




# Containers



Into the containers, images, docker  
and other shits...



# Nomenclature - Container

---

A container is a lightweight, standalone package that encapsulates application code and all of its dependencies so the application can run in any environment.

Containers are considered lightweight “virtualization” technology. They are created from a container image.

A container is a self contained execution environment that shares the kernel of the host system and which is (optionally) isolated from other containers in the system.

- Lightweight + Portable
- Fast
- Secure

# Nomenclature - Image

---

A container image is a read-only, static file that includes executable code, libraries, and binaries.

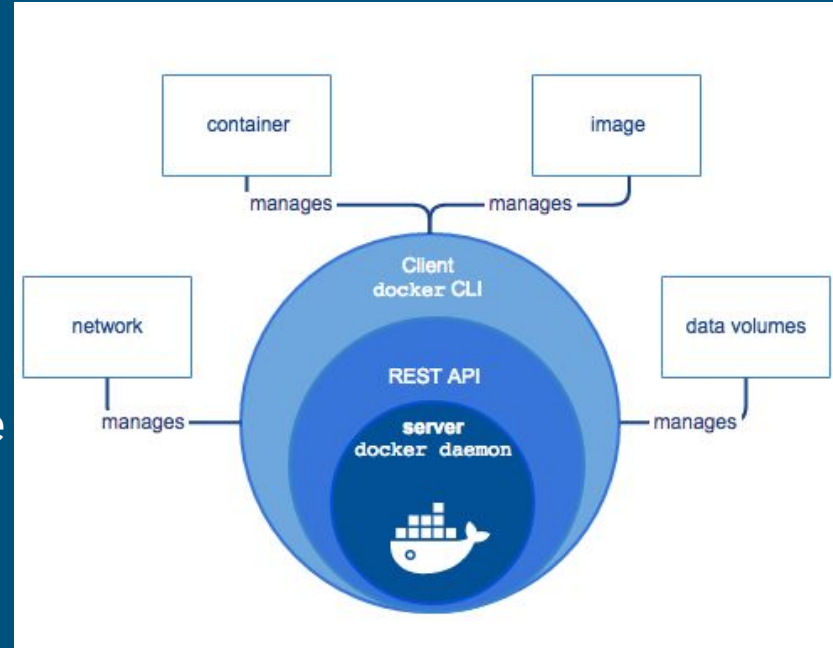
It can be roughly considered as similar to a template that is used to create something concrete that runs, which in our case is a container.

Image contains various layers as storage, which serves as a base for upper layers. The image also contains the configuration.

# Nomenclature - Docker Project

The docker container platform is a containerization platform developed by the company called Docker Inc.

What the Docker container platform does, is providing a “runtime environment” that is docker daemon, and the CLI to communicate with the daemon, in order to run containers and interact with them.



# The Container is a Linux Process

---

Containers are not the same as VMs, Docker Containers are simple Linux Processes with some additional configurations. It runs on the Host-OS. You can find the PID (Process ID) and PPID (Parent Process ID) using the following command.

# Namespace (OS Lies)

---

The Namespace is a feature of the Linux Kernel, Which limits what processes can access certain resources of the system.

The purpose of each namespace is to wrap a particular global system resource in an abstraction that makes it appear to the processes within the namespace that they have their own isolated instance of the global resource.

1. Mount (mnt, chroot)
2. Process ID (pid)
3. Network (net)
4. Interprocess Communication (ipc)
5. UTS (hostnames)
6. User ID (user)
7. Control group (cgroup)
8. Time

Processes are in hierarchy, So lying to parent will gets to childs.

# Cgroup

---

A control group (cgroup) is a Linux kernel feature that limits, accounts for, and isolates the resource usage (CPU, memory, disk I/O, network, and so on) of a collection of processes.

- Resource limits – You can configure a cgroup to limit how much of a particular resource (memory or CPU, for example) a process can use.
- Prioritization – You can control how much of a resource (CPU, disk, or network) a process can use compared to processes in another cgroup when there is resource contention.
- Accounting – Resource limits are monitored and reported at the cgroup level.
- Control – You can change the status (frozen, stopped, or restarted) of all processes in a cgroup with a single command.

# Cgroup (Contd.)

---

By default all the processes will be using common namespace and group. Because of that those are not isolated and they have wide access. But when you create a container, The container engine (containerd) will group processes using a Linux feature called **cgroup**. Then create separate instances of above namespaces and move the processes into the newly created instances of namespaces. So now the container has a dedicated namespace instance and cgroup which gives an impression it's a separate OS.

So containers are using these namespaces to create a fake isolated environment with shared and unshared resources but using the same kernel inside and outside the container.

Namespaces are created using syscalls. (sethostname, setdomainname, uname). Docker is a fancy interface of this shared and unshared namespace feature. You can even make containers without docker using these Linux features.



# No to windows/mac, All my homies use linux

---

Docker Toolbox ( For windows 7,8 and older macs) — DEPRECATED: In the older versions of windows (7,8) and Mac, Docker used a package called Docker Toolbox. It's a bundle of some necessary toolkits and Virtual Box to create a Linux VM to run docker. This feature is deprecated now!

Hyper-V: In windows 10 pro, Microsoft introduced a virtualization technology called Hyper-V. Instead of using VirtualBox, Docker migrated to Hyper-V and LinuxKit to create a lightweight Linux VM for docker operations.

WSL2 — Windows Subsystem for Linux: WSL is a Subsystem for Linux, Which allows developers to execute all Linux apps (CLI) and commands without any Virtual Machines. It just an emulation. But It had some foundation issues, So docker did not use WSL. But in the updated version (WSL2) Microsoft did some major changes. Instead of emulating, it provides a real Linux Kernel which is running inside a lightweight VM. So now docker uses WSL2 for the Linux Kernel instead of Hyper-V and LinuxKit.

MacOS: Just like windows, Docker for Mac is using a lightweight Linux VM with the support of Hypervisor and HyperKit.

# Do Yourself

---

<https://medium.com/@ssttehrani/containers-from-scratch-with-golang-5276576f9909>

<https://www.infoq.com/articles/build-a-container-golang/>

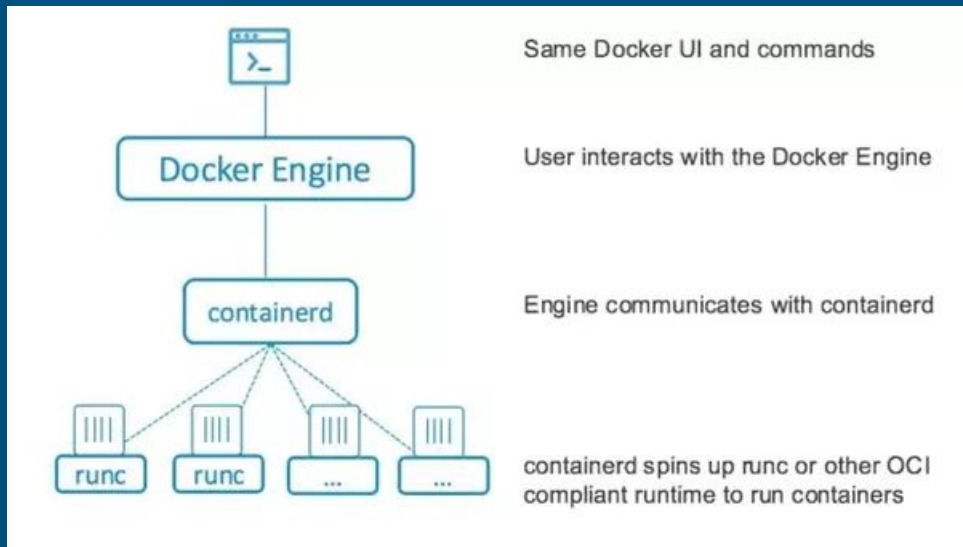
# Docker Daemon

As of 1.11, the execution of containers is handled by a container runtime which is called **containerd**. More precisely, the Docker daemon prepares the image according to OCI image spec, creates a “**filesystem bundle**” that is unpacked on the disk, and makes an API call to containerd, to start the **bundle**.

Prior to Docker version 1.11, the Docker Engine daemon downloaded container images, launched container processes, exposed a remote API, and acted as a log collection daemon, all in a centralized process running as root.

While such a centralized architecture is convenient for deployment, it does not follow best practices for Unix process and privilege separation; further, it makes Docker difficult to properly integrate with Linux init systems such as upstart and systemd.

Since version 1.11, the Docker daemon no longer handles the execution of containers itself.



# Image consists of multiple layers

---

Most container image builds are defined through a Dockerfile, and the Dockerfile gives a series of instructions, each resulting in a filesystem layer (not precisely; continue reading).

Docker CLI has an inspect command to provide low-level information on docker objects. Let's have a look:

```
$ docker inspect alpine:latest
```

```
"RootFS": {  
  "Type": "layers",  
  "Layers": [  
    "sha256:3e207b409db364b595ba862cdc12be96dcdad8e36c59a03b7b3b61c946a5741a"  
  ]  
},
```

# Open Container Initiative

---

The Open Container Initiative (OCI) was formed, to define standards around container images and runtime. The OCI specs cover an image format, which discusses how container images are built and distributed.

It took its lead from a lot of the work that had been done in Docker, so there is quite a lot in common between what happens in Docker and what is defined in the specs — in particular, a goal of the OCI was for the standards to support the same user experience that Docker users had come to expect, like the ability to run an image with a default set of configuration settings.

<https://github.com/opencontainers/image-spec>

<https://github.com/opencontainers/runtime-spec/blob/main/bundle.md>

# Run container manually from image

---

```
$ sudo ls alpine-bundle/  
config.json rootfs  
sha256_219c332183ec3800bdfda43b3b27f7f516b8683b48352190f4d14ce25be00eed.mtree
```

```
$ sudo ls alpine-bundle/rootfs  
bin dev etc home lib media mnt opt proc root run sbin srv sys tmp usr var
```

```
$ sudo runc run container1  
/ # ls  
bin      dev      etc      home     lib      media    mnt      opt      proc     root     run      sbin     srv      sys  
tmp      usr      var  
/ #
```



# History

---

Initially, Docker daemon in the beginning has a monolithic architecture that managed image build, container run times, docker client, etc. Also, Docker was built on top of LXC (Linux Container). LXC is a long time ago developed way to create lightweight Linux container. In the first its releases, Docker based its containers using the LXC technology to build environments for single applications.

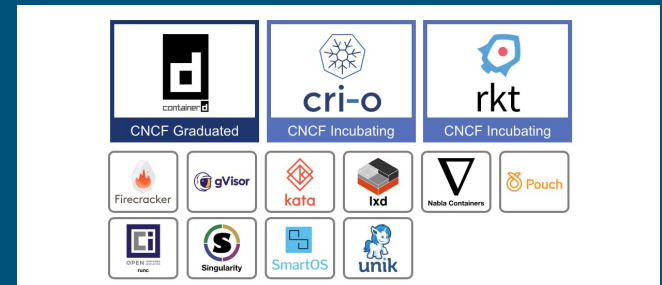
After a while, the Docker's architecture was changed. It moved from the monolithic architecture to a modular one that enabled them to innovate faster. It also got rid of LXC, which was Linux specific and did not allow them to work on Windows or Mac or 32 bit systems. Currently Docker uses it's own alternative to LXC called libcontainer.

**Libcontainer** provides a native Go implementation for creating containers with namespaces, cgroups, capabilities, and filesystem access controls. It allows you to manage the lifecycle of the container performing additional operations after the container is created.

<https://github.com/opencontainers/runc/tree/main/libcontainer>



# Containerd



The containerd component makes sure that the container image you want to run is in place, then starts the container by calling a component named **runc** to do the business of actually instantiating a container. In fact, if you want to, you can run a container yourself by invoking containerd or even runc directly without docker CLI.

Kubernetes uses an interface called the **Container Runtime Interface (CRI)** beneath which users can opt for a container runtime of their choice. But of course, the most commonly used option today is still containerd (perhaps CRI-O as well, which originated from Red Hat before being donated to the CNCF). The containerd later project was donated by Docker to the Cloud Native Computing Foundation (CNCF) back in 2017.

<https://kubernetes.io/docs/setup/production-environment/container-runtimes/>

**runc** is a lightweight CLI wrapper for the libcontainer and it is used to spawn and run containers. Basically, you can run your containers without Docker, by using the CLI only.