# Journey To Docker Mastery

Figure 1: Docker Logo

I confess that the first time I started using docker, I felt confused. I can use the containers of others. I can probably fumble through making my own container. I might be able to make my own environment of containers with a lot of pain and leveraging from others works. But I got to a point where **I REALLY WANTED TO UNDERSTAND DOCKER!** I want to understand it deeply now and become very effective with it. Thank God a professional need arose that warranted me taking the time to learn it better.

## "Docker Tutorial For Beginners" on YouTube by Programming with Mosh

I did an initial search, and found a course on YouTube given by **Programming with Mosh** called Docker Tutorial for Beginners. If you are seeking to build your own set of references for learning and reviewing Docker, I highly recommend Mosh's Docker course.

## What Is Docker?

A platform for building, running, and shipping applications so that IF, they run correctly on your machine, they will run correctly on someone else's machine.

Why will it NOT sometimes work on other machines WHEN you do NOT use Docker? Here are some common example reasons. * One or more necessary files aren't included on the new machine that weren't transferred. * There is a software version mismatch. * There is a configuration mismatch on the new machine.

Why will your application most likely work when you use Docker containers and environments? We can put all the packages and configurations needed in our containers and environment of containers. Then we ship that environment of containers. It's as though we have a large ocean going ship with containers on it. We load our interactive containers (application) on the ship (an environment) and deliver the ship anywhere. What does the new host machine need to run your application? It only needs to be running Docker too.

I like to think of Docker as being *LIKE*, but more extensive than, a Python virtual environment. In a Python virtual environment, you can control the versions of your packages in isolation from other Python environments. Now, imagine extending that type of lite weight isolation to programs other than Python. And even more, imagine that this "container" has it's own file system. Now that's extensive isolation. That's like a steel shipping container. And like a steel shipping container, it's reusable and shippable to anywhere.

I also like to think of Docker containers as being *LIKE* a virtual machine. They provides the isolation level of a virtual machines but use much fewer resources. They use the host OS and other host machine resources, but, again, each container has its own file system. The containers can interact with each other too to form a complete application of interactive containers. These interactive containers are called environments. When someone that wants to use our applications, they load our Docker environments with our interactive Docker containers. They don't have to spend a ton of time setting things up on their machine to get our shared applications to run.

## Installing Docker

You can sometimes hit issues installing docker, and I certainly did this time. I chose the install from repo method each time. After some searching, I settled on the 2nd method for a repo based install on this page - 3 Ways to Install Docker Engine on Linux Mint.

Why did I not put those commands in here? They could change from the time of this writing to the time you need them. For your sake, do a fresh Google / Web search.

## Completely Removing Docker From Your System

If you are like me, when learning or relearning or reviewing, you become uncomfortable with your memory regarding the way that you installed things the first time you were learning a new tool, OR you need some massive upgrades, and you just want to start from scratch to understand the latest install and setup practices. Well, that happened to me this time, so I wanted to add a section for such work here too.

First, IF and ONLY IF, you want to remove all old containers, running and not running, use the following.

```
sudo docker system prune -a
```

If you only want to remove certain containers, please study How To Remove Docker Images, Containers, and Volumes.

To remove docker from your system, start with the top of Install Docker Engine on Ubuntu.

From there, I learned to use the following to remove all old instances of Docker.

```
sudo apt-get remove docker docker-engine docker.io containerd runc
```

As an option for doing an extremely complete removal, go to the very bottom of that same link HERE.

Per that section, I also hunted for other things to clean up. I found a `.docker` directory in my Linux home directory, so I deleted that. I also assured myself that there were no docker related entries in my `.bashrc` file that lives in our Linux home directories.

## Installing Docker On Your System

The prerequisites needed are the following.

We need to install: * ca-certificates * curl * gnupg * lsb-release

These installs help us to prepare for installing Docker and Docker Compose from the Docker repositories.

We can do all of these installs using:

```
sudo apt-get install ca-certificates &&
sudo apt-get install curl &&
sudo apt-get install gnupg &&
sudo apt-get install lsb-release
```

The `&&`'s tell the Linux command terminal to run the next command ONLY IF the previous command(s) was/were successful.

### Creating A Secure Pathway Between Your Machine And The Docker Official Repositories

This next group of commands is quite involved. Let's take an initial look at them and then go through them piece by piece.

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | \
    sudo gpg --dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg &&
echo "deb [arch=$(dpkg --print-architecture) \
    signed-by=/usr/share/keyrings/docker-archive-keyring.gpg] \
    https://download.docker.com/linux/ubuntu focal stable" | \
    sudo tee /etc/apt/sources.list.d/docker.list > /dev/null &&
sudo apt-get update
```

There are many options for `curl`. Let's look them up with `curl --help`.

- f is `-f, --fail           Fail silently (no output at all) on HTTP errors,`
- s is `-s, --silent         Silent mode`
- S is `-S, --show-error     Show error even when -s is used`
- L is `-L, --location       Follow redirects`

In light of our curl command to get data from the download docker URL for Linux Ubuntu, we want to: * fail silently on any HTTP errors and have silence about the processing, * BUT we do want to know about any errors during this download, * and we want `curl` to allow any redirects in case the maintainers of the download site created some redirects for downloads.

Wow! The `curl` routine can do a lot. Peruse the other options shown using `curl --help` in a Linux command terminal.

In case you are not familiar, the `|` symbol is terminal command short hand for "take the output from the command to the left, and send it, or 'pipe' it, as input to the command on the right.

We need to be super user, `sudo`, for the step to the right of the pipe - `|`. As super user, gpg. Wait, what does gpg do? First, it stands for GNU Privacy Guard. Second, it allows secure information transmission between parties and can verify that the message origin is genuine.

OK, but what does the `--dearmor` do? Well, that was a bit harder to track down, because it does not show up when running `gpg --help`. First, the `--armor` option "create[s] ascii armored output". Thus, and second, the `--dearmor` option "unwrap[s] a file already in PGP ASCII armor". I learnt this from GnuPG Hacks on the Linux Journal site.

Wow, we are making progress through this long command thanks to web searching help. Now, let's understand the `-o` option for `gpg` - `-o, --output FILE            write output to FILE`. Oh nice! That helps the next part make sense. We are sending the output from the command before the pipe symbol, `|`, thru `gpg` before we write it to a file in our Linux files system.

In summary, we are securely obtaining some important `keyring` information that we need using curl that will enable us to interact with the official docker repositories, and that information is wrapped in **armor**, which we need to remove BEFORE we write it to our file system. And that file name will be named `docker-archive-keyring.gpg` and will be located in `/usr/share/keyrings/`. This **keyring** is important to allow us to securely access the docker repositories. Phew! That was a lot - now we know. BUT WAIT! There's more!

### Storing The Road To, And The Procedures For, And The Keys To Get Into, The Docker Repositories

Now we get to walk through the next command. The next command simply stores the location of the Docker repository that we plan to use. It ensures that it becomes part of our system's known set of repositories. This way, we will be informed of updates to the Docker applications, so that we can take advantage of them.

```
echo "deb [arch=$(dpkg --print-architecture) \
    signed-by=/usr/share/keyrings/docker-archive-keyring.gpg] \
    https://download.docker.com/linux/ubuntu focal stable" | \
    sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
```

I know that looks daunting, but I assure you that it's not. Let's break it down into its component parts.

The `echo` terminal command does what it says. It echos the content following it that is in quotes. If you've never played with it, I encourage you to do so.

Let's start at a high level and then go back to the details. All the stuff in the quotes is ___echo___ed as input THROUGH the pipe, `|`. The information passing THROUGH the pipe becomes input to the `tee` command that we run as superuser - `sudo`. The `tee` command copies standard input, that is coming THROUGH the pipe, to a file. We've given the file name and location that `tee` should write to. The `/etc/apt/sources.list.d/docker.list` is the directory where Ubuntu based flavors of Linux likes to store repository list files to.

What does the `> /dev/null` part do? The " file is the black hole file for Linux distributions. WHAT?!?! It's just a convenience file for throwing stuff into a digital black hole. It's best if I show you.

Notice that if echo stuff through a pipe to a file with the tee command, the echo that stuff still displays as output in the terminal. We simply want to avoid producing this output to the terminal. How do we do that?

```
thom@thom-PT5610:~/Tests$ echo "This is some stuff" | tee ./dev_null_test.txt
This is some stuff
thom@thom-PT5610:~/Tests$
```

We send the echo output to `/dev/null`. Now we get no output from our echo to the terminal.

```
thom@thom-PT5610:~/Tests$ echo "This is some stuff" | tee ./dev_null_test.txt > /dev/null
thom@thom-PT5610:~/Tests$
```

That was a lot to explain a very simple Linux command line principle. However, now it's ours. We can use it moving forward when convenient.

What does the `$(dpkg --print-architecture)` part do for us? It automates the provision of information. Run the `dpkg --print-architecture` part by itself in a terminal on the command line. It tells you what your system architecture type is. For my system ...

```
thom@thom-PT5610:~/Tests$ dpkg --print-architecture
amd64
thom@thom-PT5610:~/Tests$
```

That's cool - right? Consider this next example of a command line that was meant to automate even more information collection.

```
echo "deb [arch=$(dpkg --print-architecture) \
    signed-by=/usr/share/keyrings/docker-archive-keyring.gpg] \
    https://download.docker.com/linux/ubuntu \
    $(lsb_release -cs) stable" | \
    sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
```

The `$(lsb_release -cs)` automates filling in a specific release candidate for docker on your system. However, `https://download.docker.com/linux/ubuntu` might not have a release for your system. This happened to me as shown in the following command line run in my terminal.

```
thom@thom-PT5610:~/Tests$ lsb_release -cs
una
thom@thom-PT5610:~/Tests$
```

If you to go to The index of Linux Ubuntu dists for Docker, you will find that `una` isn't there. At least it wasn't there when I looked at the time of this writing.

Also, How To Install and Use Docker on Ubuntu 20.04 suggests ...

```
sudo add-apt-repository "deb [arch=amd64] \
    https://download.docker.com/linux/ubuntu focal stable"
```

That's why we used `focal stable`, which is there in The index of Linux Ubuntu dists.
In my experience, `focal` seems to frequently be a good ***general*** release candidate for many packages other than just Docker.

Once the above set of commands have been run successfully, we should first update our system with

```
sudo apt-get update
```

Watch while it runs. You should see it surveying the docker repository among other Docker repositories.

## Actually Installing Docker Applications

With all of the previous setup steps complete, we can **FINALLY** run the following command to install all that we need for Docker.

```
sudo apt-get install docker-ce docker-ce-cli containerd.io docker-compose-plugin
```

Here are descriptions of each of those packages:

| Package Name | Description |
|---|---|
| docker-ce | free and open-source Docker containerization platform |
| docker-ce-cli | provides the docker binary client that talks to the dockerd API, either on a local socket or remotely - you can build and run containers using only the API without this client |
| containerd.io | a daemon that manages the complete container lifecycle on a host system, from image transfer and storage to container execution and supervision to low-level storage to network attachments and beyond |
| docker-compose-plugin | a tool for defining and running multi-container Docker applications |

After all those installs, we need to start the Docker daemon. You can do so using the following command.

```
sudo dockerd
```

Sometimes, you can have some challenges starting the Docker daemon. The StackOverflow page Docker daemon start challenges will likely have one of the solutions that will help you.