- Docker is gaining popularity.
- The reason for Docker's growing popularity is the extent to which it can be used in an IT organization.
- Very few tools out there have the functionality to find itself useful to both developers and as well as system administrators.
- Docker is one such tool that truly lives up to its promise of **Build**, **Ship** and **Run**.
- In simple words, Docker is a software containerization platform, meaning you can build your application, package them along with their dependencies into a container and then these containers can be easily shipped to run on other machines.

For example: Lets consider a linux based application which has been written both in Ruby and Python. This application requires a specific version of linux, Ruby and Python. In order to avoid any version conflicts on user's end, a linux docker container can be created with the required versions of Ruby and Python installed along with the application. Now the end users can use the application easily by running this container without worrying about the dependencies or any version conflicts.

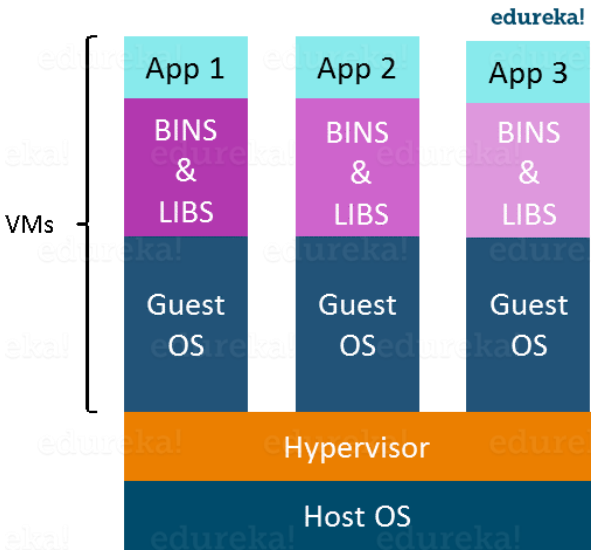
These containers uses Containerization which can be considered as an evolved version of Virtualization. The same task can also be achieved using Virtual Machines, however it is not very efficient.

what is the difference between Virtualization and Containerization? These two terms are very similar to each other.

What is Virtualization?

- Virtualization is the technique of importing a Guest operating system on top of a Host operating system.
- This technique was a revelation at the beginning because it allowed developers to run multiple operating systems in different virtual machines all running on the same host.
- This eliminated the need for extra hardware resource. The advantages of Virtual Machines or Virtualization are:

- Multiple operating systems can run on the same machine
- Maintenance and Recovery were easy in case of failure conditions
- Total cost of ownership was also less due to the reduced need for infrastructure



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

As you know nothing is perfect,

- Virtualization also has some shortcomings.
- Running multiple Virtual Machines in the same host operating system leads to performance degradation. This is because of the guest OS running on top of the host OS, which will have its own kernel and set of libraries and dependencies. This takes up a large chunk of system resources, i.e. hard disk, processor and especially RAM.
- Another problem with Virtual Machines which uses virtualization is that it takes almost a minute to boot-up. This is very critical in case of real-time applications.

Following are the disadvantages of Virtualization:

- Running multiple Virtual Machines leads to unstable performance
- Hypervisors are not as efficient as the host operating system
- Boot up process is long and takes time

These drawbacks led to the emergence of a new technique called Containerization.

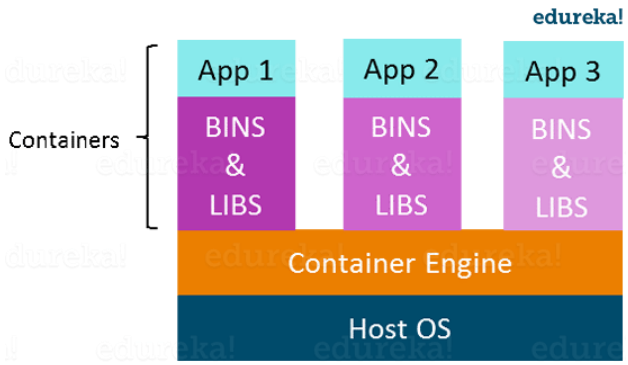
What is Containerization?

Containerization is the technique of bringing virtualization to the operating system level. While Virtualization brings abstraction to the hardware,

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

Advantages of Containerization over Virtualization:

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

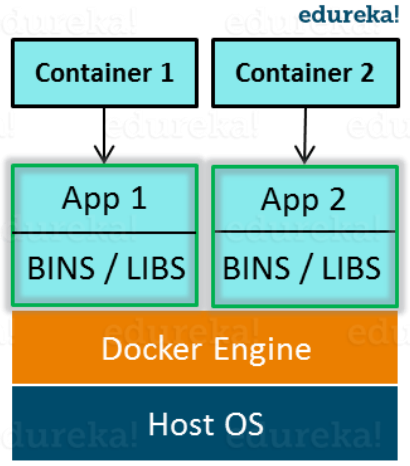


In the diagram on the right, you can see that there is a host operating system which is shared by all the containers. Containers only contain application specific libraries which are separate for each container and they are faster and do not waste any resources.

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

Docker Tutorial – Introduction To Docker

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



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

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

Benefits of Docker

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

Virtualization vs Containerization

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

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

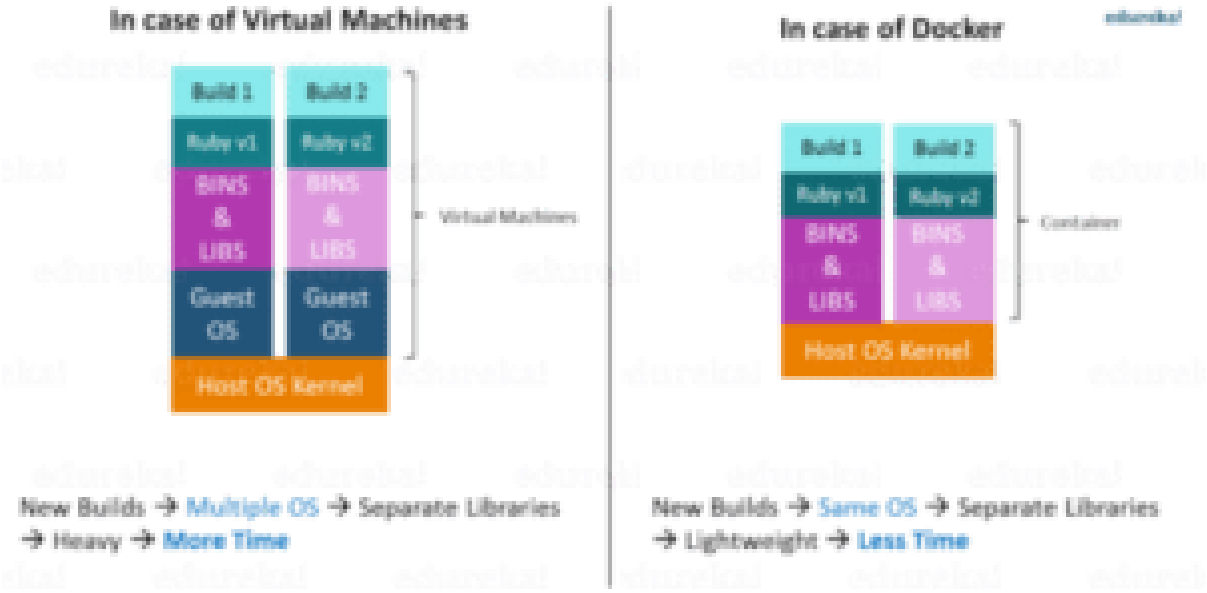


Figure: *What Is Big Data Analytics – Virtualization versus Containerization*

As we can see from the image, the major difference is that there are multiple Guest Operating Systems in Virtualization which are absent in Containerization. The best part of Containerization is that it is very lightweight as compared to the heavy virtualization.

Now, let us install Docker.

Install Docker:

I will be installing Docker in my Ubuntu 17.10 machine. Following are the steps to install Docker:

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

1. Install Required Packages:

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

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

```
edureka@edureka-VirtualBox:~$ sudo apt-get install  curl  apt-transport-https ca-certificates software-pro
perties-common
Reading package lists... Done
Building dependency tree
Reading state information... Done
ca-certificates is already the newest version (20170717).
software-properties-common is already the newest version (0.96.24.17).
The following additional packages will be installed:
  libcurl3
The following NEW packages will be installed:
  apt-transport-https curl
The following packages will be upgraded:
  libcurl3
```

2. Setup Docker Repository:

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

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

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

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

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

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

3. Install Docker On Ubuntu:

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

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

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

```
edureka@edureka-VirtualBox:~$ sudo apt-get install docker-ce
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  aufs-tools cgroupfs-mount git git-man liberror-perl pigz
Suggested packages:
  git-daemon-run | git-daemon-sysvinit git-doc git-el git-email git-gui gitk gitweb git-cvs
  git-mediawiki git-svn
The following NEW packages will be installed:
  aufs-tools cgroupfs-mount docker-ce git git-man liberror-perl pigz
0 upgraded, 7 newly installed, 0 to remove and 229 not upgraded.
Need to get 44.3 MB of archives.
```

Congratulations! You have successfully installed Docker. Also, check out a few commonly used **Docker Commands**.

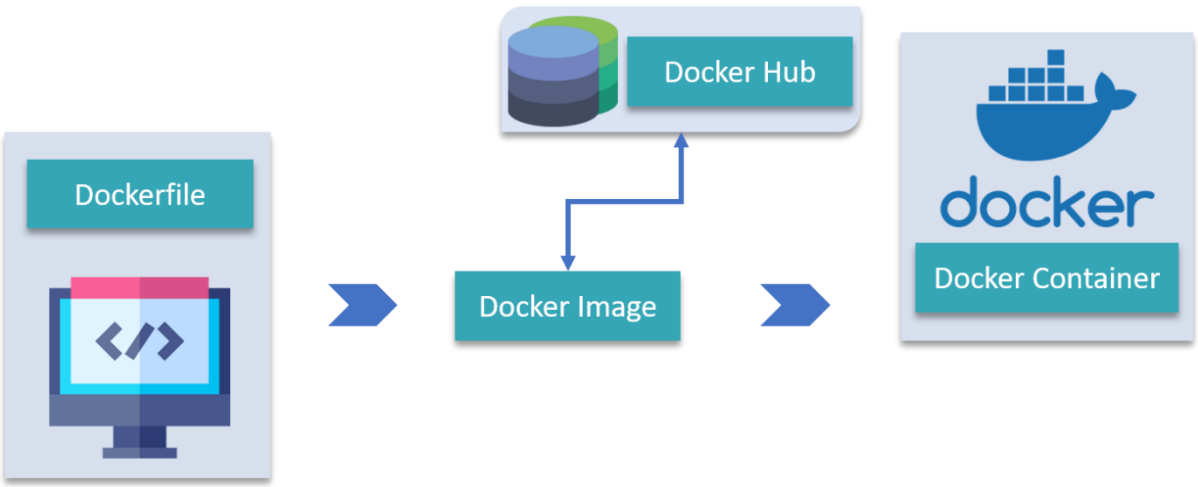
Now let us see a few important Docker concepts.

Dockerfile, Docker Image And Docker Container:

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

Docker Hub:

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



Docker Architecture:

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

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

Docker Compose:

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

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

How do containers communicate?

First, a quick overview! Although containers have a level of isolation from the environment around them, they often need to communicate with each other, and the outside world.

Networking or file sharing?

Two containers can talk to each other in one of two ways, usually:

- **Communicating through networking:** Containers are designed to be isolated. But they can send and receive requests to other applications, using networking.

  For example: a web server container might expose a port, so that it can receive requests on port 80. Or an application container might make a connection to a database container.

- **Sharing files on disk:** Some applications communicate by reading and writing files. These kinds of applications can communicate by writing their files into a **volume**, which can also be shared with other containers.

  For example: a data processing application might write a file to a shared volume which contains customer data, which is then read by another application. Or, two identical containers might even share the same files.

File sharing is great, but…., we'll look at applications that use networking as the primary way they either expose or consume services.

Communication between containers with networking

Most container-based applications talk to each other using networking. This basically means that an application running in one container will create a network connection to a port on another container.

For example, an application might call a REST or GraphQL API, or open a connection to a database.

Containers are ideal for applications or processes which expose some sort of network service. The most well-known examples of these kinds of applications are:

- Web servers - e.g. Nginx, Apache
- Backend applications and APIs - e.g. Node, Python, JBoss, Wildfly, Spring Boot
- Databases and data stores - e.g. MongoDB, PostgreSQL

There are more examples, but these are probably the most common ones!

With Docker, container-to-container communication is usually done using a virtual network.

docker images

List images

In Docker, everything is based on Images. An image is a combination of a file system and parameters. Let's take an example of the following command in Docker.

```
docker run hello-world
```

- The Docker command is specific and tells the Docker program on the Operating System that something needs to be done.
- The **run** command is used to mention that we want to create an instance of an image, which is then called a **container**.
- Finally, "hello-world" represents the image from which the container is made.

Now let's look at how we can use the CentOS image available in Docker Hub to run CentOS on our Ubuntu machine. We can do this by executing the following command on our Ubuntu machine −

```
sudo docker run -it centos /bin/bash
```

Note the following points about the above **sudo** command −

- We are using the **sudo** command to ensure that it runs with **root** access.
- Here, **centos** is the name of the image we want to download from Docker Hub and install on our Ubuntu machine.
- **−it** is used to mention that we want to run in **interactive mode**.
- **/bin/bash** is used to run the bash shell once CentOS is up and running.

Displaying Docker Images

To see the list of Docker images on the system, you can issue the following command.

```
docker images
```

This command is used to display all the images currently installed on the system.

Syntax

```
docker images
```

Options

None

Return Value

The output will provide the list of images on the system.

Example

```
sudo docker images
```

Output

When we run the above command, it will produce the following result −

```
demo@ubuntuserver:~$ sudo docker images
[sudo] password for demo:
REPOSITORY          TAG                 IMAGE ID            CREATED
VIRTUAL SIZE
newcentos           latest              7a86f8ffcb25        9 days ago
196.5 MB
jenkins             latest              998d1854867e        2 weeks ago
714.1 MB
centos              latest              97cad5e16cb6        4 weeks ago
196.5 MB
demo@ubuntuserver:~$ _
```

From the above output, you can see that the server has three images: **centos, newcentos,** and **jenkins**. Each image has the following attributes −

- **TAG** − This is used to logically tag images.
- **Image ID** − This is used to uniquely identify the image.
- **Created** − The number of days since the image was created.
- **Virtual Size** − The size of the image.

Downloading Docker Images

Images can be downloaded from Docker Hub using the Docker **run** command. Let's see in detail how we can do this.

Syntax

The following syntax is used to run a command in a Docker container.

```
docker run image
```

Options

- **Image** − This is the name of the image which is used to run the container.

Return Value

The output will run the command in the desired container.

Example

```
sudo docker run centos
```

This command will download the **centos** image, if it is not already present, and run the OS as a container.

Output

When we run the above command, we will get the following result −

```
demo@ubuntuserver:~$ sudo docker run centos
Unable to find image 'centos:latest' locally
latest: Pulling from centos

3690474eb5b4: Pull complete
af0819ed1fac: Pull complete
05fe84bf6d3f: Pull complete
97cad5e16cb6: Pull complete
Digest: sha256:934ff980b04db1b7484595bac0c8e6f838e1917ad3a38f904ece64f70bbc
Status: Downloaded newer image for centos:latest
demo@ubuntuserver:~$ _
```

You will now see the CentOS Docker image downloaded. Now, if we run the Docker **images** command to see the list of images on the system, we should be able to see the **centos** image as well.

```
demo@ubuntuserver:~$ sudo docker run centos
Unable to find image 'centos:latest' locally
latest: Pulling from centos

3690474eb5b4: Pull complete
af0819ed1fac: Pull complete
05fe84bf6d3f: Pull complete
97cad5e16cb6: Pull complete
Digest: sha256:934ff980b04db1b7484595bac0c8e6f838e1917ad3a38f904ece64f70bbc
Status: Downloaded newer image for centos:latest
demo@ubuntuserver:~$ sudo docker images
REPOSITORY          TAG                 IMAGE ID            CREATED
VIRTUAL SIZE
jenkins             latest              998d1854867e        2 weeks ago
714.1 MB
centos              latest              97cad5e16cb6        4 weeks ago
196.5 MB
demo@ubuntuserver:~$
```

Removing Docker Images

The Docker images on the system can be removed via the **docker rmi** command. Let's look at this command in more detail.

```
docker rmi
```

This command is used to remove Docker images.

Syntax

```
docker rmi ImageID
```

Options

- **ImageID** − This is the ID of the image which needs to be removed.

Return Value

The output will provide the Image ID of the deleted Image.

Example

```
sudo docker rmi 7a86f8ffcb25
```

Here, **7a86f8ffcb25** is the Image ID of the **newcentos** image.

Output

When we run the above command, it will produce the following result −

```
demo@ubuntuserver:~$ sudo docker rmi 7a86f8ffcb25
Untagged: newcentos:latest
Deleted: 7a86f8ffcb258e42c11d971a04b1145151b80122e566bc2b544f8fc3f94caf1e
demo@ubuntuserver:~$
```

Let’s see some more Docker commands on images.

docker images -q

This command is used to return only the Image ID’s of the images.

Syntax

```
docker images
```

Options

- **q** − It tells the Docker command to return the Image ID’s only.

Return Value

The output will show only the Image ID’s of the images on the Docker host.

Example

```
sudo docker images -q
```

Output

When we run the above command, it will produce the following result −

```
demo@ubuntuserver:~$ sudo docker images -q
998d1854867e
97cad5e16cb6
demo@ubuntuserver:~$ _
```

docker inspect

This command is used see the details of an image or container.

Syntax

```
docker inspect Repository
```

Options

- **Repository** − This is the name of the Image.

Return Value

The output will show detailed information on the Image.

Example

```
sudo docker inspect jenkins
```

Output

When we run the above command, it will produce the following result −

or

Usage

```
$ docker images [OPTIONS] [REPOSITORY[:TAG]]
```

Refer to the options section for an overview of available `OPTIONS` for this command.

Description

The default `docker images` will show all top level images, their repository and tags, and their size.

Docker images have intermediate layers that increase reusability, decrease disk usage, and speed up `docker build` by allowing each step to be cached. These intermediate layers are not shown by default.

The SIZE is the cumulative space taken up by the image and all its parent images. This is also the disk space used by the contents of the Tar file created when you docker save an image.
An image will be listed more than once if it has multiple repository names or tags. This single image (identifiable by its matching IMAGE ID) uses up the SIZE listed only once.
For example uses of this command, refer to the examples section below.

Options

Name, shorthand	Default	Description
--all , -a		Show all images (default hides intermediate images)
--digests		Show digests
--filter , -f		Filter output based on conditions provided
--format		Pretty-print images using a Go template
--no-trunc		Don't truncate output
--quiet , -q		Only show image IDs

Examples

List the most recently created images

```
$ docker images

REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
<none>              <none>              77af4d6b9913        19 hours ago        1.089 GB
committ             latest              b6fa739cedf5        19 hours ago        1.089 GB
<none>              <none>              78a85c484f71        19 hours ago        1.089 GB
docker              latest              30557a29d5ab        20 hours ago        1.089 GB
<none>              <none>              5ed6274db6ce        24 hours ago        1.089 GB
postgres            9                   746b819f315e        4 days ago          213.4 MB
postgres            9.3                 746b819f315e        4 days ago          213.4 MB
```

```
postgres            9.3.5               746b819f315e        4 days ago          213.4 MB
postgres            latest              746b819f315e        4 days ago          213.4 MB
```

List images by name and tag

The `docker images` command takes an optional `[REPOSITORY[:TAG]]` argument that restricts the list to images that match the argument. If you specify `REPOSITORY`but no `TAG`, the `docker images` command lists all images in the given repository.

For example, to list all images in the “java” repository, run this command :

```
$ docker images java

REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
java                8                   308e519aac60        6 days ago          824.5 MB
java                7                   493d82594c15        3 months ago        656.3 MB
java                latest              2711b1d6f3aa        5 months ago        603.9 MB
```

The `[REPOSITORY[:TAG]]` value must be an “exact match”. This means that, for example, `docker images jav` does not match the image `java`.

If both `REPOSITORY` and `TAG` are provided, only images matching that repository and tag are listed. To find all local images in the “java” repository with tag “8” you can use:

```
$ docker images java:8

REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
java                8                   308e519aac60        6 days ago          824.5 MB
```

If nothing matches `REPOSITORY[:TAG]`, the list is empty.

```
$ docker images java:0

REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
```

List the full length image IDs

```
$ docker images --no-trunc

REPOSITORY                    TAG                 IMAGE ID                                                                  CREATED             SIZE
<none>                        <none>              sha256:77af4d6b9913e693e8d0b4b294fa62ade6054e6b2f1ffb617ac955dd63fb0182   19 hours ago        1.089 GB
committest                    latest              sha256:b6fa739cedf5ea12a620a439402b6004d057da800f91c7524b5086a5e4749c9f   19 hours ago        1.089 GB
```

```
<none>                  <none>
sha256:78a85c484f71509adeaace20e72e941f6bdd2b25b4c75da8693efd9f61a37921   19 hours ago        1.089
GB

docker                  latest
sha256:30557a29d5abc51e5f1d5b472e79b7e296f595abcf19fe6b9199dbbc809c6ff4   20 hours ago        1.089
GB

<none>                  <none>
sha256:0124422dd9f9cf7ef15c0617cda3931ee68346455441d66ab8bdc5b05e9fdce5   20 hours ago        1.089
GB

<none>                  <none>
sha256:18ad6fad340262ac2a636efd98a6d1f0ea775ae3d45240d3418466495a19a81b   22 hours ago        1.082
GB

<none>                  <none>
sha256:f9f1e26352f0a3ba6a0ff68167559f64f3e21ff7ada60366e2d44a04befd1d3a   23 hours ago        1.089 GB

tryout                  latest
sha256:2629d1fa0b81b222fca63371ca16cbf6a0772d07759ff80e8d1369b926940074   23 hours ago        131.5
MB

<none>                  <none>
sha256:5ed6274db6ceb2397844896966ea239290555e74ef307030ebb01ff91b1914df   24 hours ago        1.089
GB
```

List image digests

Images that use the v2 or later format have a content-addressable identifier called a `digest`. As long as the input used to generate the image is unchanged, the digest value is predictable. To list image digest values, use the `--digests` flag:

```
$ docker images --digests

REPOSITORY                  TAG                 DIGEST                                                                    IMAGE ID
CREATED             SIZE

localhost:5000/test/busybox     <none>
sha256:cbbf2f9a99b47fc460d422812b6a5adff7dfee951d8fa2e4a98caa0382cfbdbf   4986bf8c1536        9 weeks
ago         2.43 MB
```

When pushing or pulling to a 2.0 registry, the `push` or `pull` command output includes the image digest. You can `pull` using a digest value. You can also reference by digest in `create`, `run`, and `rmi` commands, as well as the `FROM` image reference in a Dockerfile.

Filtering

The filtering flag (`-f` or `--filter`) format is of "key=value". If there is more than one filter, then pass multiple flags (e.g., `--filter "foo=bar" --filter "bif=baz"`)

The currently supported filters are:

- dangling (boolean - true or false)
- label (`label=<key>` or `label=<key>=<value>`)
- before (`<image-name>[:<tag>]`, `<image id>` or `<image@digest>`) - filter images created before given id or references

- since (`<image-name>[:<tag>]`, `<image id>` or `<image@digest>`) - filter images created since given id or references
- reference (pattern of an image reference) - filter images whose reference matches the specified pattern

Show untagged images (dangling)

```
$ docker images --filter "dangling=true"

REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
<none>              <none>              8abc22fbb042        4 weeks ago         0 B
<none>              <none>              48e5f45168b9        4 weeks ago         2.489 MB
<none>              <none>              bf747efa0e2f        4 weeks ago         0 B
<none>              <none>              980fe10e5736        12 weeks ago        101.4 MB
<none>              <none>              dea752e4e117        12 weeks ago        101.4 MB
<none>              <none>              511136ea3c5a        8 months ago        0 B
```

This will display untagged images that are the leaves of the images tree (not intermediary layers). These images occur when a new build of an image takes the `repo:tag` away from the image ID, leaving it as `<none>:<none>` or untagged. A warning will be issued if trying to remove an image when a container is presently using it. By having this flag it allows for batch cleanup.

You can use this in conjunction with `docker rmi ...`:

```
$ docker rmi $(docker images -f "dangling=true" -q)

8abc22fbb042
48e5f45168b9
bf747efa0e2f
980fe10e5736
dea752e4e117
511136ea3c5a
```

Docker warns you if any containers exist that are using these untagged images.

Show images with a given label

The `label` filter matches images based on the presence of a `label` alone or a `label` and a value.

The following filter matches images with the `com.example.version` label regardless of its value.

```
$ docker images --filter "label=com.example.version"

REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
match-me-1          latest              eeae25ada2aa        About a minute ago  188.3 MB
match-me-2          latest              dea752e4e117        About a minute ago  188.3 MB
```

The following filter matches images with the `com.example.version` label with the `1.0` value.

```
$ docker images --filter "label=com.example.version=1.0"

REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
match-me            latest              511136ea3c5a        About a minute ago  188.3 MB
```

In this example, with the `0.1` value, it returns an empty set because no matches were found.

```
$ docker images --filter "label=com.example.version=0.1"
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
```

Filter images by time

The `before` filter shows only images created before the image with given id or reference. For example, having these images:

```
$ docker images

REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
image1              latest              eeae25ada2aa        4 minutes ago       188.3 MB
image2              latest              dea752e4e117        9 minutes ago       188.3 MB
image3              latest              511136ea3c5a        25 minutes ago      188.3 MB
```

Filtering with `before` would give:

```
$ docker images --filter "before=image1"

REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
image2              latest              dea752e4e117        9 minutes ago       188.3 MB
image3              latest              511136ea3c5a        25 minutes ago      188.3 MB
```

Filtering with `since` would give:

```
$ docker images --filter "since=image3"
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
image1              latest              eeae25ada2aa        4 minutes ago       188.3 MB
image2              latest              dea752e4e117        9 minutes ago       188.3 MB
```

Filter images by reference

The `reference` filter shows only images whose reference matches the specified pattern.

```
$ docker images

REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
busybox             latest              e02e811dd08f        5 weeks ago         1.09 MB
busybox             uclibc              e02e811dd08f        5 weeks ago         1.09 MB
```

```
busybox         musl          733eb3059dce    5 weeks ago     1.21 MB
busybox         glibc         21c16b6787c6    5 weeks ago     4.19 MB
```

Filtering with `reference` would give:

```
$ docker images --filter=reference='busy*:*libc'

REPOSITORY      TAG           IMAGE ID        CREATED         SIZE
busybox         uclibc        e02e811dd08f    5 weeks ago     1.09 MB
busybox         glibc         21c16b6787c6    5 weeks ago     4.19 MB
```

Filtering with multiple `reference` would give, either match A or B:

```
$ docker images --filter=reference='busy*:uclibc' --filter=reference='busy*:glibc'

REPOSITORY      TAG           IMAGE ID        CREATED         SIZE
busybox         uclibc        e02e811dd08f    5 weeks ago     1.09 MB
busybox         glibc         21c16b6787c6    5 weeks ago     4.19 MB
```

Format the output

The formatting option (`--format`) will pretty print container output using a Go template.

Valid placeholders for the Go template are listed below:

Placeholder	Description
`.ID`	Image ID
`.Repository`	Image repository
`.Tag`	Image tag
`.Digest`	Image digest
`.CreatedSince`	Elapsed time since the image was created
`.CreatedAt`	Time when the image was created
`.Size`	Image disk size

When using the `--format` option, the `image` command will either output the data exactly as the template declares or, when using the `table` directive, will include column headers as well.

The following example uses a template without headers and outputs the `ID` and `Repository` entries separated by a colon (`:`) for all images:

```
$ docker images --format "{{.ID}}: {{.Repository}}"

77af4d6b9913: <none>
b6fa739cedf5: committ
78a85c484f71: <none>
30557a29d5ab: docker
5ed6274db6ce: <none>
746b819f315e: postgres
746b819f315e: postgres
746b819f315e: postgres
746b819f315e: postgres
```

To list all images with their repository and tag in a table format you can use:

```
$ docker images --format "table {{.ID}}\t{{.Repository}}\t{{.Tag}}"

IMAGE ID            REPOSITORY          TAG
77af4d6b9913        <none>              <none>
b6fa739cedf5        committ             latest
78a85c484f71        <none>              <none>
30557a29d5ab        docker              latest
5ed6274db6ce        <none>              <none>
```

A container creates a specific environment in which a process can be executed. Many containers are created, tested, and abandoned during the development lifecycle.

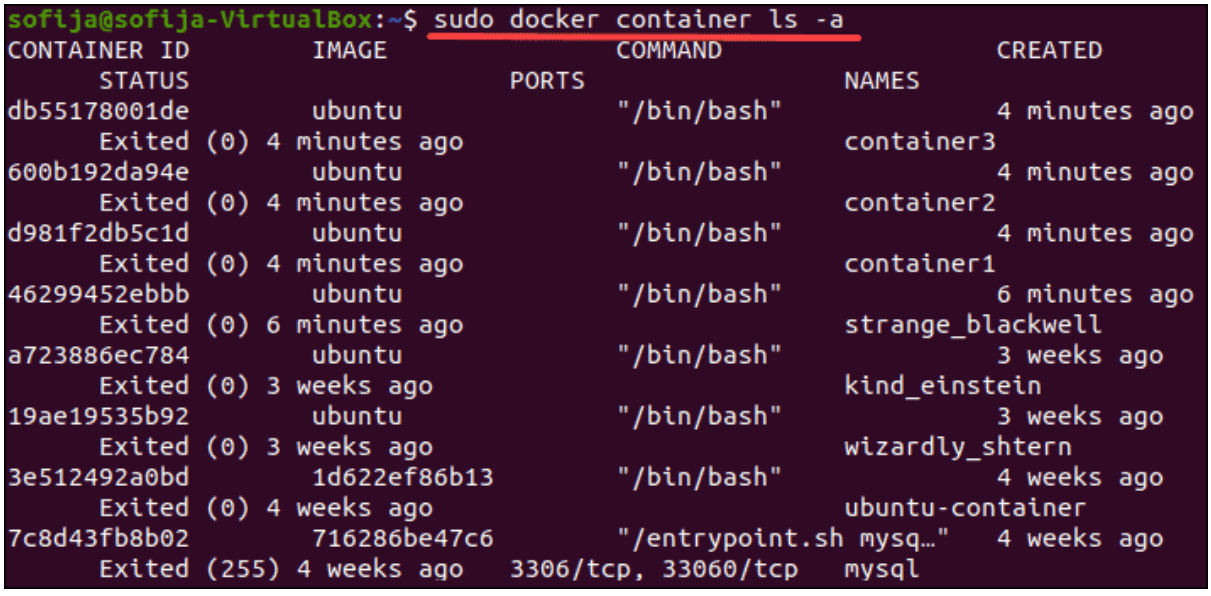
Therefore, it's important to know how to find unnecessary containers and remove them.

Stop Containers

1. First, list all Docker containers using the command:

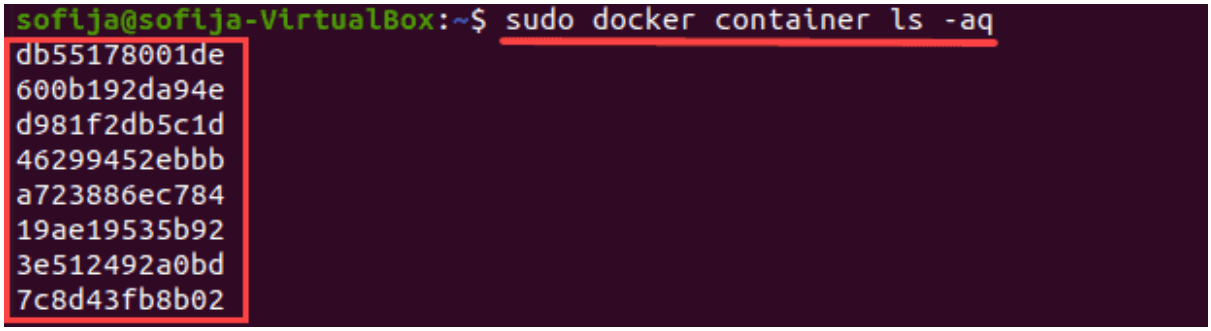
```
docker container ls -a
```

The output displays a list of all running containers, their IDs, names, images, status and other parameters.



You can also generates a list of all the containers only by their numeric ID's, run the command:

```
docker container ls -aq
```



2. To **stop a specific container**, enter the following:

```
docker container stop [container_id]
```

Replace `[container_id]` with the numeric ID of the container from your list.

You can enter multiple container IDs into the same command.

To **stop all containers**, enter:

```
docker container stop $(docker container ls -aq)
```

This forces Docker to use the list of all container IDs as the target of the `stop` command.

Note: If you are logged in as the **sudo** user, make sure to add the `sudo` prefix before both `docker` commands when stopping all containers (`sudo docker container stop $(sudo docker container ls -aq)`).

Remove a Stopped Container

To **remove a stopped container**, use the command:

```
docker container rm [container_id]
```

Like before, this removes a container with the ID you specify.

Remove All Stopped Containers

To **remove all stopped containers**:

```
docker container rm $(docker container ls -aq)
```

Remove All Docker Containers

To wipe Docker clean and start from scratch, enter the command:

```
docker container stop $(docker container ls -aq) && docker sys
tem prune -af --volumes
```

This instructs Docker to stop the containers listed in the parentheses.

Inside the parentheses, you instruct Docker to generate a list of all the containers with their numeric ID. Then, the information is passed back to the `container stop` command and stops all the containers.

The `&&` attribute instructs Docker to remove all stopped containers and volumes.

`-af` indicates this should apply to all containers (`a`) without a required confirmation (`f`).

Removing Container With Filters

You can also specify to delete all objects that do not match a specified label.

To do so, use the command:

```
docker container prune --filter="label!=maintainer=Jeremy"
```

This command tells Docker to **remove all containers** that are not labeled with a maintainer of "**jeremy**." The `!=` command is a logical notation that means "**not equal to**."

A breakdown of the `label` commands:

- `label=<key>`
- `label=<key>=<value>`
- `label!=<key>`
- `label!=<key>=<value>`

Using these terms in conjunction with labels gives you in-depth control over removing assets in Docker.

Manage data in Docker

By default all files created inside a container are stored on a writable container layer. This means that:

- The data doesn't persist when that container no longer exists, and it can be difficult to get the data out of the container if another process needs it.
- A container's writable layer is tightly coupled to the host machine where the container is running. You can't easily move the data somewhere else.
- Writing into a container's writable layer requires a storage driver to manage the filesystem. The storage driver provides a union filesystem, using the Linux kernel. This extra abstraction reduces performance as compared to using *data volumes*, which write directly to the host filesystem.

Docker has two options for containers to store files on the host machine, so that the files are persisted even after the container stops: *volumes*, and *bind mounts*.

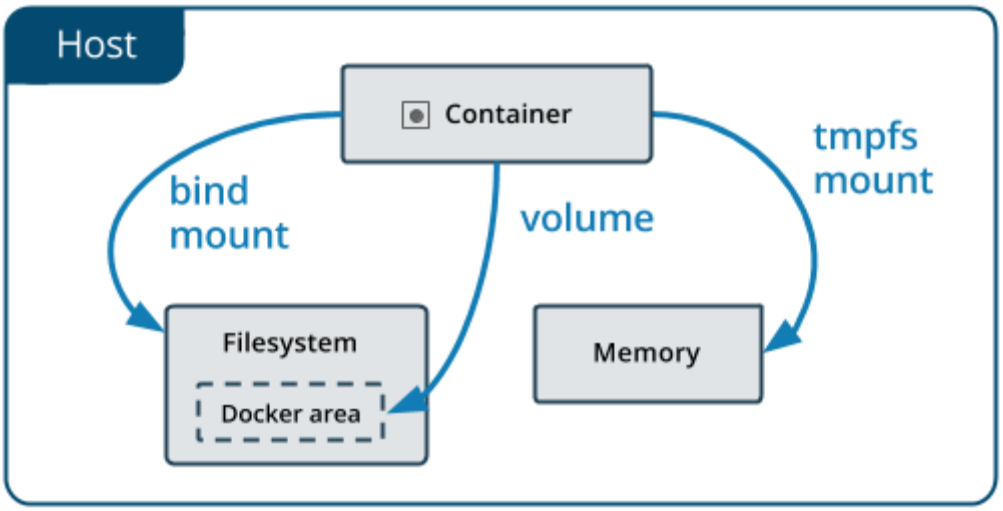
Docker also supports containers storing files in-memory on the the host machine. Such files are not persisted. If you're running Docker on Linux, *tmpfs mount* is used to store files in the host's system memory. If you're running Docker on Windows, *named pipe* is used to store files in the host's system memory.

Keep reading for more information about persisting data or taking advantage of in-memory files.

Choose the right type of mount

No matter which type of mount you choose to use, the data looks the same from within the container. It is exposed as either a directory or an individual file in the container's filesystem.

An easy way to visualize the difference among volumes, bind mounts, and `tmpfs` mounts is to think about where the data lives on the Docker host.



- **Volumes** are stored in a part of the host filesystem which is *managed by Docker* (`/var/lib/docker/volumes/` on Linux). Non-Docker processes should not modify this part of the filesystem. Volumes are the best way to persist data in Docker.
- **Bind mounts** may be stored *anywhere* on the host system. They may even be important system files or directories. Non-Docker processes on the Docker host or a Docker container can modify them at any time.
- `tmpfs` **mounts** are stored in the host system's memory only, and are never written to the host system's filesystem.

More details about mount types

- **Volumes**: Created and managed by Docker. You can create a volume explicitly using the `docker volume create` command, or Docker can create a volume during container or service creation.
  When you create a volume, it is stored within a directory on the Docker host. When you mount the volume into a container, this directory is what is mounted into the container. This is similar to the way that bind mounts work, except that volumes are managed by Docker and are isolated from the core functionality of the host machine.

  A given volume can be mounted into multiple containers simultaneously. When no running container is using a volume, the volume is still available to Docker and is not removed automatically. You can remove unused volumes using `docker volume prune`.
  When you mount a volume, it may be **named** or **anonymous**. Anonymous volumes are not given an explicit name when they are first mounted into a container, so Docker gives them a random name that is guaranteed to be unique within a given

Docker host. Besides the name, named and anonymous volumes behave in the same ways.

Volumes also support the use of *volume drivers*, which allow you to store your data on remote hosts or cloud providers, among other possibilities.

- **Bind mounts**: Available since the early days of Docker. Bind mounts have limited functionality compared to volumes. When you use a bind mount, a file or directory on the *host machine* is mounted into a container. The file or directory is referenced by its full path on the host machine. The file or directory does not need to exist on the Docker host already. It is created on demand if it does not yet exist. Bind mounts are very performant, but they rely on the host machine's filesystem having a specific directory structure available. If you are developing new Docker applications, consider using named volumes instead. You can't use Docker CLI commands to directly manage bind mounts.

  Bind mounts allow access to sensitive files

  One side effect of using bind mounts, for better or for worse, is that you can change the **host** filesystem via processes running in a **container**, including creating, modifying, or deleting important system files or directories. This is a powerful ability which can have security implications, including impacting non-Docker processes on the host system.

- **tmpfs mounts**: A `tmpfs` mount is not persisted on disk, either on the Docker host or within a container. It can be used by a container during the lifetime of the container, to store non-persistent state or sensitive information. For instance, internally, swarm services use `tmpfs` mounts to mount secrets into a service's containers.
- **named pipes**: An `npipe` mount can be used for communication between the Docker host and a container. Common use case is to run a third-party tool inside of a container and connect to the Docker Engine API using a named pipe.

Bind mounts and volumes can both be mounted into containers using the `-v` or `--volume` flag, but the syntax for each is slightly different. For `tmpfs` mounts, you can use the `--tmpfs` flag. We recommend using the `--mount` flag for both containers and services, for bind mounts, volumes, or `tmpfs` mounts, as the syntax is more clear.

Good use cases for volumes

Volumes are the preferred way to persist data in Docker containers and services. Some use cases for volumes include:

- Sharing data among multiple running containers. If you don't explicitly create it, a volume is created the first time it is mounted into a container. When that container stops or is removed, the volume still exists. Multiple containers can mount the same volume simultaneously, either read-write or read-only. Volumes are only removed when you explicitly remove them.
- When the Docker host is not guaranteed to have a given directory or file structure. Volumes help you decouple the configuration of the Docker host from the container runtime.
- When you want to store your container's data on a remote host or a cloud provider, rather than locally.
- When you need to back up, restore, or migrate data from one Docker host to another, volumes are a better choice. You can stop containers using the volume, then back up the volume's directory (such as `/var/lib/docker/volumes/<volume-name>`).
- When your application requires high-performance I/O on Docker Desktop. Volumes are stored in the Linux VM rather than the host, which means that the reads and writes have much lower latency and higher throughput.
- When your application requires fully native file system behavior on Docker Desktop. For example, a database engine requires precise control over disk flushing to guarantee transaction durability. Volumes are stored in the Linux VM and can make these guarantees, whereas bind mounts are remoted to macOS or Windows, where the file systems behave slightly differently.

Good use cases for bind mounts

In general, you should use volumes where possible. Bind mounts are appropriate for the following types of use case:

- Sharing configuration files from the host machine to containers. This is how Docker provides DNS resolution to containers by default, by mounting `/etc/resolv.conf` from the host machine into each container.
- Sharing source code or build artifacts between a development environment on the Docker host and a container. For instance, you may mount a Maven `target/` directory into a container, and each time you build the Maven project on the Docker host, the container gets access to the rebuilt artifacts.

If you use Docker for development this way, your production Dockerfile would copy the production-ready artifacts directly into the image, rather than relying on a bind mount.

- When the file or directory structure of the Docker host is guaranteed to be consistent with the bind mounts the containers require.

Good use cases for tmpfs mounts

`tmpfs` mounts are best used for cases when you do not want the data to persist either on the host machine or within the container. This may be for security reasons or to protect the performance of the container when your application needs to write a large volume of non-persistent state data.

Tips for using bind mounts or volumes

If you use either bind mounts or volumes, keep the following in mind:

- If you mount an **empty volume** into a directory in the container in which files or directories exist, these files or directories are propagated (copied) into the volume. Similarly, if you start a container and specify a volume which does not already exist, an empty volume is created for you. This is a good way to pre-populate data that another container needs.
- If you mount a **bind mount or non-empty volume** into a directory in the container in which some files or directories exist, these files or directories are obscured by the mount, just as if you saved files into `/mnt` on a Linux host and then mounted a USB drive into `/mnt`. The contents of `/mnt` would be obscured by the contents of the USB drive until the USB drive were unmounted. The obscured files are not removed or altered, but are not accessible while the bind mount or volume is mounted.