Docker - A Container

A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another.

#virtual-machine #docker #container

Last update: 2021-08-04 17:31:07

Table of Content

Docker gained so much popularity and adoption in the DevOps community in a short time because of the way it's developed for portability and designed for modern microservice architecture.

1. Linux container

The concept of containers started way back in the 2000s. In fact, the roots go back to 1979 where chroot was introduced - it's a concept of changing the root directory of a process.

In a typical virtualized environment, one or more virtual machines run on top of a physical server using a hypervisor like Xen, Hyper-V, etc.

Containers, on the other hand, run on top of operating systems kernel - so called OS-level virtualization.

A container is a Process

When a process is started, it runs in a self-contained virtual space. However, it still interact with external environment.

If a process is isolated with all of its files, configurations to make it run and operate, it needs to be in a container, and a container actually to that.

A container is basically a process with enough isolation of user-space components so that it gives a feeling of a separate operating system.
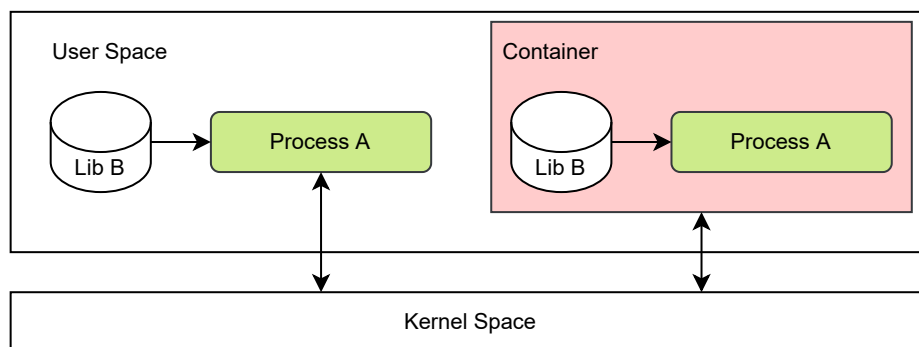

A process and A container

2. Docker

Docker evolved on Linux. A container is considered "native", if it can run directly on the host operating system. Therefore, Docker runs on Linux is a native docker container.

To run a container on Windows, Docker has to create a Linux Virtual Machine using virtualization to emulate a Linux environment. This virtualization can be:

- VirtualBox (Docker Toolbox)

- Hyper-V backend or WSL2 backend (Docker Desktop)
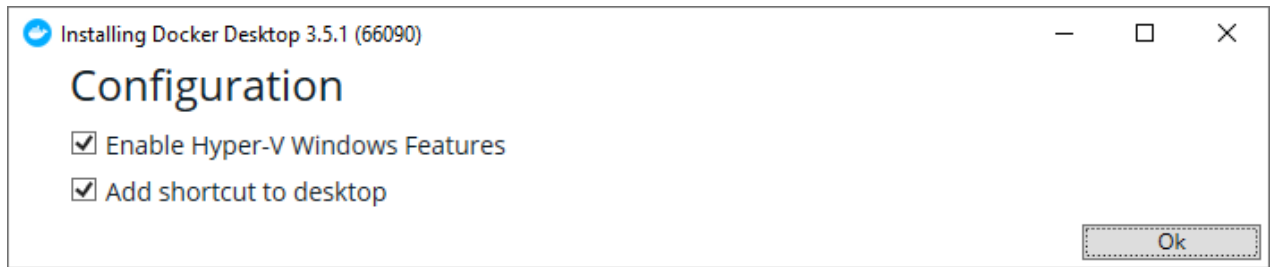
Install Docker on Windows

On Windows, there are Windows (Server) Containers: Windows applications that run in isolated Windows environment. Windows Hyper-V can be used to run even native Windows containers, which is generally a source of confusion:

- Process Isolation: This is the "traditional" isolation mode for containers. It is approximately the same as how Linux containers run on Linux

- Hyper-V isolation: This isolation mode offers enhanced security and broader compatibility between host and container versions.

3. How Docker works

When Docker starts at the first time, Docker will create a Virtual Machine which runs a Linux OS. Open the Hyper-V Manager to see the configuration of that VM.
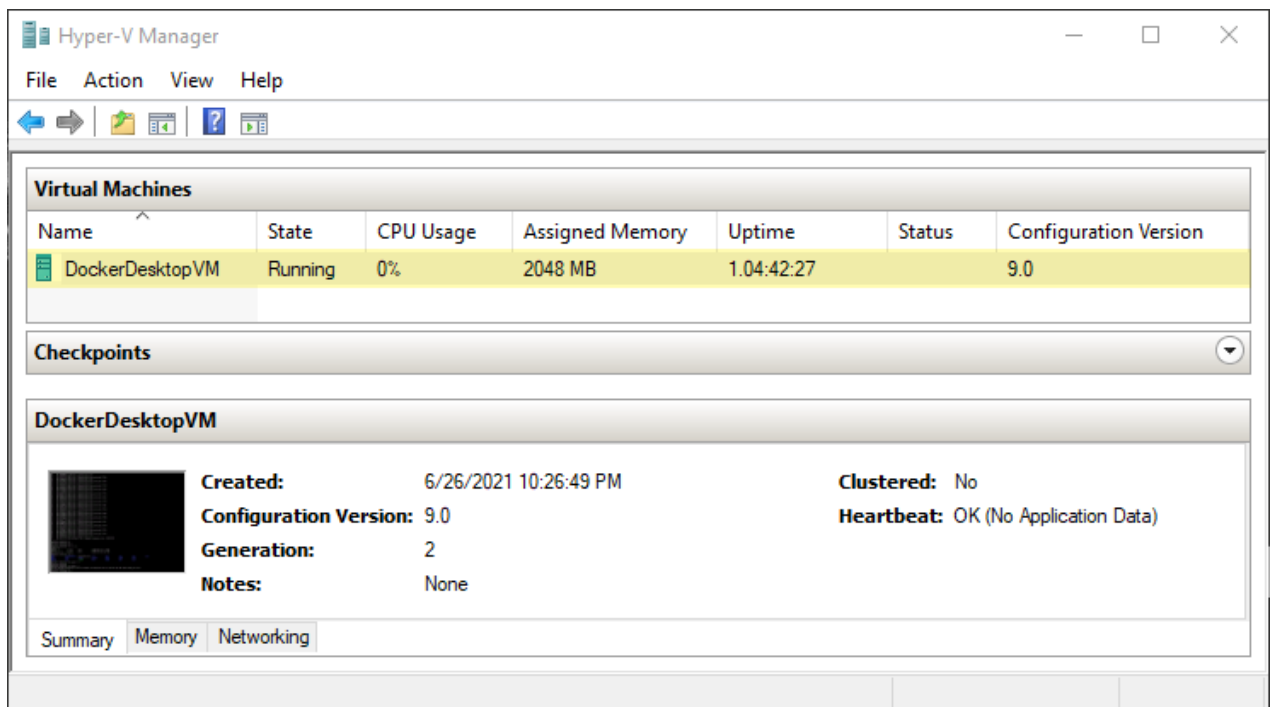

Docker Desktop Virtual Machine

This virtual machine is initialized from an ISO disk image located in `C:\Program Files\Docker\Docker\resources\docker-desktop.iso`.

Let's inspect the virtual machine by connecting it and access into its terminal:

```
uname -a
```

```
Linux 6a7ab2c7921c 5.10.25-linuxkit #1 SMP Tue Mar 23 09:27:39 UTC 2021 x86_64 x86_64
x86_64 GNU/Linux
```

It shows a Linux kernel version 5.10 in `linuxkit` repo! What is the `linuxkit` repo? It is a special kernel under development of LinuxKit team which provides a toolkit for building custom minimal, immutable Linux distributions. Read more in https://github.com/linuxkit/linuxkit.
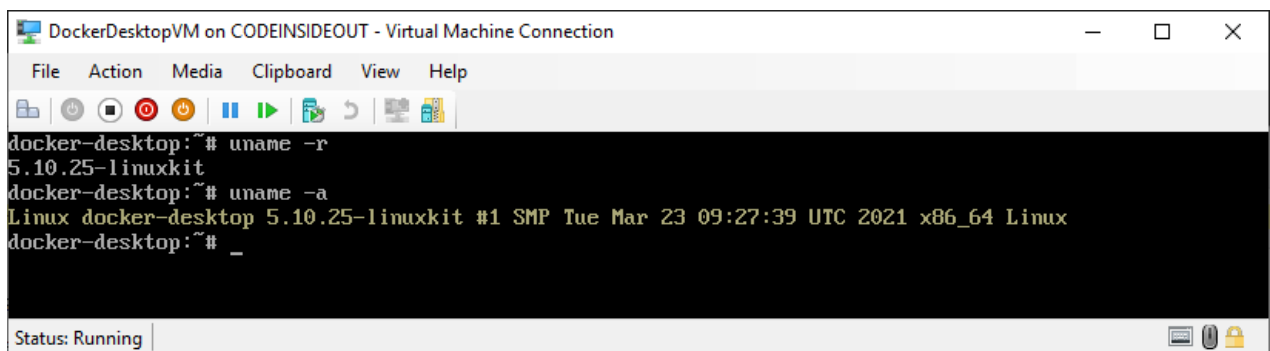
Docker Desktop Kernel

Docker takes advantage of several features of the Linux kernel to deliver its functionality.

Namespaces

Docker makes use of kernel `namespaces` to provide the isolated workspace called the `container`. When container runs, Docker creates a set of namespaces for that container. These namespaces provide a layer of isolation. Each aspect of a container runs in a separate namespace and its access is limited to that namespace.

Docker Engine uses the following namespaces on Linux:

- PID namespace for process isolation.
- NET namespace for managing network interfaces.
- IPC namespace for managing access to IPC resources.
- MNT namespace for managing filesystem mount points.
- UTS namespace for isolating kernel and version identifiers.

Cgroups

Docker also makes use of kernel `control groups` for resource allocation and isolation. A `cgroup` limits an application to a specific set of resources. Control groups allow Docker Engine to share available hardware resources to containers and optionally enforce limits and constraints.

Docker Engine uses the following cgroups:

- **Memory cgroup** for managing accounting, limits and notifications.
- **HugeTBL cgroup** for accounting usage of huge pages by process group.
- **CPU cgroup** for managing user / system CPU time and usage.
- **CPUSet cgroup** for binding a group to specific CPU. Useful for real time applications and NUMA systems with localized memory per CPU.
- **BlkIO cgroup** for measuring & limiting amount of `blckIO` by group.
- **net_cls and net_prio cgroup** for tagging the traffic control.
- **Devices cgroup** for reading / writing access devices.
- **Freezer cgroup** for freezing a group. Useful for cluster batch scheduling, process migration and debugging without affecting `prtrace`.

Union File Systems

Union file systems operate by creating layers, making them very lightweight and fast. Docker Engine uses UnionFS to provide the building blocks for containers. Docker Engine can use multiple UnionFS variants, including AUFS, btrfs, vfs, and devicemapper.

Container Format Docker Engine combines the namespaces, control groups and UnionFS into a wrapper called a container format. The default container format is `libcontainer`. Here are main points about container:
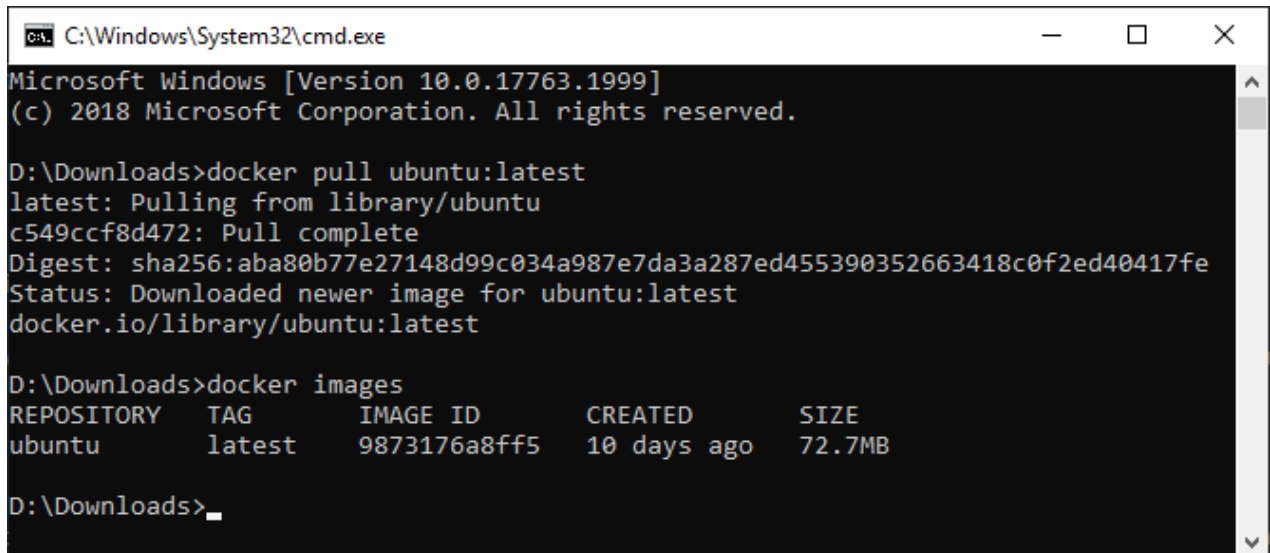
- Containers share the host kernel
- Containers use the kernel ability to group processes for resource control
- Containers ensure isolation through namespaces
- Containers feel like lightweight VMs (lower footprint, faster), but are not Virtual Machines!

When a container runs, it basically extracts the container image content then add its layers using UnionFS into the host kernel, in an isolated namespace, under a control group, and finally expose to user that isolated live running container.

3.1. Inspect an image

To get an image, pull it from Docker Hub. Let get a latest Ubuntu image from https://hub.docker.com/_/ubuntu using image name `ubuntu` and the tag name `latest` :

```
docker pull ubuntu:latest
```

```
C:\Windows\System32\cmd.exe                                    —    □    ✕

Microsoft Windows [Version 10.0.17763.1999]
(c) 2018 Microsoft Corporation. All rights reserved.

D:\Downloads>docker pull ubuntu:latest
latest: Pulling from library/ubuntu
c549ccf8d472: Pull complete
Digest: sha256:aba80b77e27148d99c034a987e7da3a287ed455390352663418c0f2ed40417fe
Status: Downloaded newer image for ubuntu:latest
docker.io/library/ubuntu:latest

D:\Downloads>docker images
REPOSITORY      TAG        IMAGE ID       CREATED       SIZE
ubuntu          latest     9873176a8ff5   10 days ago   72.7MB

D:\Downloads>_
```

Pull a latest Ubuntu image

To export the downloaded image to a file, use *save* command as below:

```
docker image save ubuntu > ubuntu.tar
```

Extract the `ubuntu.tar` file to get its content:

```
|   X.json
|   manifest.json
|   repositories
|
└───X
        json
        layer.tar
        VERSION
```

The `repositories` has the SHA id `X` of the image:

repositories

```
{
    "ubuntu": {
        "latest":"X"
    }
}
```

Then the `manifest.json` has information about the image configuration and its layers.

manifest.json

```
[
    {
        Config: "Y.json",
        RepoTags: ["ubuntu:latest"],
        Layers: ["X/layer.tar"],
    },
];
```

In the configure file, there are some more information that can be seen in clear text:

- Environment PATH
- Build Command
- Layers

```
{
  "architecture": "amd64",
  "config": {
    ...
    "Env": [
      "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
    ],
    "Cmd": ["bash"],
    "Image": "sha256:Z",
    ...
  },
  "container": "A",
  "container_config": {
    ...
    "Env": [
      "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
    ],
    "Cmd": ["/bin/sh", "-c", "#(nop) ", "CMD [\"bash\"]"],
    "Image": "sha256:Z",
    ...
  },
  "created": "2021-06-17T23:31:29.779641053Z",
  "docker_version": "19.03.12",
  "history": [
    {
      "created": "2021-06-17T23:31:29.39657203Z",
      "created_by": "/bin/sh -c #(nop) ADD file:B in / "
    },
    {
      "created": "2021-06-17T23:31:29.779641053Z",
      "created_by": "/bin/sh -c #(nop)  CMD [\"bash\"]",
      "empty_layer": true
    }
  ],
```

```
    "os": "linux",
    "rootfs": {
      "type": "layers",
      "diff_ids": [
        "sha256:C"
      ]
    }
  }
}
```

And zipped files `.tar` contain filesystem of each layers, which will be populated into the host kernel in an isolated namespaces.

File system in a layer

That's how container works.

4. Reference

- https://medium.com/@BeNitinAgarwal/understanding-the-docker-internals-7ccb052ce9fe
- http://docker-saigon.github.io/post/Docker-Internals/