with a network

In this demo, we will see how to create a new network and work with it.

# STEP 1: Create a network

Let us now create a new network with the name “newnet”.

1. *# docker network create newnet*

The newly created network can be checked using the following command:

1. *# docker network ls*

# STEP 2: Place container on the network

We will now create a new nginx container, “new1” and connect it to the network “newnet”.

1. *# docker run -d --name new1 --net newnet nginx*

# STEP 3: Create another container

Create a new centos container and connect it to the same network “newnet”

1. *# docker run -ti --net newnet centos*

We are now inside the container environment. Let us try to ping the container “new1”, which is in the same network(newnet) as this container.

1. *# ping -c 2 web1*

We can see that from this new container, we can resolve and ping the other one, using its assigned name. Docker Engine uses dynamic resolver to resolve these names.

Using the container network model, the containers inside the same network are able to communicate with each other.

# What are volumes?

Now that we are comfortable with creating a container, starting it, stopping it and a few other actions, we will have a deeper look at what is happening inside a container. Containers, as we know, have their filesystem. What happens to the data inside the container once the container is destroyed? Can we share data from one container to another?

Once a container is destroyed, its filesystem is destroyed too thereby making it impossible to persist the data inside the container.

Docker offers two ways to solve this problem:

# 1.Bind mounts:

In bind mount, a directory in the host machine is mapped to a directory in the container filesystem. This directory does not get affected if the container gets destroyed. It can exist anywhere in the host machine making it vulnerable and available to processes outside the container.

# 2. Docker Volumes:

Docker volumes are the preferred mechanism to persist data inside the container and to share data among other containers. Volumes are also stored in the host filesystem but it is completely managed by Docker.

Volumes are often found to be a better option than persisting data in the container’s writable layer. Because, volumes’ contents exist outside the container’s lifecycle and they do not increase the size of the container

# Working with volumes

Docker volumes can be declared in two ways:

* Declaring volume inside the Docker file with a volume instruction.
* Declaring volume while running “docker run” command with -v flag (which will be dealt with in the upcoming demos).

In both the cases, the directory will be a volume inside the container.

# Significance of volumes

Volumes are the special directories inside a container that can be used to achieve the following:

* Sharing data between multiple containers
* Sharing a directory/file between host and container.
* Achieve native disk I/O performance by bypassing the copy-on-write system.
* Bypassing copy-on-write to leave some files out of docker commit.

# Understanding volumes

The logs of any container can be checked using the "docker logs" command. These logs are originally present inside the container and retrieved by the Docker engine when the "docker logs" command is executed. The location of these logs inside the container varies according to the container that is getting launched.

As mentioned above, the logs of a container can be checked using "docker logs" command which communicates with the container and gives us the result. Effectively, using docker volumes, we can access these logs present inside the container from the Docker host itself, This behaviour of Docker volumes is not restricted to accessing the logs of a container. It can be configured as to access any content present inside the container.

The following demo involves running two Nginx containers and check their access logs present inside the container using docker command and using volumes.

# I. Using docker command

# STEP 1: Start a Nginx container

Start a Nginx container. Before starting the container, make sure that there is no other container running in port no.80 (which is the default port for http service).

1. $ docker run --name "nginxsrv1" -d -p 7000:80 nginx
2. Unable to find image 'nginx:latest' locally
3. latest: Pulling from library/nginx
4. 8559a31e96f4: Pull complete
5. 1cf27aa8120b: Pull complete
6. 67d252a8c1e1: Pull complete
7. 9c2b660fcff6: Pull complete
8. 4584011f2cd1: Pull complete
9. Digest: sha256:a93c8a0b0974c967aebe868a186e5c205f4d3bcb5423a56559f2f9599074bbcd
10. Status: Downloaded newer image for nginx:latest
11. f081814e27cd3401e89d39482f932c85423607d98b14e70683ee4426d46d11ea

Check whether the Nginx container is up and running.

1. $ docker ps -l
2. CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
3. f081814e27cd nginx "/docker-entrypoint.…" About a minute ago Up 59 seconds 0.0.0.0:7000->80/tcp nginxsrv1

# STEP 2: Check the access log

The access log of a container is present inside the container and it can be checked by using the following docker command.

1. $ docker logs --tail 1 --follow f081814e27cd
2. /docker-entrypoint.sh: Configuration complete; ready for start up

While this command is up, connect to http://192.168.10.2:7000. (192.168.10.2 is the IP address of the Docker host used for this demo.)

1. $ docker logs --tail 1 --follow f081814e27cd
2. /docker-entrypoint.sh: Configuration complete; ready for start up
3. 172.17.0.5 - - [13/Jul/2020:05:13:18 +0000] "\x16\x03\x01\x00\xC7\x01\x00\x00\xC3\x03\x03d}\xB9}\xA0E]\xAAI\xFEg\x087\xC0\xCE\x8A7\x0C\xEC\x02\xAD\x07\x07`\x106\x82\x01w\xC9y\xFF E\x1F\xA6\xE7\xA2Z\x7F\x1AqQ\xDEq"400 157 "-" "-" "-"
4. 172.17.0.5 - - [13/Jul/2020:05:13:18 +0000] "GET / HTTP/1.1" 200 612 "https://2886795291-ollie08.environments.katacoda.com/" "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/83.0.4103.116 Safari/537.36" "34.93.163.13, 35.201.124.219, 130.211.2.72, 35.186.156.29, 172.17.0.6"
5. 172.17.0.2 - - [13/Jul/2020:05:13:20 +0000] "\x16\x03\x01\x00\xC7\x01\x00\x00\xC3\x03\x03o#\xB6\x0C0z\x910zl\x10\x9E.S\xE6\x5C\xEDE\xFE5.6\xF3\xFD\x19\xCE\xE0\xCCy\x1Dk7 \xB2\xA3\_\xE5\xF7z\xDF\_\x95\xBA\xE2\xFF\xA7Hy$-\xF3\x94l\x02:)\x0F\*%\x9CU\xD5b\x9C\xD7\x00 \xC0/\xC00\xC0+\xC0,\xCC\xA8\xCC\xA9\xC0\x13\xC0\x09\xC0\x14\xC0" 400 157 "-" "-" "-"
6. 2020/07/13 05:13:20 [error] 27*#27: \*4 open() "/usr/share/nginx/html/favicon.ico" failed (2: No such file or directory), client: 172.17.0.2, server: localhost, request: "GET /favicon.ico HTTP/1.1", host: "2886795291-7000-ollie08.environments.katacoda.com", referrer: "https://2886795291-7000-ollie08.environments.katacoda.com/"*
7. 172.17.0.2 - - [13/Jul/2020:05:13:20 +0000] "GET /favicon.ico HTTP/1.1" 404 555 "https://2886795291-7000-ollie08.environments.katacoda.com/" "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/83.0.4103.116 Safari/537.36" "34.93.163.13, 35.201.124.219, 130.211.2.75, 35.186.156.29, 172.17.0.6"

Using the docker command, you will be able to see the access logs of the container which is present inside the container. But for this, we have to access the container everytime. Using docker volumes, we can access the content present inside a container without even accessing it.

# II. Using docker volumes

In this demo, we are going to mount a local directory in the Docker host in a directory inside the Docker container. The content of the directory inside the container can now be accessed through the local directory in the Docker host. This magic is done by using the concept of bind mounts in Docker.

# STEP 1: Create a directory

Create am empty local directory in the Docker host.

1. $ mkdir -p /var/tmp/nginx/logs
2. $ ls -l /var/tmp/nginx/logs
3. total 0

Step 2: Start an Nginx container

Now, start an Nginx container with the newly created directory in the Docker host (/var/tmp/nginx/logs) mounted as the log directory of the container (/var/log/nginx). To do this, execute the following command.

1. *# docker run --name "nginxsrv2" -d -p 8000:80 \-v /var/tmp/nginx/logs:/var/log/nginx docker.io/nginx*

When you check the contents of the local directory now, you will notice that it got autopopulated with two log files which are actually present inside the container's log directory.

1. *# ls -l /var/tmp/nginx/logs*

# STEP 2: Check the access log

Connect to http://192.168.10.2:8000. (192.168.10.2 is the IP address of the Docker host used for this demo.)

Check the access logs of the container from the local directory of the Docker host.

1. *# tailf /var/tmp/nginx/logs/access.log*

The above scenario implies that using bind mount, you could access the logs of the container from the local directory of the Docker host. Likewise, volumes can be shared from host to container and also across multiple containers.

Docker Volume Management

# Sharing volume from host to containers

Docker volumes allows us to share data from host to containers. In this demo, we will launch an Nginx container with a directory from host which contains the index file required for the web server, mounted as the index directory of the container.

# STEP 1: Create a directory in Docker host

Create a local directory in the Docker host.

1. $ mkdir -p apps/website/nginx/html

# Step 2: Launch two Nginx containers

Run two Nginx containers by mounting the newly created directory inside the container. This directory will contain the index page required for the web service inside the container.

1. $ docker run --name "nginxsrv3" -d -p 9000:80 -v /apps/website/nginx/html:/usr/share/nginx/html nginx
2. f7a361634c4e87e62831396ffae31eb39dd6f140ce0b7b384d0791c48118552d
3. $ docker run --name "nginxsrv4" -d -p 9001:80 -v /apps/website/nginx/html:/usr/share/nginx/html nginx
4. 7fc80488e849ea885a15fb7133211d1ebb54ee2a12d94a06b18c4602fa6a1213

Check whether your new containers are up and running.

1. $ docker ps
2. CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
3. 7fc80488e849 nginx "/docker-entrypoint.…" 26 seconds ago Up 25 seconds 0.0.0.0:9001->80/tcp nginxsrv4
4. f7a361634c4e nginx "/docker-entrypoint.…" 53 seconds ago Up 51 seconds 0.0.0.0:9000->80/tcp nginxsrv3

STEP 3: Access the web page

Create an index file under the local directory in the Docker host, /apps/website/nginx/html/index.html.

1. $ echo "This is from the Nginx server" > apps/website/nginx/html/index.html

Now, access the web pages using the link, http://192.168.10.2:9000

You are now able to see the web page using the html page present the html page present in the local directory of the host.

# Sharing volume across containers

Volumes can be shared across containers. We can start a container with the same volumes as that of another one.  The new container will have the same volumes, in the same directories. They both will contain the same thing and remain in sync. Technically, they are the same directories in the host.

In docker, this can be done using the –volumes-from command.

Follow the steps below in order to share volumes across containers.

# STEP 1: Creating an independent volume

Create an independent volume in the docker host machine. This is the volume that we are going to share it with containers.

1. *# docker volume create --name Volume1*

Now, we have created an independent volume that can be attached to any number of containers.  But the objective behind sharing volumes across containers is technically to share the data present in the volume of one container across other containers.

# STEP 2: Attach it to a container

Attach the independent volume, Volume1 to a container.

1. *# docker run -ti --rm -v Volume1:/container\_volume1 centos*

In the above command,

--rm is used so that the container gets automattically deleted when exited from it.

-v is used to mount the volume created. Following it, the volume name must be given, which is Volume1 here.

/container\_volume1 is the absolute path in the container where we want the volume to appear inside the container.

If this directory is not present inside the container, it will be created when the command runs. If it is present, the data will be hidden by the mounted volume.

Execute the following command to check the mount.

1. *# df -h /container\_volume1*

# STEP 3: Write some data

Now that the volume is successfully mounted inside the container, let us add some data to it.

1. *# echo "This is in Volume1" > /container\_volume1/ Volume1.txt*

Exit from the container now.(Press CTRL-P + CTRL -Q).

# STEP 4: Inspecting the volume

We have exited the container now which implies that the container would have been deleted (since –rm option is used).

Execute the following command to inspect the volume.

1. *# docker volume inspect Volume1*

Check the directory in the host as well.

1. *# ls -l /var/lib/docker/volumes/Volume1/\_data*

We can also check the data on the host at the path listed as the mountpoint. We should avoid altering it as it can cause data corruption if applications or containers are unaware of changes.

# STEP 5: Attach the volume to a new container

Now, we will attach the volume, Volume1 to a new container.

1. *# docker run --rm -ti -v Volume1:/container\_volume1 centos*

Execute the following command to check the mount.

1. *# df -h /container\_volume1*

In order to check whether the cvolume is shared successfully, let us try to access the file we created in previous steps, Volume1\_file.txt.

1. *# cat /container\_volume1/Volume1.txt*

We are able to access the file created in the volume when it was attached to the previous container. This proves that docker volumes persist data.

In the coming demo, we will see how to create a volume along with the container creation.

# Mounting read-only volumes

In the previous demo, we mounted volumes of a container inside another container and both the containers were able to read and write from the data volume. Now, we will try to mount read-only volumes.

# Step 1: Create a new container with Volume4

Run a new centos container, Container6 and mount the volumes from Container4 as read-only volumes using the following command.

1. $ docker run -ti --name=Container6 --volumes-from Container4:ro centos
2. [root@ee3937e795d4 /]#

# Step 2: Access the files inside container\_volume4

Try to read the file created by us in the previous demo from Container4 and Container5.

1. [root@ee3937e795d4 /]# cat /container\_volume4/Container4.txt
2. This file is created in Container4
3. [root@ee3937e795d4 /]# cat /container\_volume4/Container5.txt
4. This is written from Container5

# Step 3: Delete any file inside container\_volume4

Now try to delete any of these files or write some data to the them.

1. [root@ee3937e795d4 /]# rm /container\_volume4/Container4.txt
2. rm: remove regular file '/container\_volume4/Container4.txt'? y
3. rm: cannot remove '/container\_volume4/Container4.txt': Read-only file system
4. [root@ee3937e795d4 /]# echo "This is written from Container6" >> /container\_volume4/Container5.txt
5. bash: /container\_volume4/Container5.txt: Read-only file system

From the above screenshots, it is evident that we are able to access contents of /container\_volume4. However since /container\_volume4 is mounted as a read-only volume, there is no way Container6 can make any changes to it.

Along with mounting volumes to multiple containers, Docker also allows us to mount multiple volumes to a single container, which is being dealt with in the upcoming demo.

# Mounting multiple volumes

Dealing with docker volumes also demands us to know the fact that multiple volumes can be mounted on a container. Some of those volumes can be mounted in read-only mode as well.

For this purpose, let us create few independent volumes, Volume5, Volume6 and Volume7. We will also attach these new independent volumes and volume from Container4 in read-only mode to our container.

# STEP 1: Create new volumes

Create Volume5, Volume6 and Volume7 as follows:

1. $ docker volume create --name Volume5
2. Volume5
3. $ docker volume create --name Volume6
4. Volume6
5. $ docker volume create --name Volume7
6. Volume7

# 

# STEP 2: Attach multiple volumes

Run a new centos container, Container7. Attach the newly created independent volumes to it. Along with those volumes, attach volumes from Container4 in read-only mode.

1. $ docker run -ti --name=Container7 --volumes-from Container4:ro -v Volume5:/container\_volume5 -vVolume6:/container\_volume6 -v Volume7:/container\_volume7 centos
2. [root@11145f163084 /]#

Check the list of docker volumes present using the following command.

1. $ docker volume ls
2. DRIVER VOLUME NAME
3. local Volume1
4. local Volume2
5. local Volume3
6. local Volume4
7. local Volume5
8. local Volume6
9. local Volume7

Thus, Docker lets us mount multiple volumes inside a container.

# Understanding the lifecycle of a docker volume.

* Docker volumes can be created independently or created along with container creation.
* Containers can be created with the same volumes as another container.
* Multiple volumes can be attached to a container.
* Docker volume cannot be created if it is used by a container.
* Docker volumes continue to stay even if the container gets removed.
* Docker is responsible for logging, monitoring, and backup of the volumes.

 What is docker more