[Where Are Docker Images & Containers Stored on the Host?](https://www.howtogeek.com/devops/where-are-docker-images-containers-stored-on-the-host/)

<https://www.howtogeek.com/devops/where-are-docker-images-containers-stored-on-the-host/#:~:text=If%20you%20want%20to%20access,~%2FLibrary%2FContainers%2Fcom.>

Docker uses two kinds of formats to represent running processes—images, and containers, and both store data on your computer’s drive. We’ll talk about the commands Docker provides for handling data, and how you can use them to access image and container files.

# The Difference Between Images and Containers

Images are what you create when you run docker build; they’re stored in a container registry like the Docker Hub, and contain all the files and code to run an app. You can think of them like ISO files for a virtual machine operating system.

Containers are created from images, and they’re like the actual virtual machine that runs the application. You might have multiple containers running in parallel off the same image. Each container will have its own file system, optionally created with “volume mounts” that bind data from the host to the container.

# Working With Docker Image Storage

Images store the entire contents of the image on your drive. Whenever you pull an image from the internet, it’s downloaded and stored, usually forever. Images can be very large, so this can add up over time, especially for laptops with limited storage.

If you want to access the image data directly, it’s usually stored in the following locations:

* Linux: /var/lib/docker/
* Windows: C:\ProgramData\DockerDesktop
* macOS: ~/Library/Containers/com.docker.docker/Data/vms/0/

However, touching this data is likely a **bad idea**. Docker’s storage is complicated, and actually varies wildly depending on [what storage driver](https://github.com/moby/moby/blob/990a3e30fa66e7bd3df3c78c873c97c5b1310486/daemon/graphdriver/driver.go#L37-L43) it’s using. Linux now defaults to overlay2 on most distros, which isn’t even accessible for most end users. Messing with this can lead to data loss.

Instead, Docker provides managed commands to handle images. You can view all versions of downloaded images with a simple command:

docker image ls

Luckily, it isn’t as bad as it looks, since Docker images store versions incrementally. That means, whenever you download a new version, it only replaces the parts that were changed. If you frequently use the same image over and over, you probably won’t rack up too much storage cost.

However, if you use a lot of different images, you might have many images saved that aren’t even used anymore. To clean these up, Docker provides a built-in command to run garbage collection. This will prune all images that have no references, i.e. not tagged or not referenced by any container:

docker image prune

To prune all old images not used by existing containers, run it with the -a flag:

docker image prune -a

That covers the main use case, but there are a few more useful commands:

* inspect: displays info about a container version.
* save & load: saves and loads images to a tar archive.
* rm: removes an image directly.
* pull/push: updates from a remote registry.
* history: provides a changelog.

# Working With Docker Container Storage

You can view all info about a container with docker inspect, which shows the filesystem drivers and data, as well as all the existing mounts and volumes:

docker inspect containerID

Containers store data in two ways. First is the base filesystem, which is copied from the image and is unique to each container. Docker uses a “lower dir” and “upper dir,” which are separate layers that get merged into one hybrid filesystem. The lower dir stores the base image data, and the upper dir stores everything that was changed at runtime, such as log files. In either case, the storage of these depends on the filesystem driver Docker is configured to use.

Then, there are mounts, which bind directories from the host to the container, usually managed automatically with a Docker feature called [*volumes*](https://docs.docker.com/storage/volumes/#populate-a-volume-using-a-container). These are stored normally, and are accessible to end users. If you’re doing any work that requires you to modify data on running containers, you should probably be modifying a volume or bind mount.

# Accessing Volumes

Bind mounts can be accessed directly, and are a great choice if you want to store config that’s used for many containers, or store accessible data that persists across container restarts.

If you want to modify data stored in volumes, you can do so too. They’re stored in a standard format accessible from Linux:

/var/lib/docker/volumes/volumeID/\_data

You can get the volume ID and information with docker volume inspect.

Much like images, volumes can also become stale. You can remove them easily, but [backing up and transferring them is a trickier process](https://docs.docker.com/storage/volumes/#backup-restore-or-migrate-data-volumes).

docker volume prune

docker volume rm volumeID

# Modifying a Docker Container’s Filesystem

If you want to modify the container’s filesystem, much like images, this is a **bad idea.** In most cases, you should create a new version of the container with the updated changes, and deploy an update.

However, if you want to make some quick changes without stopping the container, the best way is to just pop open a bash shell inside the container and modify it through Docker. Doing so is very easy—run docker exec on the container, and pass “bash” as the command:

docker exec -it container bash

From here, you are free to use normal Linux commands. If you want to do this remotely, you can [install an SSH server in your container](https://www.howtogeek.com/devops/how-to-ssh-into-a-docker-container/), and bind port 22 to another port on the host.