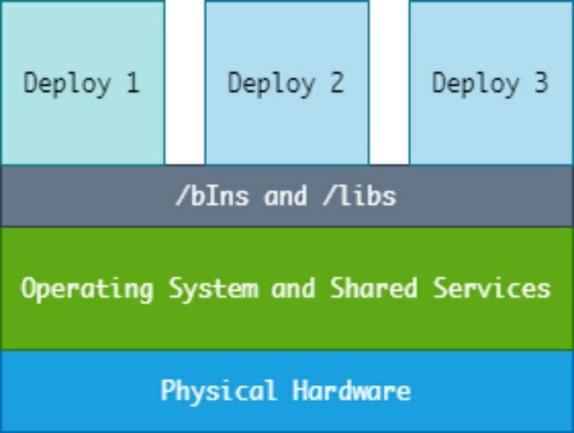
**Introduction**

In the earlier days of technological evolution, developers deployed applications directly on physical machines, with each equipped with an Operating System. Because of the single user-space, applications shared runtime.

Although deployment on physical machines was stable, the maintenance was long and arduous, more so when each host used a different operating system. There was no flexibility for developers and the hosted applications.

As you can imagine, this caused many issues when there was more than one application built that required regular maintenance and a standalone machine for it.

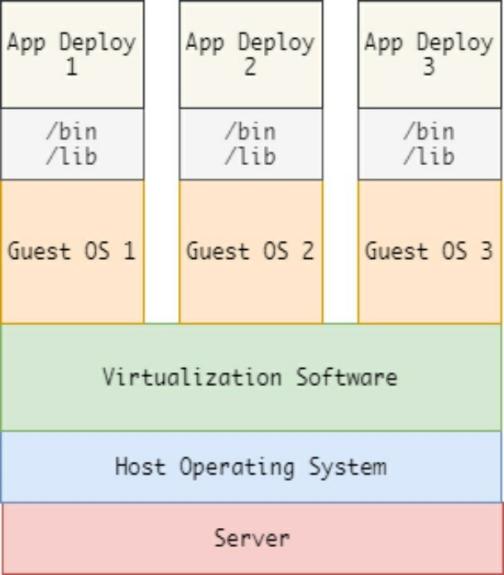


Because of the limitations of deploying applications on physical hardware and utilizing the resources of the entire host system, Virtualization technology came into being, which is when the dynamics of app development started changing.

Using tools known as hypervisors such as Hyper-V, ESX, KVM, VMware, and others, developers started being able to create virtual machines that they could use to deploy host operating systems (guest OS) on one physical machine.

Virtual machines have independent virtual machines, and applications deployed on virtual machines are what we call “isolated and standalone.” That means, thanks to the complete isolation of the entire infrastructure running the application, it’s possible to perform updates and patches on one instance of the application without affecting other applications.

The diagram below illustrates virtualization.



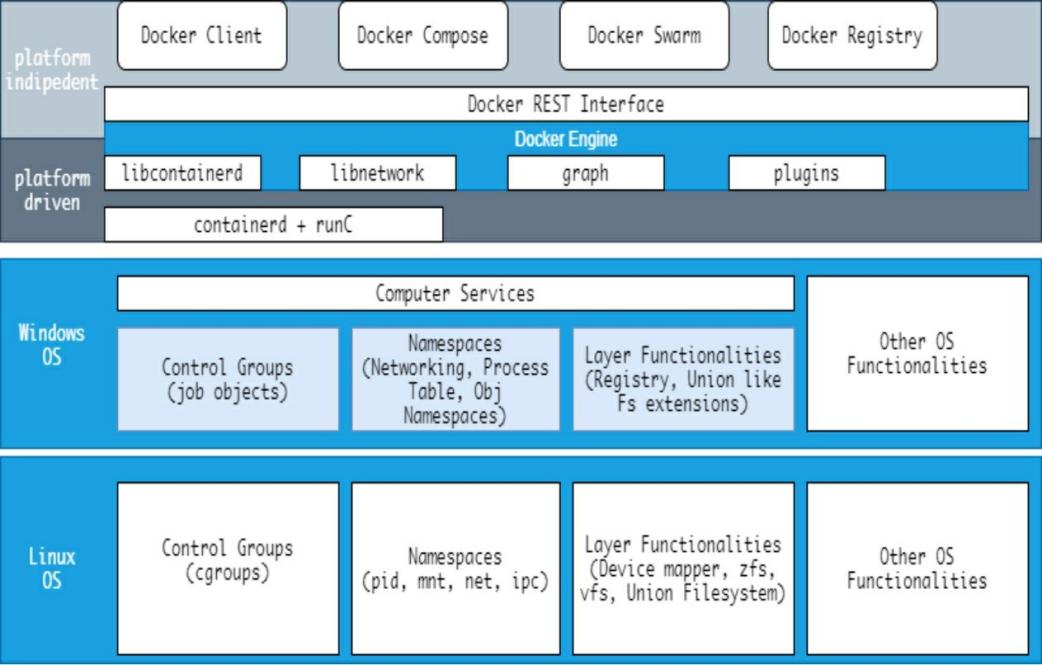
As technology grows, virtual machines, development tools that were once instrumental, are becoming redundant, in part because of the complexity of using Virtual Machines, but mainly because of the technological changes precipitated by the process of reducing hardware emulation. *Containers* are taking over the territory once owned by Virtual Machines.

By using Containers, developers can package applications with their standalone environments. Although compared to VM, Containers are less flexible because they rely on the host operating system; as we will discuss in upcoming chapters, they are a better alternative.

# Section 1: Docker Platform And Architecture

Docker is a containerization software developed by Solomon Hykes, the founder of Dot Cloud. It was initially meant to be an internal project. In March 2013, Docker became an open-source software using the Apache License.

Docker utilizes a host’s operating system kernel feature that allows containerization. We can illustrate Docker’s platform and kernel features using the image below.



To help you develop a deeper understanding of Docker better, let’s look at the facets that make up its core architecture:

## Namespaces

Namespaces are the main building blocks of Docker Containers. There are various types of namespaces, with one of them acting as an isolating block from one application to another. The creation of Namespaces happens through the clone system call. Developers sometimes also attach existing namespaces. Some of the namespaces used by Docker include:

The PID namespace The net namespace IPC namespace

UTS namespace MNT namespace User namespace

Let’s discuss each of the above namespaces in detail:

## The PID Namespace

The PID namespaces in Linux system isolates the process Identification Number space. That means processes using different PID namespaces can have the same PID.

The PID namespace allows containers to deliver functionalities like suspending and resuming the set of processes within the container. It also gives developers the ability to migrate the container from one host to another while the processes running within the container maintain their original PIDs.

PIDs in a new PID namespace start at 1, somewhat like a standalone system, and calls to fork , vfork , or clone will produce processes with PIDs that are unique within the namespace.

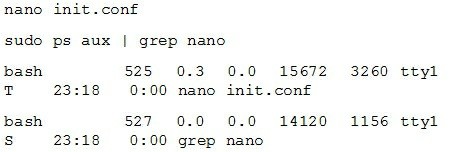
As mentioned, the PID namespaces allow each container to have an isolated set of PIDs. Each PID creates a unique process hierarchy.

A “parent” (main) namespace can manage the “children” namespaces and perform actions to alter their functionality. A child namespace, however, cannot see or perform any particular action on the parent namespace node.

Other than the initial (“root”) PID namespace, each of the nested namespaces has a parent. The parent in this case refers to the PID namespace of the process that actually created the namespace using unshare or clone . Thus, the PID namespaces make some sort of tree that contains all all namespaces that ultimately trace their ancestry to the root namespace.

If there are two levels of hierarchy, then at the top level, we would see the process running inside the child namespace with a different PID. Therefore, a process running within a child namespace usually has two PIDs: one for the parent namespace and one for the child namespace.

Let me give an illustration: If we start a program such as nano , the action creates a parent and child process:



You can find more information on PID namespace from the resource page below:

<http://man7.org/linux/man-pages/man7/pid_namespaces.7.html>

## The NET Namespace

The network namespace is a logical copy of the network stack. It has individual building blocks like firewall rules, network interfaces, and routing tables.

The default location for the network namespaces is:

/var/run/netns/NAME

By convention, a process usually inherits all its network namespace from the parent namespace, where, initially, the same namespace from the init process is shared by all processes.

By using the PID namespace only, we can run one program more than once in isolated environments. For example, we can run a service such as NGINX on different containers since the PIDs do not conflict, but we cannot communicate with the service on port 80 without using the net namespace.

The net namespaces make it possible to create multiple network interfaces on every container, thus allowing services to communicate on their respective ports. It is also good to note that loopback addresses on each container are unique.

For networking to occur within the containers, a special pair of network interfaces are created in two different net namespaces, allowing communication with each interface. Each interface resides inside a container, and the other resides inside a host operating system. The interface in the container is eth0, and the host interface gets allocated a random but unique name.

The interfaces interlink via a network bridge, which, in most cases, is docker0 that allows the communication between the container and the packets routes.

To learn more about the net namespace, check the Linux programmers page available here.

<http://man7.org/linux/man-pages/man8/ip-netns.8.html>

## The IPC Namespace

The IPC or inter-communication namespace helps isolate and provide System V IPC objects like semaphores, shared memory segments, and POSIX message ques within a system. Although the use of the IPC namespace is not common today, older, and even some new processes, still depend on it.

An IPC resource initialized by one container can be consumed or terminated by another container. If this happens, the application tied to the IPC resource on the first container fails. That is where the IPC namespace comes in hand: it prevents processes running in one namespace from accessing other resources of another IPC namespace.

## The USER Namespace.

The USER namespace isolates the security-based identifiers and attributes within a unique user ID or group ID. It is possible to nest user namespaces. Except for the “root” or initial namespace, each user namespace has a primary or parent user namespace and as many child user namespaces it needs (sometimes none).

The parent user namespace is the user namespace of the process that creates the user namespace via a call to unshare or clone with the CLONE\_NEWUSER flag.

Since the user namespace allows for the mapping of users and groups per namespace instance, therefore, using the user namespace, it makes it possible to have users with a zero identifier within a container and users with non-zero identifiers on the host.

## The UTS Namespace

The Linux UTS namespace provides an idea isolation mechanism for two system hostname and NIS domain name identifiers.

Linux users can retrieve the identifiers using the uname , gethostname or getdomainname . To set the identifiers, you can use the sethostname or setdomainname .

Any changes done on either variable become globally broadcasted and reflected on all processes in the same UTS namespace but not to other processes in other namespaces.

Thus, the UTS namespace allows developers to assign different hostnames on a single container. For more information on Linux UTS namespaces, check the following resource page:

<http://man7.org/linux/man-pages/man7/uts_namespaces.7.html>

## The MNT Namespace

Linux Systems use the MNT or mount namespaces to provide isolation for the list of mount points available to the processes in a single namespace instance. We create a mount namespace by using clone or unshare with the CLONE\_NEWNS flag.

Without the mnt namespace, you can only use the chroot operation to check the relative paths of a particular system from a chrooted directory/namespace .

However, with the mnt namespace, a single container can have its own individual set of mounted root directories and filesystem tree.

It’s worth noting that processes from one mnt namespace cannot interact with a mounted filesystem of another mnt namespace.

## Cgroups

Control groups, also called cgroups **,** refers to a feature that allows processes within a Linux System to be arrangeable in hierarchical groups. These groups make it possible to monitor and limit the usage of various resources on the system.

The cgroup interface in the kernel is available courtesy of a pseudo-filesystem called cgroupfs . The execution of the grouping mechanism occurs in the core kernel group code, and the resource monitoring and limiting is possible because of a set of per-resource-type subsystems such as Memory, CPU, etc.

Cgroups provide resource monitoring and limitation features for containers. They’re comparable to the ulimit and setrlimit system calls.

However, instead of performing limitations to a single process within the system, the cgroups makes it possible to perform limitations on a group of operations to various resources on a single system.

Control groups divide further into various subsystems such as Memory, CPU, CPU sets Input/Output memory blocks, Freezers, etc. Linux subsystems are usable independently but are also groupable.

A subsystem refers to a kernel component that modifies the original behavior in a cgroup ; we can also call them controllers or resource controllers.

Control groups for a resource controller (a subsystem) are set in a hierarchical paradigm defined by either CRUD (create, rename, or removing) directories with the cgroup filesystem.

Control groups provide features such as:

**Accounting**: This feature allows users to monitor and measure resource usage for various subsystems; the use of this feature is routine in billing.

**Resource Limitation:** This feature allows users to bound control groups to a specific subsystem, thus allowing the processes in the specified cgroup to run on the set subsystem.

**Control**: The Cgroups feature makes it possible to perform actions such as freezing and performing restarts on groups.

**Priority:** Control groups also have features that allow for prioritization, allowing you to allocate higher or lower subsystem share to some groups.

The main subsystems that are directly manageable by the control groups include:

**CPU:** The CPU limits are directly manageable by the groups.

**Memory:** Control groups can set limits on memory usage by the processes in the cgroups .

**Freezer:** The groups can also suspend or resume processes in a

group

**Blkio**: The block IO controller sets Input/Output access to and from the block devices within the system such as storage devices, etc.

**Cpuacct**: The CPU accounting controller groups tasks by using the cgroups and account for the CPU usage of the tasks. That means the cpuacct subsystem generates the CPU utilization report.

**Cpusets**: Cpusets provide a mechanism for assigning a set of CPUs and Memory Nodes to a set of tasks. Cgroups uses the cpusets subsystem to assign CPUs on multicore systems to various tasks in a group.

**Devices**: Control groups also manage the devices, granting or denying access to tasks within a group.

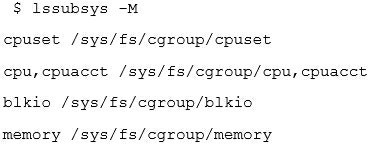
There are various ways to control work when working with cgroups . The most common way is to access the cgroups via the virtual cgroup filesystem ( cgroupfs ) or by accessing it using the libcgroup library packages.

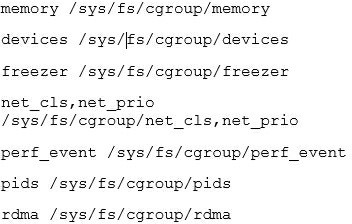
View the cgroups in the /sys/fs/cgroup directory .

You can install the libcgroup package on Debian using the following command:

sudo apt-get update && sudo apt-get install cgroup-tools -y

Once you have the packages installed, you can view the mounted subsystems and their mount points in the pseudo filesystem using the following command:





Let’s discuss the union filesystem.

## The Union Filesystem

To gain a better understanding of the composition of Docker containers, we will have to understand the Union filesystem first, which, although steeped in technicalities, is fundamental to understand.

Most GNU Linux and Unix users are aware that, except for individual regular files, almost everything in the system appears as a file. System devices in the file system also appear as files under /dev/device\_name .

A formatted partition —meaning it has a filesystem such as NFTS, FAT, FAT32, EXT fat, etc.— has to mounted on the system so that you can interact with it. The process of mounting involves attaching the file system on a logical file to a specific directory within the system tree, which is where the Union File system comes into play. The UNION file system concept comes from sets and sets theory in mathematics.

You can read more about sets and sets theory from the following resource: <https://en.wikipedia.org/wiki/Union_(set_theory)>

Let take an example where you mount two file systems on a single mount point of a system.

Without using the UNION file system, you will only see the files of the mount point you mounted last. The UNION file system allows us to view all the contents of mounted file systems on a single mount point.

The UNION filesystem works by mounting the files and directories of various file systems (also known as layers) and creating a new virtual file system.

Docker uses the UNION mounting system by overlaying all the layers attached to a specific image and initializing a read-only filesystem while starting a container. Docker also creates a new read-only layer that the container runtime environment uses.

Docker uses various flavors of the Union filesystem. These include DeviceMapper, overlay1, overlay2, ZFS, AUFS, etc. Docker also creates a virtual file system storage driver that does not support copy-on-write.

It’s worth noting that the VFS is not a UNION filesystem. The VFS driver means that the layer within the file system is a directory, and the new layer created requires a deep copy of the existing parent layer. That often leads to

low performance and increased disk usage—however, its stable and works in most environments.

## Container Format

The Docker Engine encapsulates the UnionFS, control groups and namespaces into a single wrapper referred to as a container format.

libcontainer is the default container format.

Now that you know more about the Docker platform and its base architecture, let’s move on and learn how to install Docker on various systems:

# Section 2: Installing Docker On Windows, Linux (Debian-Based), And Osx

In this section, we are going to cover how to set up Docker on various platforms.

We are going to use Debian Buster for Linux illustrations, Windows 10 for Windows Illustration, and the Google Cloud Platform for cloud-based Docker instances. Docker has support on most Linux distributions, including REHL flavors, ARCH, Gentoo, Ubuntu, Fedora, and many others.

Let us first see the requirements for setting up Docker on our local machines.

## System Requirements

For Windows users, the following are the recommended requirements:

Windows 10 build 1506 or higher A 64-bit, Intel or AMD processor

At least 4 GB of memory. For smoother performance, it’s better to have 8 GB of memory or and higher.

Hardware-level virtualization enabled. You can use the system BIOS to enable VT-X on your local machine.

If you are using Linux, you can check the system requirements for your favorite distribution below.

NOTE: Debian users must have at least 4 GB of RAM and running Debian 10 Buster or Debian 9 Stretch. Your system should also be running on a 64-bit architecture since Docker does not support x-86 architecture.

**CentOS:** <https://docs.docker.com/engine/install/centos/> **Ubuntu and Ubuntu-based distributions:** <https://docs.docker.com/engine/install/ubuntu/> **Debian and Raspbian Distributions:** <https://docs.docker.com/engine/install/debian/>

**Fedora:**

<https://docs.docker.com/engine/install/fedora/>

To check the architecture of your current system. Use the command:



The system should also be running kernel version 3.8 or higher. Check the kernel version using the command:

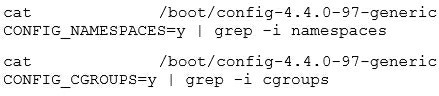


You can also check for supported storage backends such as DeviceMapper, VFS, AUFS, ZFS, and Overlay filesystem. Systems such as Ubuntu may use Overlay FS.

Most Linux distributions should have device-mapper thin-provisioning module for implementing the layers. To check whether you have device-mapper installed on your distro, use the command.

dmsetup -ls

Finally, you should ensure you’ve enabled support for namespaces and cgroups . Since most Linux distributions have made them available and supported for a while, your Linux distro should have that support in-built. To check cgroups and namespaces, check the kernel configuration file using the command:



For smooth experiences, Mac users should have the following system

requirements.

The Mac hardware must be a 2010 or newer model macOS must be version 10.13 or later.

At least 4 GB of RAM.

You should have VirtualBox versions 4.3.30 or later installed on your system; earlier versions are not compatible with Docker Desktop

## How To Install Docker Desktop On Windows

We are going to install Docker on our local Windows OS-based machine using the Docker desktop installer. The installer has the Docker engine, the Docker client, Docker Compose, Kubernetes, and Credential Helper.

**NOTE:** Docker containers created using the Docker desktop are sharable with all the user accounts available on the installation host. That’s because Windows user accounts use the same VM to run the containers.

First, open your browser and navigate to: <https://www.docker.com/products/docker-desktop>

Once on the landing page, download the docker desktop installer for windows.



As a recommendation, enable Hyper-V features before the installation. If not enabled, you can allow the Docker installer to enable it for you.

Once downloaded, start the installer and allow it to download any required packages.

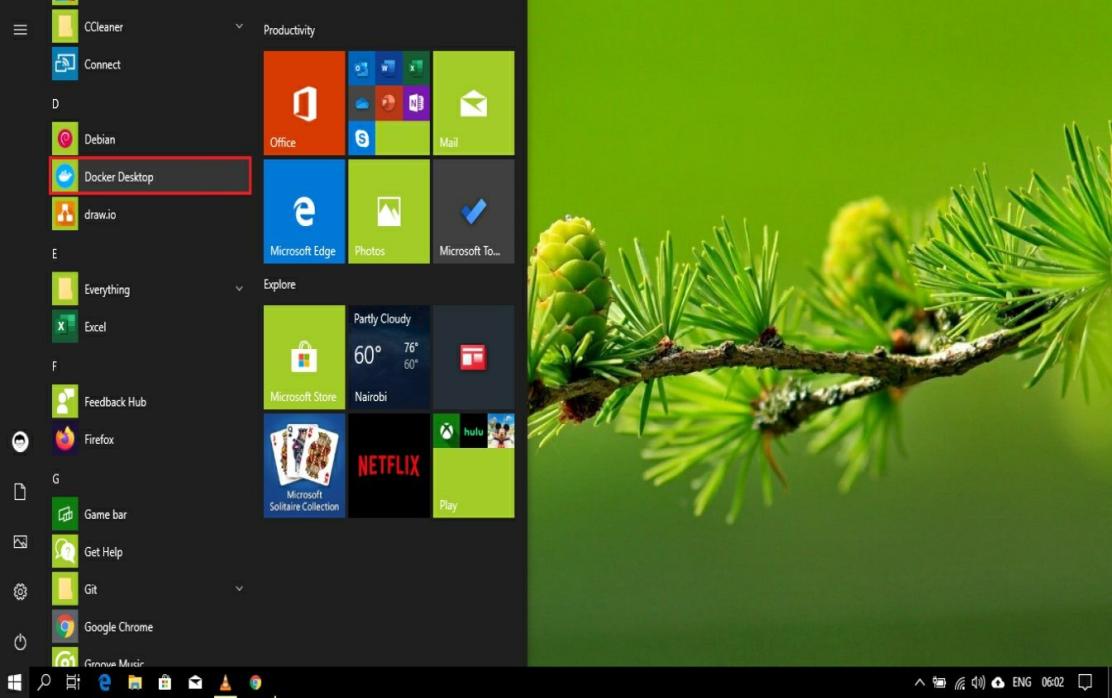
During the installation, select “Enable Hyper-V Windows features.” Selecting

this feature will require a reboot after installation.

Once the installation completes, click close, if required, reboot your computer, and start Docker.

## How To Run Docker Desktop

Once the installation completes and you have everything set up, click on the Docker desktop icon on the desktop or open from the start menu to launch Docker.



**NOTE:** For initialization, ensure you have at least 2 GB of free memory to avoid any errors.

Once successfully initialized, you will get a Docker welcome screen that has a tutorial on how to build your first Docker image. We will cover this in later sections of the book, but feel free to experiment.

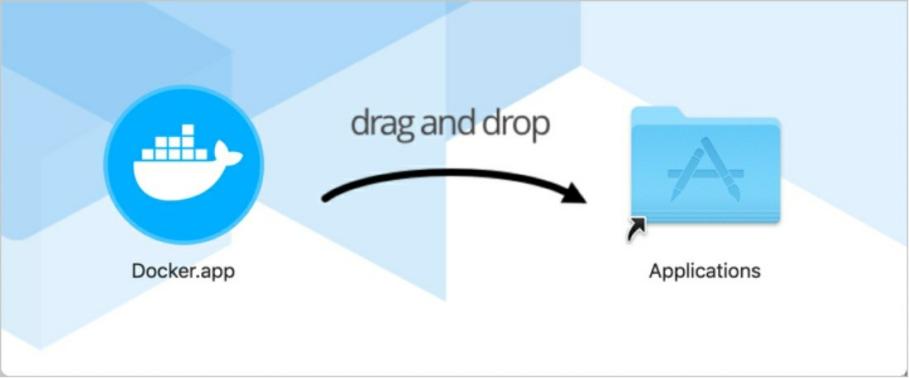
## How To Install Docker On Mac OS

The Docker desktop installer for Mac also comes coupled with the Docker engine, Docker client, and all the packages included in the Windows installer.

Open your browser and navigate to the following URL: <https://hub.docker.com/editions/community/docker-ce-desktop-mac> Once on the landing page, download the docker installer for mac.



Once downloaded, open the docker.dmg file to open the installer and drag the docker icon to the Applications folder on your system.



To start Docker, click on the docker icon in the Applications folder and wait for initialization. Once Docker is running, you will get a welcome window with a starter tutorial.

## Installing Docker On Linux (Debian Buster)

Before installing Docker on Linux, ensure you are using a system that meets the requirements listed earlier.

Next, we need to remove all previous installations of Docker on the system. The names of docker installations might include: docker.io docker-engine, Docker, containerd, runc

Open the terminal and enter the command below:

sudo apt-get remove docker.io Docker, docker-engine –y

We can now install Docker without the probability of running into problems caused by previous installations.

In the terminal, start by executing the command:



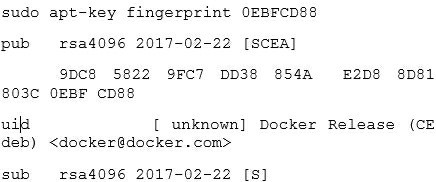
Next, we add the docker GPG key using the command:



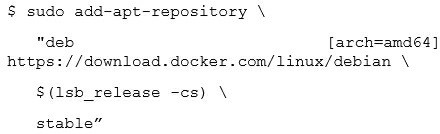
Next, verify that you are using the Docker official key with fingerprint:



Use the command below to search the last characters of the key and verify:



The next step is to add the docker apt repository to the stable repository channel. Use the commands below:



If you prefer frequent updates, you can use the nightly test channel by changing the stable value to test in the above command.

**NOTE:** Nightly channel may have a few bugs.

Now we can install Docker engine using the command below:



Once the installation completes, verify that the installation is working using the command:

docker container run hello-world

You can also configure the docker daemon to start during boot time using the systemctl

Start the docker service using the command sudo systemctl start docker

Enable the docker service at startup using the command: sudo systemctl enable Docker

Stop the service with sudo systemctl stop Docker.

## How To Use A Script To Automate Docker Install

In most cases, you will need to setup docker on a single host, which is a relatively simple process. However, if you need to setup Docker on hundreds of hosts, the task will be repetitive and tedious. You can use a script to automate this process.

Open the command prompt and start a new bash file. Ensure you are comfortable with what the script does before executing it. You will require sudo or root permissions.

Navigate to the following URL:

[https://get.docker.com](https://get.docker.com/)

Once there, copy the script and save it. Once done as saved, execute the file using the command:



# Section 3: How to Pull Docker Images and Run Containers

This section intends to test whether Docker is running as expected, not to explain the concepts. We will cover the entire Docker workflow in later sections.

We will start by pulling a docker image and running a container using the image. If you prefer to using the graphical interface provided by Docker desktop, we will be using the commands throughout the book. To avoid potential errors, ensure that Docker daemon is running. Open the terminal and enter the command:



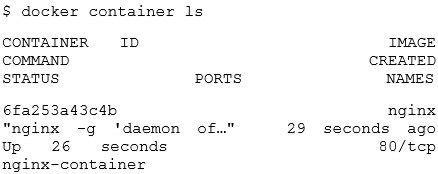
Once you have the image downloaded, you can view the list of images using the command:



For example, to create an Nginx container, we use the command:



To view the containers created, use the command:



$ docker container ls

Docker uses a client-server architecture. The Docker binary has the Docker client and the Docker server daemon that exist in a single host.

The Docker client can communicate with a local or remote docker server daemon via the network sockets or RESTful API clients. The docker server daemon is responsible for performing tasks such as building, running, and distributing containers.

The docker client daemon sends commands to the docker server daemon that is running on a remote or localhost, which then connects to the Docker registry to get the images requested by the docker client.

In our simple example above, the docker client installed with the docker binary communicates with the docker server daemon that then connects to the docker registry requesting an NGINX image. Once downloaded or found, we can use it to create containers.

A docker image refers to a read-only template used to create containers during runtime. Docker image templates depend on the base image and all the layers residing in it.

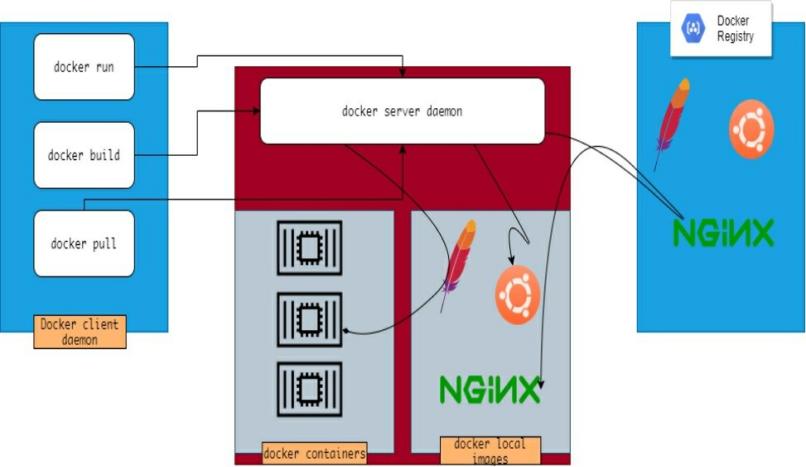
A docker registry stores the docker images; the by docker daemon references it when pulling images. Docker registries can be public or private; it all depends on the specified setting and the location of pulling and pushing the images. Public docker images are available in the Docker hub.

An image repository refers to a collection of a similar set of images distinguished by their GUIDs. For example, you can install various versions of nginx image by passing the tag such as docker image pull nginx:latest where latest becomes substituted with the correct version.

Containers refer to “virtual machines” that run base containers and the accompanying layers. Containers have all the requirements for running applications on them.

A Docker registry index manages accounts, searches, tags, permissions, etc. in a public docker image registry.

The concept is illustratable using the following image:



Let’s move on to the next section and learn how to work with containers: