

Docker Masters

Łukasz Lach



Łukasz Lach

Docker Captain

Docker Community Leader | Warsaw, Poland

Lecturer | University of Warsaw

Senior Software Architect | CD Projekt

llach@llach.pl | <https://lach.dev>



Training structure, requirements, used software and technologies



What are Docker, Docker Engine and Docker Client?



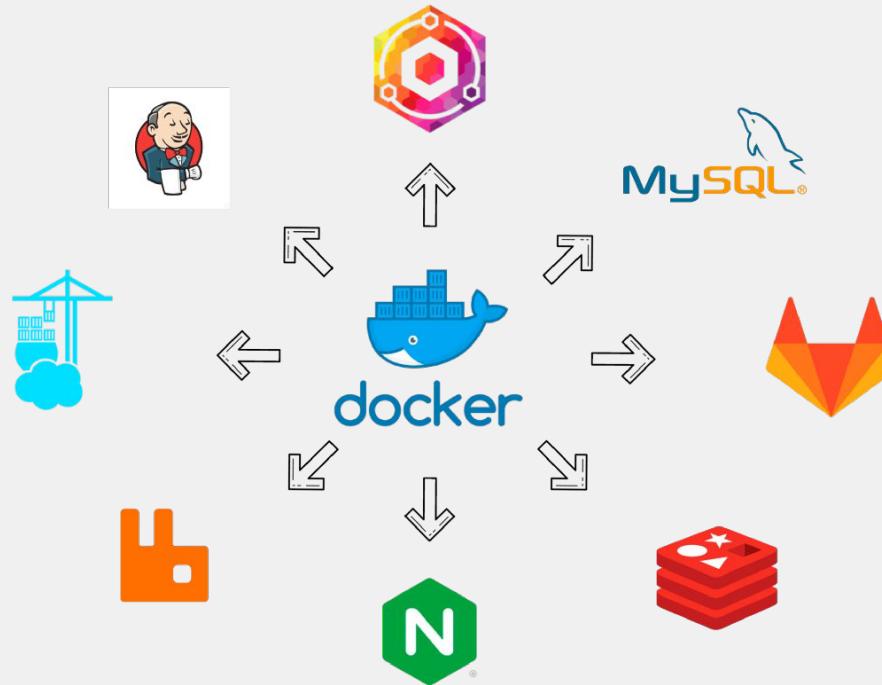
Docker

Docker provides the ability to package and run an application in a loosely isolated environment called a container. The isolation and security allow you to run many containers on a given host.

Containers are lightweight because they don't need the extra load of a hypervisor, but **run directly within the host machine's kernel**. There is **no virtualization** involved when working with containers.



Popular Docker images in 2020



Docker Client

The Docker client (`docker` command-line) is the primary way that many Docker users interact with Docker.

When you use commands such as `docker run`, the client sends these commands to `dockerd` using the Docker HTTP API.

Commands payload include locally stored registry credentials.

The Docker client can communicate with more than one daemon.



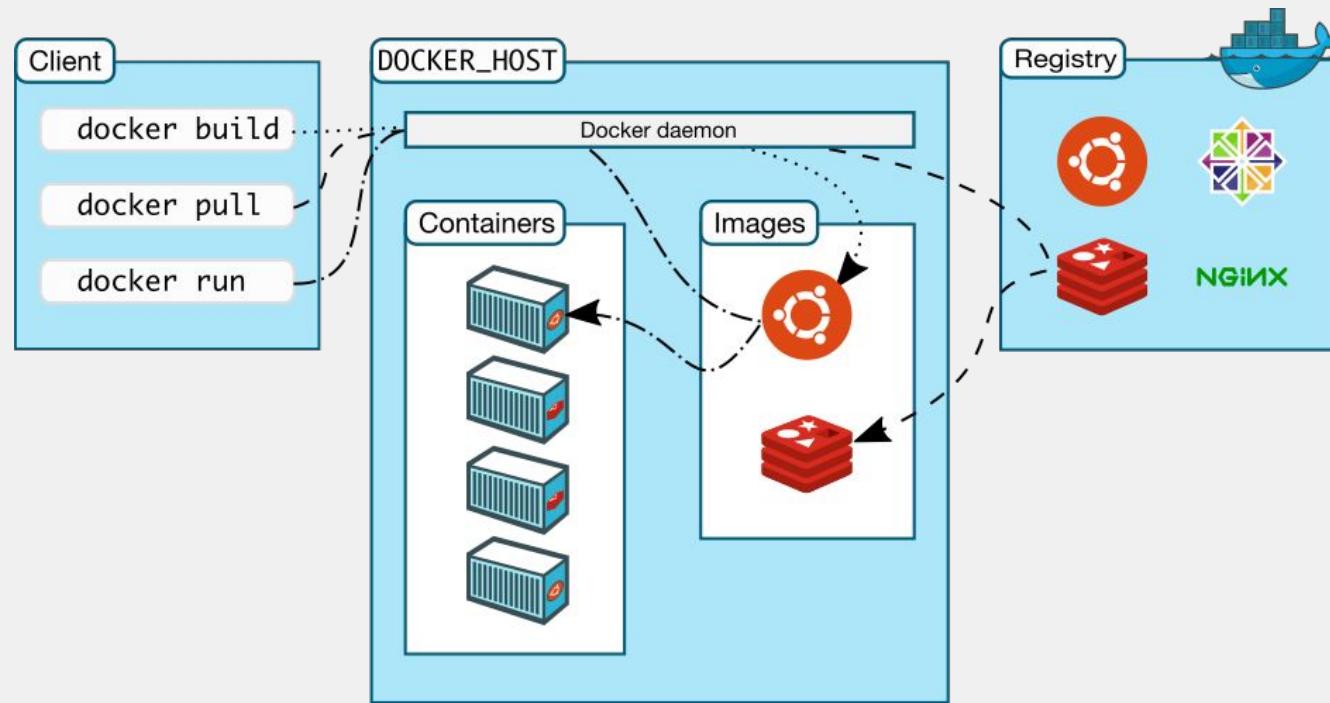
Docker Engine

Docker Engine is an open source containerization technology for building and containerizing your applications.

The CLI uses Docker APIs to control or interact with the Docker daemon through scripting or direct CLI commands. The daemon creates and manage Docker objects, such as images, containers, networks, and volumes.



Docker architecture



Trigger the Docker Client

Run `docker version` to ensure you have rights to use the Docker Client.

```
bash
$ docker version
Client: Docker Engine - Community
Version:          19.03.1
API version:      1.40
Go version:       go1.12.5
Git commit:       74b1e89
Built:            Thu Jul 25 21:18:17 2019
OS/Arch:          darwin/amd64
Experimental:     true
```



Check if the daemon is running

Run `docker info` to ensure connectivity with the Docker Engine.

```
● ● ● bash
$ docker info
Client:
  Debug Mode: false
  Plugins:
    app: Docker Application (Docker Inc., v0.8.0-beta1)
    buildx: Build with BuildKit (Docker Inc., v0.2.0-tp)

Server:
  Containers: 34
  Running: 2
  Paused: 0
  Stopped: 32
  Images: 128
```



Dump Docker Engine communication

Monitor the traffic between Docker client and Docker Engine.

With socat you can tunnel the packets through another unix socket.

```
bash
$ sudo socat -d -d -t100 \
-lf /dev/stdout \
-v UNIX-LISTEN:/var/run/docker.debug,mode=777,reuseaddr,fork \
UNIX-CONNECT:/var/run/docker.sock

2019/06/27 15:28:23 socat[83151] N listening on LEN=28 AF=1
"/var/run/docker.debug"
2019/06/27 15:28:43 socat[83151] N accepting connection from LEN=16
AF=1 "" on LEN=28 AF=1 "/var/run/docker.debug"
2019/06/27 15:28:43 socat[83161] N opening connection to LEN=22 AF=1
"/var/run/docker.sock"
```



Point the Docker Engine

With the DOCKER_HOST environment variable you can target the command to use specific unix socket or host and port.

Starting from 19.03 you can use the docker context command.



bash

```
$ DOCKER_HOST=unix:///var/run/docker.debug \
  docker ps
```





socat

```
> 2019/06/27 15:28:43.335395  length=101 from=86 to=186
GET /v1.40/containers/json HTTP/1.1\r
Host: docker\r
User-Agent: Docker-Client/19.03.0-rc2 (darwin)\r
\r

< 2019/06/27 15:28:43.352685  length=1138 from=283 to=1420
HTTP/1.1 200 OK\r
Api-Version: 1.40\r
Content-Length: 932\r
Content-Type: application/json\r
Date: Thu, 27 Jun 2019 13:28:43 GMT\r
Docker-Experimental: true\r
Ostype: linux\r
Server: Docker/19.03.0-rc2 (linux)\r
\r
[{"Id":"2743db7f7e2f4716dd794e9ef50ccaccfa9609218c8504bc220d602d808d690b","Names":["/nginx"],"Image":"nginx","ImageID":"sha256:719cd2e3ed04781b11ed372ec8d712fac66d5b60a6fb6190bf76b7d18cb50105","Command":"nginx -g 'daemon off;'","Created":1561641675,"Ports":[{"IP":"0.0.0.0","PrivatePort":80,"PublicPort":80,"Type":"tcp"}],"Labels":{"maintainer":"NGINX Docker Maintainers"}}
```



Send HTTP request to a Docker Engine

You can use any HTTP client to act as a Docker client.

For example using curl with a --unix-socket parameter.

```
bash
$ curl -sSf --unix-socket /var/run/docker.sock \
    0/containers/json
[
  {
    "Id": "535fd2f2d2021a0b5046922e5d38ba669e342a6897065c5f59d4120453",
    "Names": [
      "/nginx"
    ],
    "Image": "nginx",
    "ImageID": "sha256:719cd2e3ed04781b11ed372ecb6190bf76b7d18cb50105",
    "Created": "2021-07-20T14:45:21.000Z",
    "Status": "running",
    "Ports": [
      {
        "Protocol": "tcp",
        "Port": 80,
        "HostPort": 0
      }
    ],
    "Size": 1234567890
  }
]
```



Documentation

Docker documentation is available under <https://docs.docker.com/> but it can also be ran locally, with search support, in a Docker container. Learn and develop with Docker whether you have Internet connection or not.

```
$ git clone --recursive \
  https://github.com/docker/docker.github.io.git
$ cd docker.github.io/
$ docker-compose up
```



Docker 19 / docker ps — DevD x +

← → ⌂ devdocs.io/docker~19/engine/reference/commandline/ps/index Incognito ⌂ ⌂ ⌂

Docker 19 ps

docker ps

Description

List containers

Usage

`docker ps [OPTIONS]`

Options

Name, shorthand	Default	Description
<code>--all , -a</code>		Show all containers (default shows just running)
<code>--filter , -f</code>		Filter output based on conditions provided

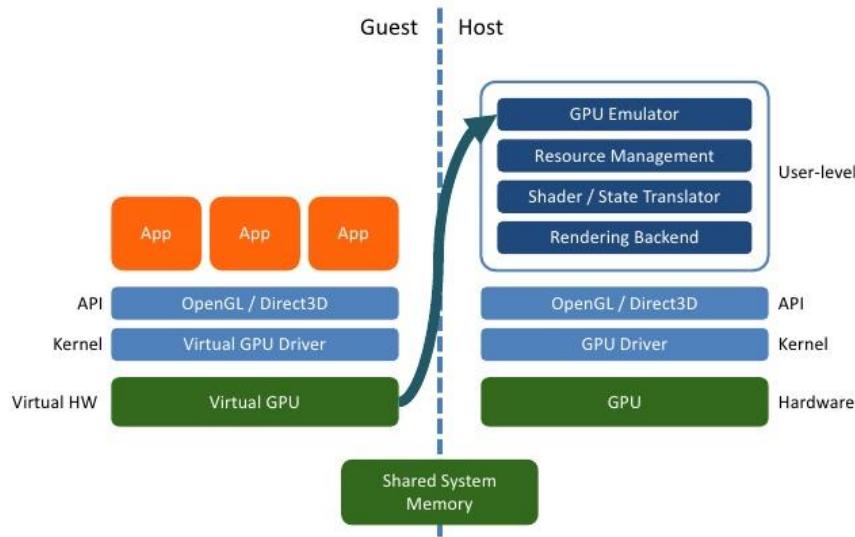
<https://devdocs.io>



Emulation, virtualization and containerization



Emulation

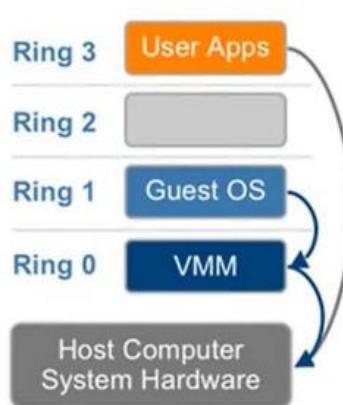


Source: [vmware](#)



Virtualization

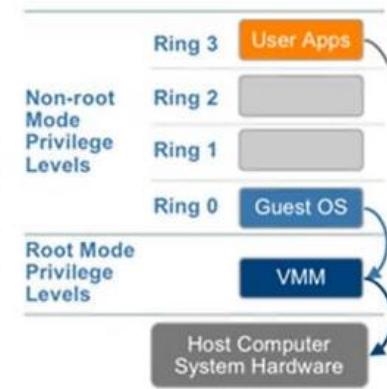
Full Virtualization



Paravirtualization



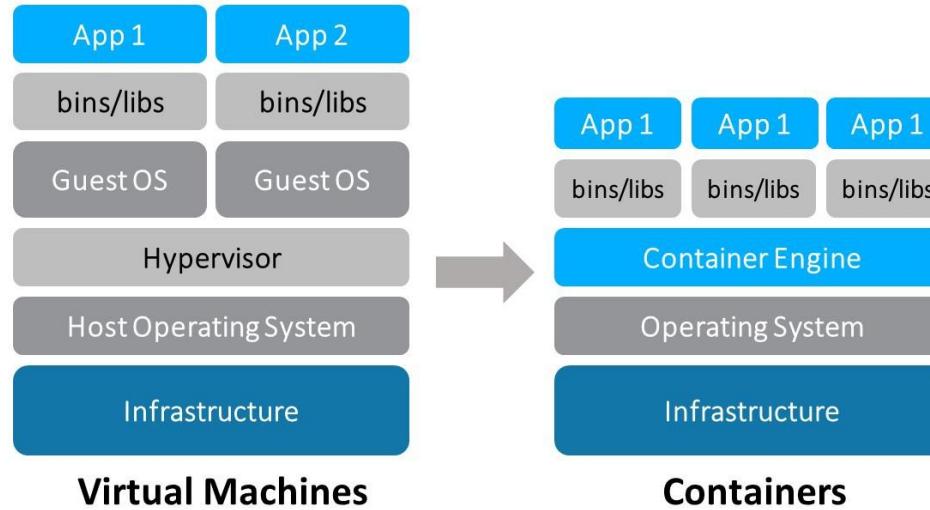
Hardware Assisted



Source: [Zack's Ad hoc Page](#)



Virtualization vs Containerization



Source: [Medium.com](#)



Why containers?



Why containers?

Virtual machines

Occupies a lot of memory space,
requires pre-allocation of RAM memory

Containers

Use a lot less memory space which
can be limited if needed and is unlimited by default



Why containers?

Virtual machines

Long boot-up and restart times

Containers

Short start and restart times



Why containers?

Virtual machines

Compatibility issues when porting across different platforms and operating systems

Containers

Work on different operating systems using the same "build, ship, run" tool



Why containers?

Virtual machines

Partially closed-source or commercial software

Containers

Open-source with more than 1000 contributors,
built on open standards like OCI and backed by
big technology companies (Google, Red Hat, Docker)



Why containers?

Virtual machines

Hard to setup a local environment (for debugging)

Containers

The exact same commands can be run by developers, administrators, DevOps, QA engineers and inside a CI/CD job, no matter of used operating system



Why containers?

Virtual machines

Heavy VM images including the entire guest operating system

Containers

Lightweight and portable container images built with layers which can be shared between multiple images



Total Size:
2 GiB

Unique Layers:
62

Average Layer Size:
32 MiB

Largest Layer Size:
349 MiB

ruby:latest
729 MiB
Layers: 18

python:latest
674 MiB
Layers: 13

node:latest
651 MiB
Layers: 10

golang:latest
744 MiB
Layers: 14

java:latest
643 MiB
Layers: 14

php:latest
490 MiB
Layers: 13

ADD file:dc2eddd5d35b9d66e4db747f5939b2be7f863dcee64c934b0da690f55a23aae8 in /
125 MiB

CMD "/bin/bash"
0 Bytes

RUN apt-get update && apt-get install -y --no-install-recommends ca-certificates curl wget && rm -rf /var/lib/apt/lists/*
44 MiB

RUN apt-get update && apt-get install -y --no-install-recommends bzip2 git mercurial openssh-client subversion procps && rm -rf /var/lib/apt/lists/*
123 MiB

RUN apt-get update && apt-get install -y --no-install-recommends autoconf automake bztp2 file g++ ...
318 MiB

RUN apt-get update && ap...
134 MiB

RUN apt-get update && ap...
1 MiB

RUN apt-get update && ap...
178 MiB

RUN apt-get update && ap...
17 MiB

RUN mkdir -p /usr/local/et...
45 Bytes

RUN apt-get purge -y pyth...
988 KIB

RUN set -ex && for key in 9...
79 KIB

ENV GOLANG_VERSION=...
0 Bytes

RUN echo 'deb http://http...
61 Bytes

RUN mkdir -p \$PHP_INI_DL...
0 Bytes

ENV RUBY_MAJOR=2.3
0 Bytes

ENV LANG=C.UTF-8
0 Bytes

ENV NPM_CONFIG_LOGL...
0 Bytes

ENV GOLANG_DOWNLOA...
0 Bytes

ENV LANG=C.UTF-8
0 Bytes

ENV GPG_KEYS=1A4E8B7...
0 Bytes

ENV RUBY_VERSION=2.3.1
0 Bytes

ENV GPG_KEY=97FC712E...
0 Bytes

ENV NODE_VERSION=6.2.0
0 Bytes

ENV GOLANG_DOWNLOA...
0 Bytes

RUN { echo '#!/bin/sh'; ech...
87 Bytes

ENV PHP_VERSION=7.0.6
0 Bytes

ENV RUBY_DOWNLOAD_...
0 Bytes

ENV PYTHON_VERSION=3...
0 Bytes

RUN curl -SLO "https://no...
42 MiB

RUN curl -fsSL "\$GOLANG...
318 MiB

ENV JAVA_HOME=/usr/lib/j...
0 Bytes

ENV PHP_FILENAME=php-...
0 Bytes



2020 Stack Overflow Developer Survey

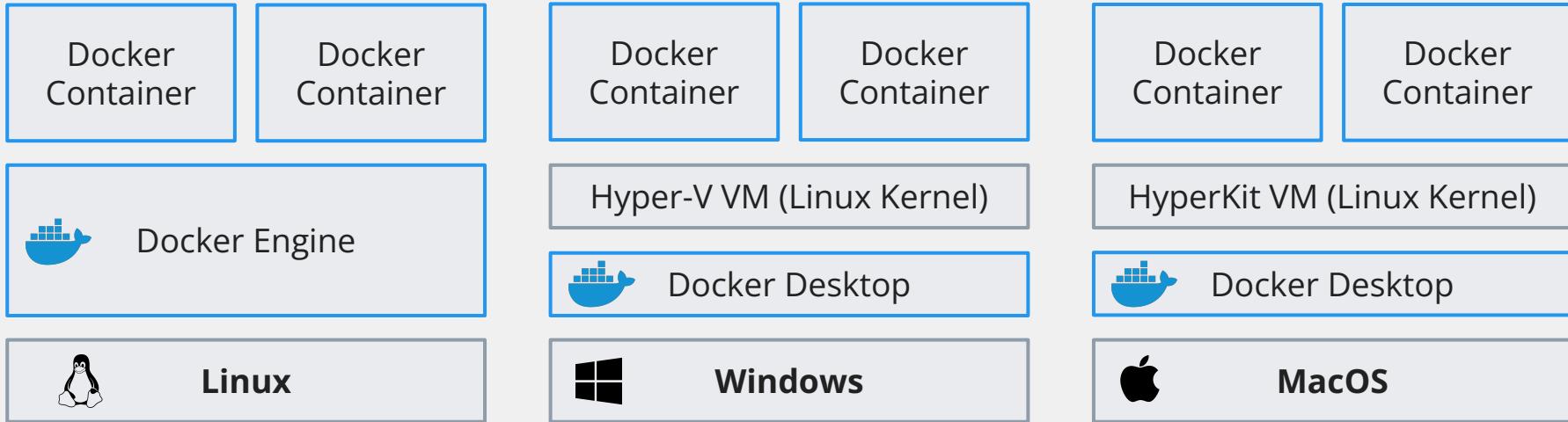
- As for 2020, **in the most wanted platform category, nearly 25% of developers expressed interest in developing with Docker**, ahead of AWS (20.2%), Kubernetes (18.5%) and Linux (16.6%).
- **In the most loved platform category, 73.6% of developers expressed interest in continuing to develop with Docker**, trailing only first-placed Linux (76.9%), and leading Kubernetes (71.1%) and AWS (66.4%).
- **And in the most popular platform category, 35% of respondents chose Docker**, behind first-placed Linux (55%) and second-placed Windows (53.1%).



How do Linux containers run on Windows and Mac



Docker on Linux, Windows and Mac



Docker Desktop VM

You can enter the shell of an underlying Hyper-V / HyperKit virtual machine to browse running processes and the filesystem.
This is not needed on Linux as there is no virtual machine running.

```
bash  
# Windows and Mac  
$ docker run --rm -it --privileged --pid=host \  
justincormack/nsenter1  
  
# Mac  
$ screen \  
~/Library/Containers/com.docker.docker/Data/vms/0/tty
```



Host operating system and kernel in terms of Docker containers



Operating system containers

are virtual environments that share
the kernel of the host operating system
but provide user space isolation.

[LXC](#), [OpenVZ](#), [Linux VServer](#), [BSD Jails](#) and [Solaris zones](#)



While operating system containers are designed to run multiple processes and services, **application containers** are designed to run a single service.

Docker, OpenShift



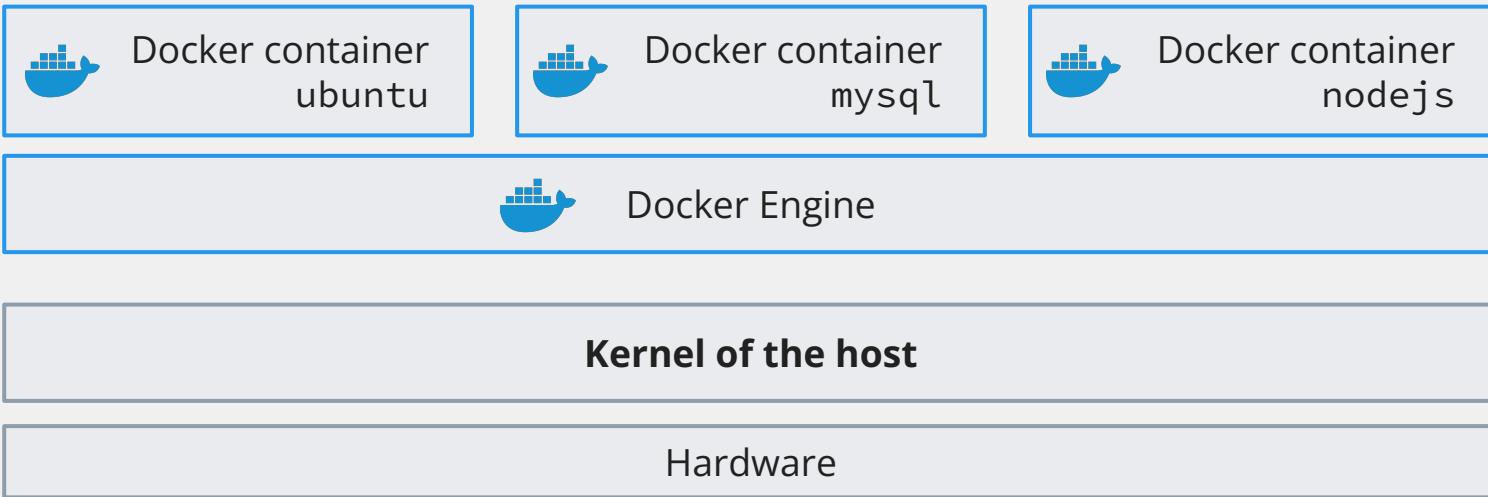
The difference between
a **containerized process** and an ordinary
process is the restrictions put
on the containerized one and how
it sees the environment around it.



Kernel can be shared across different Linux distributions, even if each has its own configuration of the kernel.



Shared kernel



Installing Docker Engine on Linux



Automated installation script available
on <https://get.docker.com> allows
to install Docker Engine on **all supported**
Linux distributions and architectures.



Install Docker on Linux

Automatic install and upgrade script works
for Ubuntu, Debian, Fedora, CentOS, also Raspbian.

```
bash

# Install newest stable Docker
$ curl -fsSL https://get.docker.com | sh

# Install Docker 19.03
$ curl -fsSL https://get.docker.com \
      | VERSION=19.03 CHANNEL=stable sh

# Install Docker under your $HOME as a non-root
$ curl -fsSL https://get.docker.com/rootless | sh
```



Install Docker Desktop on Windows and Mac



Docker Desktop on Windows and Mac

The Docker Desktop installation includes:

- Docker Engine,
- Docker CLI client (`docker` command),
- Docker Compose (`docker-compose` command),
- Notary,
- Kubernetes (`kubectl` command),
- Credential Helper.



Docker Desktop on Windows

- **Windows 10 64-bit** (Pro, Enterprise, or Education).
Windows 10 Home 64-bit with WSL2.
- Hyper-V and Containers Windows features must be enabled.

Requirements to successfully run Client Hyper-V on Windows 10:

- 64-bit processor with Second Level Address Translation (SLAT)
- 4GB system RAM
- BIOS-level hardware virtualization support
must be enabled in the BIOS settings.



Hyper-V

Enable Hyper-V to create virtual machines on Windows.

Hyper-V can be enabled in many ways including using the Windows control panel, PowerShell or using the Deployment Imaging Servicing and Management tool. Hyper-V is built into Windows as an optional feature.

[https://docs.microsoft.com/en-us/virtualization/
hyper-v-on-windows/quick-start/enable-hyper-v](https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/quick-start/enable-hyper-v)



Docker Desktop for Windows

hub.docker.com/editions/community/docker-ce-desktop-windows/

Explore Pricing Sign In Sign Up

Docker Desktop for Windows

 Docker Desktop for Windows

By Docker

The fastest and easiest way to get started with Docker on Windows

Edition Windows x86-64

Get Docker Desktop for Windows

Docker Desktop for Windows is available for free.

Requires Microsoft Windows 10 Professional or Enterprise 64-bit, or Windows 10 Home 64-bit with WSL 2.

By downloading this, you agree to the terms of the [Docker Software End User License Agreement](#) and the [Docker Data Processing Agreement \(DPA\)](#).

[Get Docker](#)

Description Reviews Resources

<https://hub.docker.com/editions/community/docker-ce-desktop-windows/>



Docker Desktop on Windows

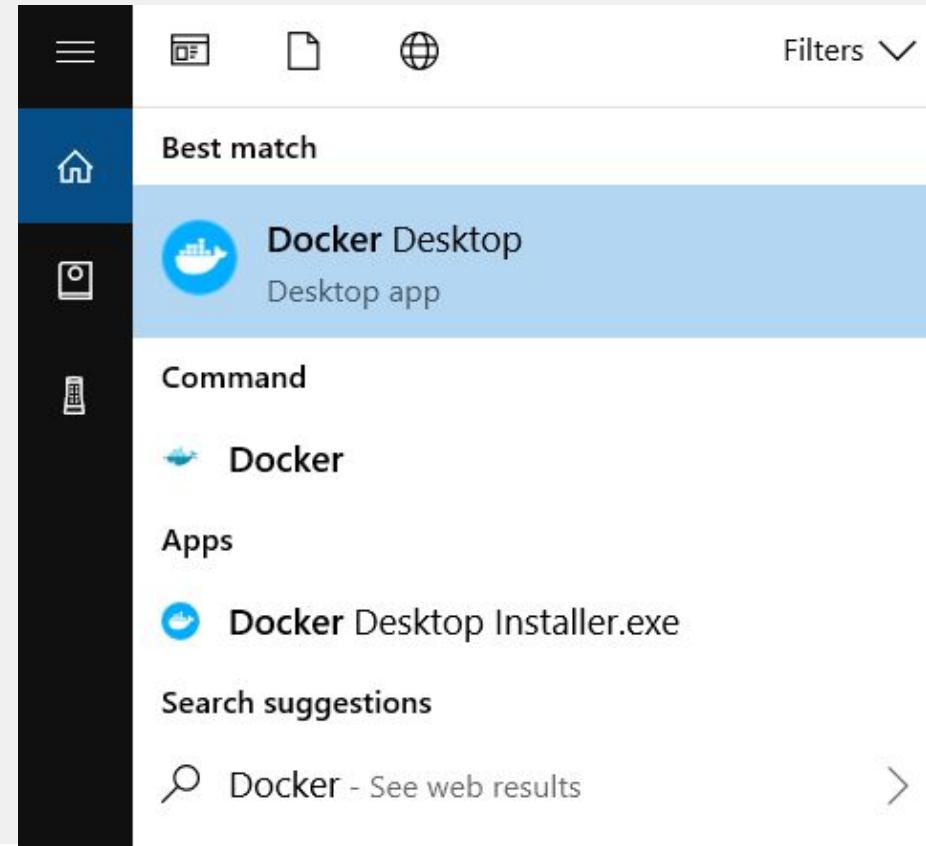
1. Double-click Docker Desktop Installer.exe to run the installer.
2. When prompted, ensure the Enable Hyper-V Windows Features option is selected on the Configuration page.
3. Follow the instructions on the installation wizard to authorize the installer and proceed with the install.
4. When the installation is successful, click Close to complete the installation process.
5. If your admin account is different to your user account, you must add the user to the docker-users group. Run Computer Management as an administrator and navigate to Local Users and Groups > Groups > docker-users. Right-click to add the user to the group. Log out and log back in for the changes to take effect.



Docker Desktop on Windows

Docker Desktop does not start automatically after installation.

To start Docker Desktop, search for Docker, and select Docker Desktop in the search results.



Docker Desktop on Mac

- **macOS must be version 10.14 or newer**
(Mojave, Catalina, Big Sur or newer)
- **Support for the Hypervisor framework**, which can be verified
by running `sysctl kern.hv_support` in the terminal
- At least 4 GB of RAM



Docker Desktop for Mac

hub.docker.com/editions/community/docker-ce-desktop-mac/

Explore Pricing Sign In Sign Up

Docker Desktop for Mac

 Docker Desktop for Mac

By Docker

The fastest and easiest way to get started with Docker on Mac

Edition macOS x86-64

Get Docker Desktop for Mac

Docker Desktop for Mac is available for free.

Docker Desktop - macOS must be version 10.14 or newer: i.e. Mojave (10.14), Catalina (10.15), or Big Sur (11.0). Mac hardware must be a 2010 or a newer model with an Intel processor.

By downloading this, you agree to the terms of the [Docker Software End User License Agreement](#) and the [Docker Data Processing Agreement \(DPA\)](#).

 Get Docker

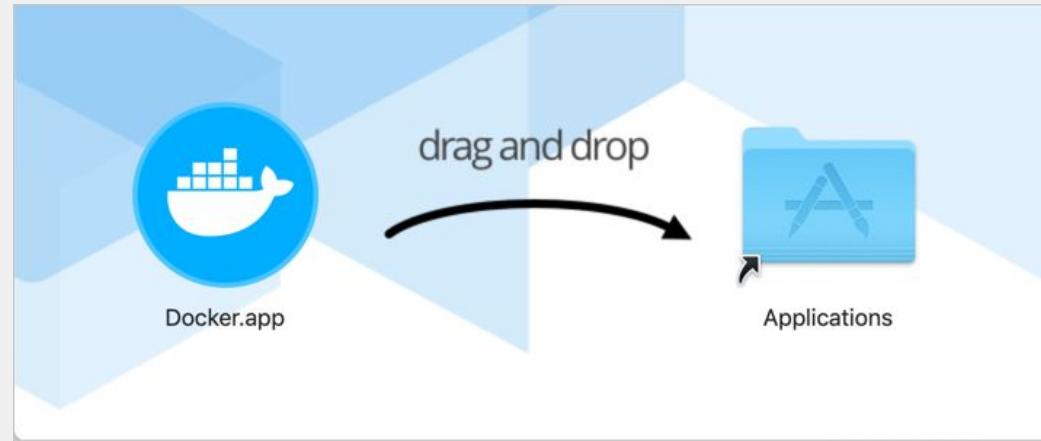
Description Reviews Resources

<https://hub.docker.com/editions/community/docker-ce-desktop-mac/>



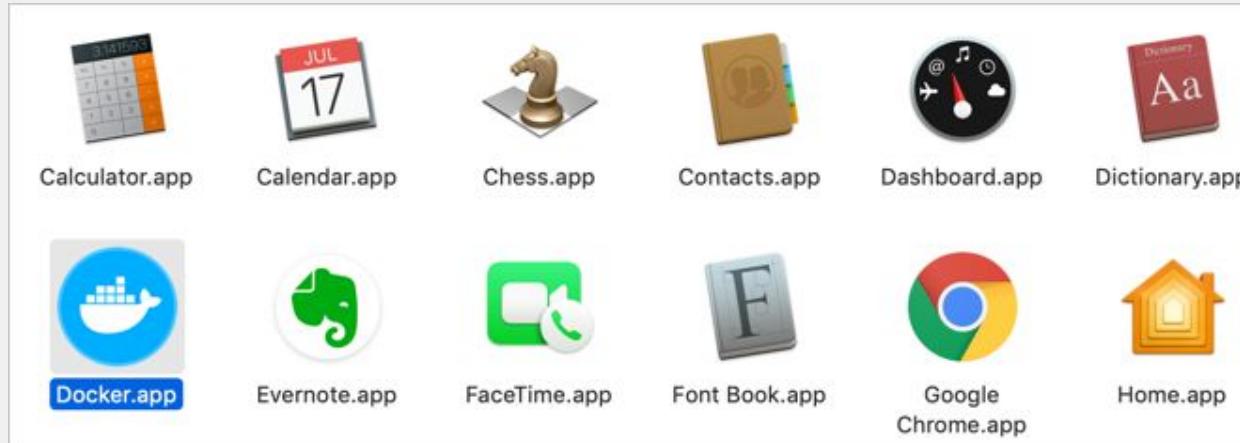
Docker Desktop on Mac

Double-click Docker .dmg to open the installer,
then drag the Docker icon to the Applications folder.



Docker Desktop on Mac

Double-click Docker.app in the Applications folder to start Docker.



Docker Engine and Client configuration



Docker Engine configuration

Configure the Docker daemon using a JSON file:

C:\ProgramData\docker\config\daemon.json on Windows

Whale > Preferences > Daemon > Advanced on Mac

Reload or restart of the Docker Engine is needed to load the new configuration.



The default location of the configuration file on Linux is `/etc/docker/daemon.json`.

The `--config-file` flag can be used to specify a non-default location.



Some options can be **reconfigured when the daemon is running without requiring to restart the process.**

We use the **SIGHUP** signal in Linux to reload, and a global event in Windows with the key **Global\docker-daemon-config-\$PID**.



Configuration files

Docker Engine

<https://docs.docker.com/engine/reference/commandline/dockerd/#daemon-configuration-file>

Docker Client

<https://docs.docker.com/engine/reference/commandline/cli/#configuration-files>



Docker Engine signals

Docker Engine handles two signals you can use to control it:

- | | |
|---------|--|
| SIGHUP | reload the configuration |
| SIGUSR1 | force a full stack trace to be logged,
even if the daemon is unresponsive |

```
$ sudo kill -SIGUSR1 $(pidof dockerd)
```



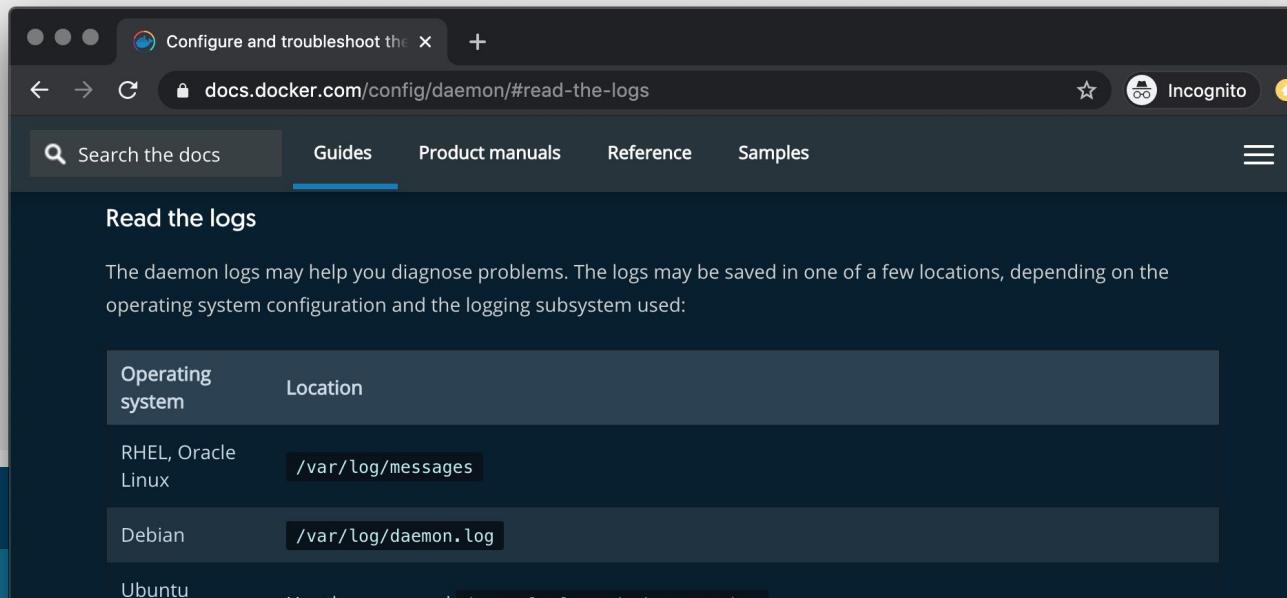
Accessing Docker Engine logs and files



Docker Engine logs

Engine logs are stored in one of a few locations.

Navigate to lach.dev/docker-engine-logs for details on your OS.



The screenshot shows a dark-themed web browser window. The address bar contains the URL `docs.docker.com/config/daemon/#read-the-logs`. The page title is "Configure and troubleshoot the daemon". The navigation menu includes "Search the docs", "Guides" (which is underlined), "Product manuals", "Reference", and "Samples". Below the menu, a section titled "Read the logs" is visible. The text states: "The daemon logs may help you diagnose problems. The logs may be saved in one of a few locations, depending on the operating system configuration and the logging subsystem used:". A table lists log locations for different operating systems:

Operating system	Location
RHEL, Oracle Linux	/var/log/messages
Debian	/var/log/daemon.log
Ubuntu	/var/log/syslog



Docker daemon CPU/RAM/HDD resources usage



Check what is eating the disk

Show total disk space used by Docker with `docker system df` command. Pass the `-v` parameter for a detailed per-object view.



bash

```
$ docker system df
```

TYPE	TOTAL	ACTIVE	SIZE	RECLAIMABLE
Images	13	3	1.563GB	631.9MB (40%)
Containers	4	1	133.4MB	0B (0%)
Local Volumes	12	1	3.6GB	2.576GB (71%)
Build Cache	0	0	0B	0B

```
$ docker system df -v
```



Containers resource usage

Display a live stream of containers resource usage statistics, including CPU, RAM, network and block I/O, in a terminal.

```
bash  
$ docker stats  
CONTAINER ID  NAME          CPU %     MEM USAGE / LIMIT      MEM % ...  
7aa86357b9bc  orca          0.00%    1.281MiB / 2.934GiB  0.04%  
621b0fef5728  orca_mysql_1  0.18%    180.4MiB / 2.934GiB  6.01%  
  
$ docker stats --no-stream \  
--format '{{.Name}}\t{{.CPUPerc}}'  
orca          0.00%  
orca_mysql_1  0.16%
```



Prune (unused) Docker objects



Docker objects

Docker maintains and stores several types of object types:

- Images
- Containers
- Local daemons
- Volumes
- Networks
- Swarm nodes and services



A dangling object

In programming languages like Java or Go **a dangling block of memory** is a block that is not referenced by any piece of code.

The garbage collection system periodically marks the dangling blocks and return it back to the heap, so that these memory blocks are available for future allocations.



A dangling image and layer

A **dangling image** is one that is not tagged and is not referenced by any container.

Similarly a **dangling file system layer** is something that is unused and is not being referenced by any images.

```
$ docker images --filter "dangling=true"  
$ docker rmi $(docker images -f "dangling=true" -q)
```



Prune Docker objects

Docker takes a conservative approach to cleaning up unused objects, such as images, containers, volumes, and networks - these objects are generally not removed unless you explicitly ask Docker to do so.

```
$ docker [image|container|network|volume] prune  
garbage collect a specific Docker type of objects
```

```
$ docker system prune  
a shortcut that prunes images, containers, and networks
```



Prune the system

Call the docker system prune command to clean up dangling images, unused containers, stale volumes and networks.

-a / --all

Keep only the things that are actually running on your system.

-af / --all --force

Do not ask for confirmation.



Prune the system

Free some of the disk space by removing unused and non-referenced Docker objects.

```
● ● ● bash
$ docker system prune -f
Deleted Containers:
0973f3a9a127e4f4c85dc819d021ac3b507dda90e0648f5e991971675a9cc
f0b7235b8d388450e3cc4e05a0c92361766cca066b9d0e07e4858053b7a63
fc57ac3e26b587b82b4487af701d10310579e184440a028a822fac472c030
Deleted Networks:
orca_default
Deleted Images:
deleted: sha256:eab451c7c33f81e99b12820cf485b70ddad3fc10e92a8
```



Troubleshoot the Docker daemon



Troubleshoot the Docker daemon

docker events

Debug mode

Dump a stack trace

Direct HTTP communication



Docker Engine configuration

Configure the Docker daemon using a JSON file:

C:\ProgramData\docker\config\daemon.json on Windows

Whale > Preferences > Daemon > Advanced on Mac

Reload or restart of the Docker Engine is needed to load the new configuration.



Docker Engine signals

Docker Engine handles two signals you can use to control it:

- | | |
|---------|--|
| SIGHUP | reload the configuration |
| SIGUSR1 | force a full stack trace to be logged,
even if the daemon is unresponsive |

```
$ sudo kill -SIGUSR1 $(pidof dockerd)
```



Send HTTP request to a Docker Engine

You can use any HTTP client to act as a Docker client.

For example using curl with a --unix-socket parameter.



```
bash | Training

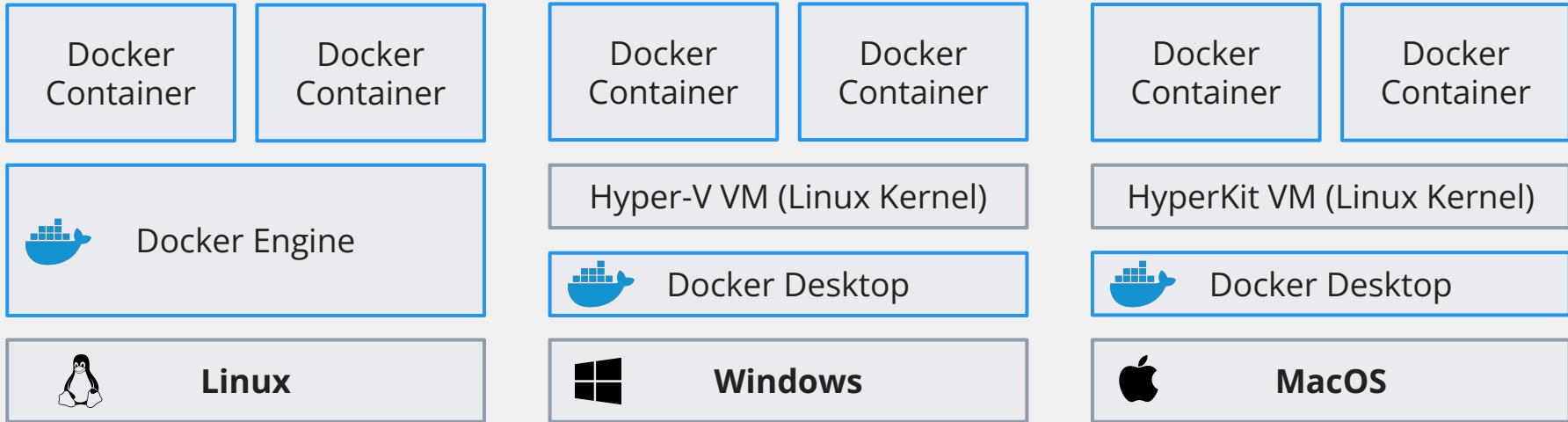
$ curl -sSf --unix-socket /var/run/docker.sock \
  0/containers/json
[
{
  "Id": "535fd2f2d2021a0b5046922e5d38ba669e342a6897065c5f59d4120453",
  "Names": [
    "/nginx"
  ],
  "Image": "nginx",
  "ImageID": "sha256:719cd2e3ed04781b11ed372ecb6190bf76b7d18cb50105",
  "Created": 1595205400,
  "Status": "running",
  "Ports": [
    {
      "Protocol": "tcp",
      "Port": 80
    }
  ],
  "HostConfig": {
    "Binds": [
      "/var/run/docker.sock:/var/run/docker.sock"
    ],
    "PortBindings": {
      "80/tcp": [
        {
          "HostIp": "0.0.0.0",
          "HostPort": "80"
        }
      ]
    }
  }
}
```



Docker on Linux, Windows and Mac



Docker on Linux, Windows and Mac



Docker Desktop

\$ Fully featured Docker Engine with desktop interface, allows to run Linux containers just after installation.

\$ Docker Volume driver allows to share files between containers and Mac or Windows host.

\$ Registry credentials are securely stored (using MacOS Keychain or Windows Credential Store).

\$ Docker Desktop on Windows allows to switch between Windows and Linux containers mode and work on multi-platform apps.

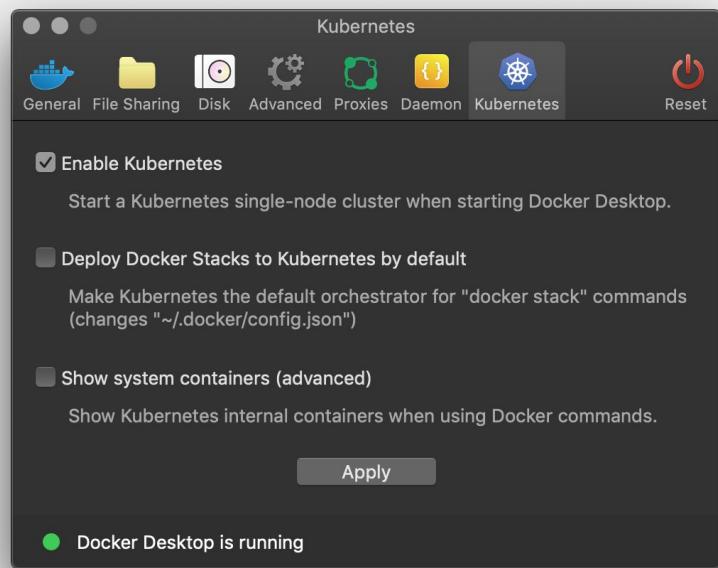


Docker Desktop and Kubernetes

\$ Kubernetes and Docker Desktop share the same image store, so built images can be immediately used in Kubernetes deployments, without having to setup a local registry.

\$ Local Kubernetes cluster can be disabled and resumed anytime (to save battery life).

\$ Kubernetes load balanced services are published on localhost.



Docker Volumes

[Linux](#) - bind-mounting a directory selectively exposes host resources directly to a container. Consequently, access to bind mounts carries little-to-no overhead compared to filesystem access in a regular process.

[Mac and Windows](#) - containers running on top of the Linux system cannot directly access the host filesystem or networking resources, and so Docker Desktop includes libraries that expose those resources in a way that the Docker engine can consume.



User-guided caching

On Mac and Windows you can allow temporary inconsistency which makes it possible to cache filesystem state, avoiding unnecessary communication between the container and the host, and therefore increasing performance.

:consistent Full consistency. The container runtime and the host maintain an identical view of the mount at all times. This is the default.

:cached The host's view of the mount is authoritative. There may be delays before updates made on the host are visible within a container.

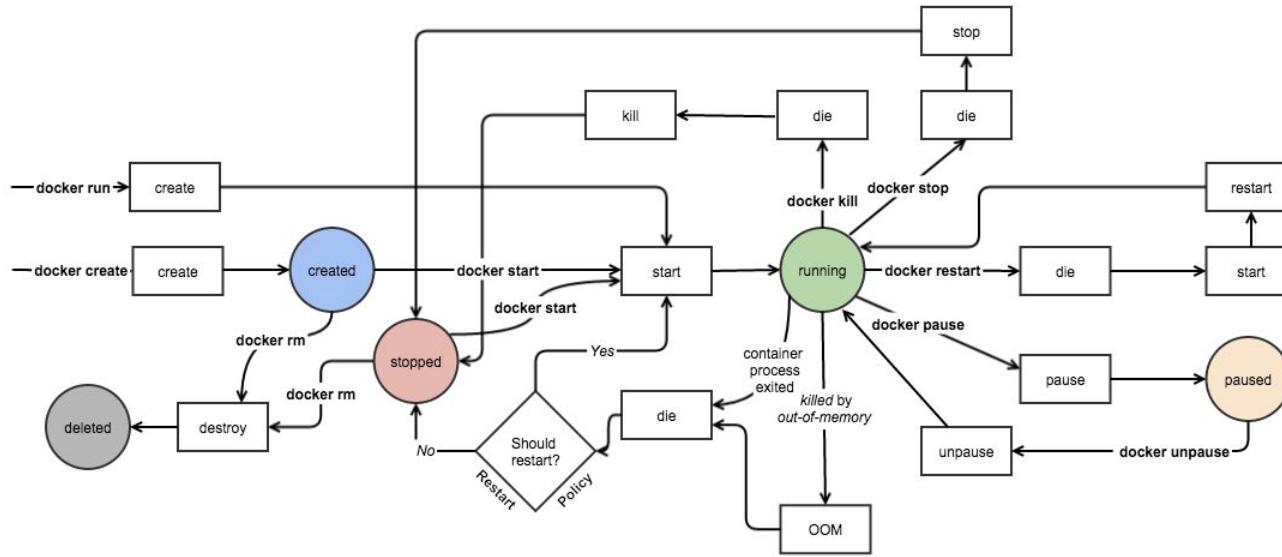
```
$ docker run \
-v /var/www:/var/www:cached \
-v /etc/nginx:/etc/nginx:consistent nginx
```



Container lifecycle



Container lifecycle



Source: [FoxuTech](#)



Running a first Docker container



Hello World

After you have installed Docker Engine and it is up and running, run below command to bring up the first Docker container.



bash

```
$ docker run hello-world
Hello from Docker!
This message shows that your installation appears to be
working correctly.
...
```



Hello World internals

```
$ docker run image
```

client contacts the Docker daemon and sends the command with all parameters and stored credentials

daemon uses the local image or pulls the image from the registry (Docker Hub)

daemon creates a new container from that image and runs the command (passed or the default one)

daemon streams the command output to the Docker client

client sends the output to the terminal



Exit status

The exit code from `docker run` gives information about why the container failed to run or why it exited.

- 125 the error is with Docker daemon itself
- 126 the contained command cannot be invoked
- 127 the contained command cannot be found
- else exit code of contained command otherwise



Hello World

Whatever you type after the image name in docker run is the command you want to execute inside the container.



Enable an **interactive mode** when
running a container to actively work
with it and execute commands inside it.



Containerized OS

You can run the whole different OS using Docker.

Remember that host kernel is shared between all the containers.

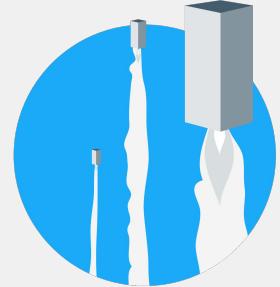
```
bash
$ docker run -it debian:stretch
root@f54f8e846835:/# cat /etc/os-release
PRETTY_NAME="Debian GNU/Linux 9 (stretch)"
NAME="Debian GNU/Linux"
VERSION_ID="9"
VERSION="9 (stretch)"
```



Interactive mode

This is a freshly created new container,
it runs a Debian Stretch operating system.

- it is a shorthand for -i -t
- i tells Docker to bind to the container's standard input
- t tells Docker to allocate a pseudo-TTY (terminal)



Automatic cleanup

If the container you are about to run will not be needed after it is done, you can make Docker cleanup all its resources automatically.

`--rm` automatically remove the container when it exits,
Docker also removes the anonymous volumes associated
with the container when it is removed.



Obtaining a local copy of the Docker image



Download the image

Docker provides a few ways to download images.

Explicit:

\$ docker pull download a newest version of an image

Implicit:

\$ docker run download when not available locally

\$ docker build download when the image pointed by
 the FROM instruction is not available locally



When you run an image, **Docker will always use its local copy if available.**

You need to explicitly call `docker pull` to download the fresh version.



Pull an image

Use the docker pull command to fetch all the necessary layers of a nginx image with tag :1.19, which indicates the exact version of this software.

```
● ● ● bash  
$ docker pull nginx:1.19  
1.19: Pulling from library/nginx  
bb79b6b2107f: Downloading 8.913MB/27.09MB  
111447d5894d: Downloading 7.799MB/26.49MB  
a95689b8e6cb: Download complete  
1a0022e444c2: Download complete  
32b7488a3833: Download complete
```



Image pull states

Downloading

download the layer from a remote registry
layer was downloaded

Pull complete

extract the layer tar archive

Extracting

layer found locally

Already exists

verify the Docker Content Trust signature

Verifying checksum

image with all layers was downloaded

Download complete



Running containers in the daemon mode



Non-interactive detached mode

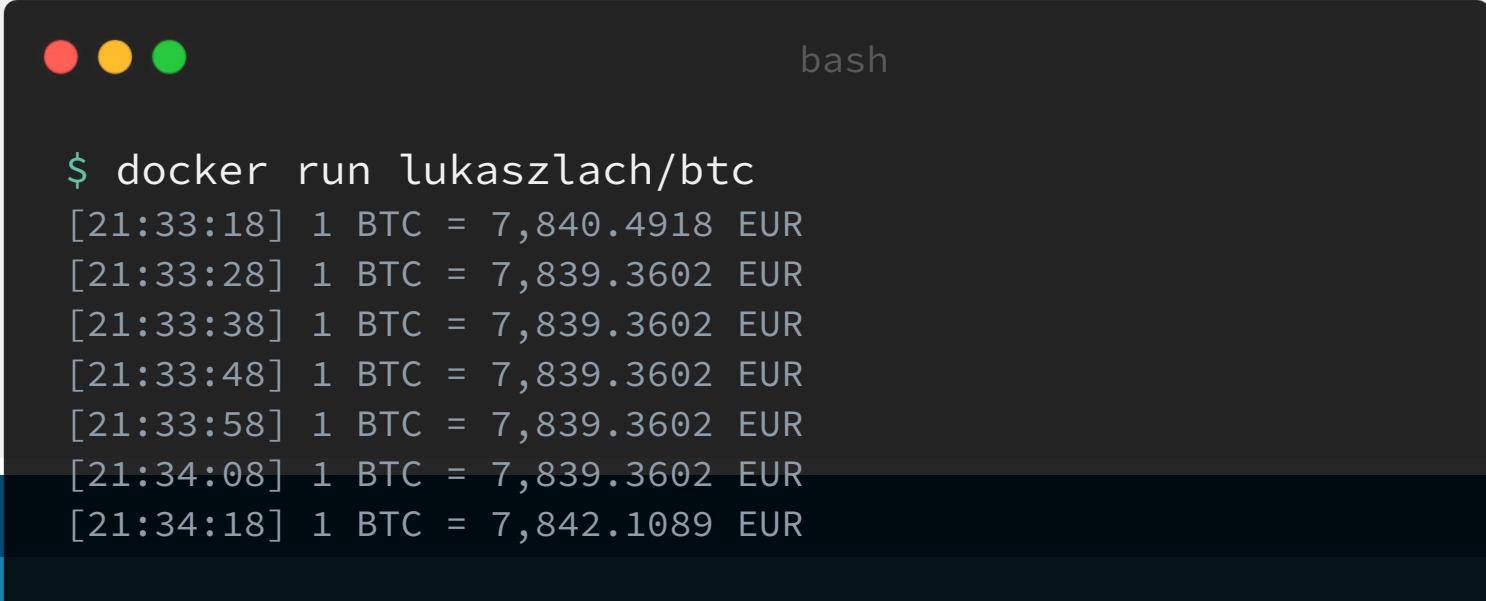
An interactive process reads commands from user input on a tty.
But a process inside a Docker container can run in a non-interactive mode, working in an infinite loop.

Nearly all applications are in an infinite loop, that accept user input continuously, process the input and generate some output until user manually exit from the application.



Run a process in an infinite loop

Run a container from `lukaszlach/btc` image, it displays the current BTC/EUR rate every 10 seconds. It will run forever, press `^C` to stop it.



A terminal window with three colored window control buttons (red, yellow, green) at the top left. The title bar on the right says "bash". The terminal content shows the command `$ docker run lukaszlach/btc` followed by a series of timestamped outputs displaying the BTC/EUR exchange rate. The last output is at [21:34:18].

```
$ docker run lukaszlach/btc
[21:33:18] 1 BTC = 7,840.4918 EUR
[21:33:28] 1 BTC = 7,839.3602 EUR
[21:33:38] 1 BTC = 7,839.3602 EUR
[21:33:48] 1 BTC = 7,839.3602 EUR
[21:33:58] 1 BTC = 7,839.3602 EUR
[21:34:08] 1 BTC = 7,839.3602 EUR
[21:34:18] 1 BTC = 7,842.1089 EUR
```



Pass the -d parameter to
the docker run command
to start a container
in a detached (daemon) mode.



Detached mode

Container can be started in the background, meaning "in detached mode". Docker returns the long ID (SHA256) of the container.

```
● ● ● bash  
$ docker run -d lukaszlach/clock  
009ffca7b10ea401b10e3161959f992974e37bec35ca0184d29c1c6c7c...  
$ docker ps  
CONTAINER ID        IMAGE               ...      CREATED  
009ffca7b10e        lukaszlach/clock  ...      3 seconds ago
```



Detached container output

Use the `docker logs` command to show output of the detached container. This will display all logs starting from the beginning, which may be a huge payload.



bash

```
$ docker logs 009
Wed Jul 18 18:51:50 UTC 2020
Wed Jul 18 18:51:51 UTC 2020
Wed Jul 18 18:51:52 UTC 2020
...
```



Container processes and signal handling



PID

Process identifiers (PIDs) are unique identifiers that the Linux kernel gives to each process. **PIDs are namespaced, meaning that a container has its own set of PIDs that are mapped to PIDs on the host system.**

The first process launched when starting a Linux kernel has the PID 1. For a normal operating system, this process is the init system, for example, systemd or SysV. **Similarly, the first process launched in a container gets PID 1.**



Signal handling

A process running as PID 1 inside a container is treated specially by Linux - it **ignores any signal with the default action.**

This means that the process will **not terminate** on SIGINT or SIGTERM or handle any other signal unless it is coded to do so.

The SIGKILL signal cannot be ignored by the process and it never gets the opportunity to catch the signal and act on it.



Zombie process

**A child that terminates,
but has not been waited for becomes a "zombie".**

The kernel maintains a minimal set of information about the zombie process (PID, termination status, resource usage information) in order to allow the parent to later perform a wait to obtain information about the child.



Container processes

Display the running processes of a container with no external dependencies.

```
● ● ● bash  
$ docker run -d --name clock lukaszlach/clock  
9e598513e0a50f153d3990e8df937362ee698833b1668d8ae2376888e0...  
  
$ docker top clock  
PID      USER      TIME      COMMAND  
8598      root      0:00      sh -c while true; do date; ...  
14450      root      0:00      sleep 1
```



Container processes

If there is a command available inside the container you can use `docker exec` to run it and work directly with container processes.

```
● ● ● bash
$ docker exec clock \
  pstree -p
sh(1)---sleep(757)
```



Signal ignoring

If you decide to put any application (including a shell script) as PID 1 running in a container - **make sure it does handle TERM (and INT) signal.**

```
● ● ● bash
$ docker run -d --name sleep \
    alpine:3 sleep 1d
$ docker kill --signal TERM sleep
orca

$ docker ps
CONTAINER ID        IMAGE       ...        NAMES
d18d3d403cb2        alpine:3     sleep
```



Shell script as PID 1

If you want a shell script to run as PID 1 in a container set up a trap in the source code.

```
start.sh

#!/bin/bash
trap '' EXIT
while true; do
    ...
done
```



Shell script as PID 1

Always remember to start the actual application with exec so that it replaces the shell script on PID 1.

```
start.sh

#!/bin/bash
#
# prepare your application, drop privileges etc
#
exec /usr/bin/application -daemon
```



Dropping privileges at runtime

By using tools like gosu or su-exec, Docker can run the container start script as root which then drops the privileges when no longer needed.

```
$ su-exec user[:group] command
```

<https://github.com/ncopa/su-exec>

```
$ gosu user[:group] command
```

<https://github.com/tianon/gosu>



Specify an init process

**You can use the `--init` flag to indicate that
an init process should be used as the PID 1 in the container.**

Specifying an init process ensures the usual responsibilities of an init system, such as reaping zombie processes, are performed inside the created container. The default init process used is the first docker-init executable found in the system path of the Docker daemon process (by default tini).



Specify an init process

When running the previous example with --init parameter included the TERM signal is now properly handled.

```
bash
$ docker run --init -d --name sleep \
    alpine:3 sleep 1d
$ docker kill --signal TERM sleep
orca

$ docker ps
CONTAINER ID        IMAGE       ...
NAMES
$
```



Container name



You can use either the **container ID**
(short ID or even a part of it)
or the **container name** (given or auto-
generated) whenever Docker client
expects a container in the parameters.



The container name

When referencing containers with their IDs, this needs copy/pasting the ID or running `docker ps` to get the shortened ID.

Consider the following:

```
$ docker stop clock  
$ docker rm clock  
$ docker logs clock
```



Default container name

When you run a container and do not give a specific name,
Docker will randomize one for you.

The default name contains:

mood

awesome, nervous, romantic, ...

name of scientist or hacker

Darwin, Curie, DaVinci, ...

lach.dev/docker-name



Specify the container name

Run the clock image in the background, define a custom name with the --name parameter and monitor the logs.

```
● ● ● bash  
$ docker run -d --name clock lukaszlach/clock  
9e598513e0a50f153d3990e8df937362ee698833b1668d8ae2376888e0...  
  
$ docker logs clock  
...  
Wed Feb  3 19:30:49 UTC 2021  
Wed Feb  3 19:30:50 UTC 2021  
  
$ docker kill clock
```



Rename the container

rename the container so you can work on it later.
Use can pass the container ID or name.

```
bash  
$ docker run -d lukaszlach/clock  
442f2d226d273d8a62fca77d88560d3c593c97533ec1cb9c2e8525af4...  
$ docker rename 442 new_clock  
$ docker logs new_clock  
...  
Wed Feb  3 19:30:49 UTC 2021  
Wed Feb  3 19:30:50 UTC 2021
```



Reuse the container name

Container name is bind to all runtime parameters used to run it.
To reuse the container name you first need to free it.

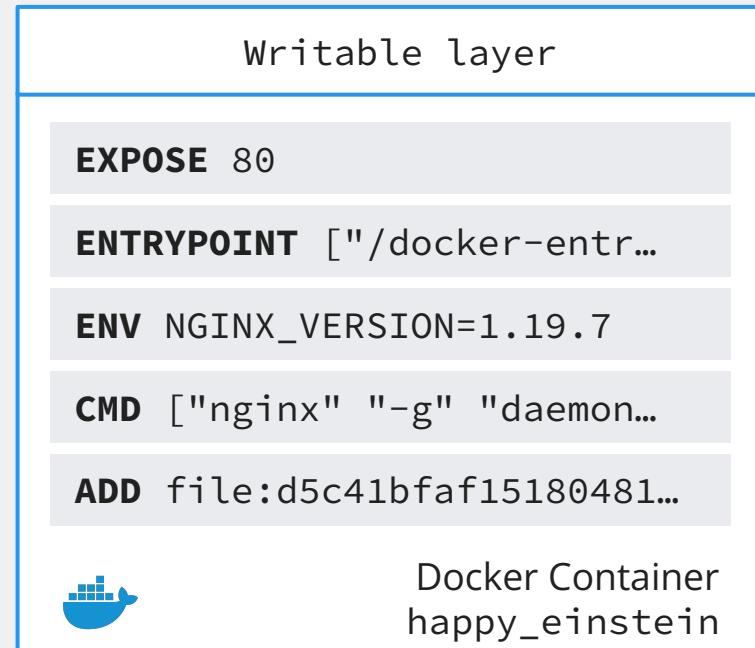
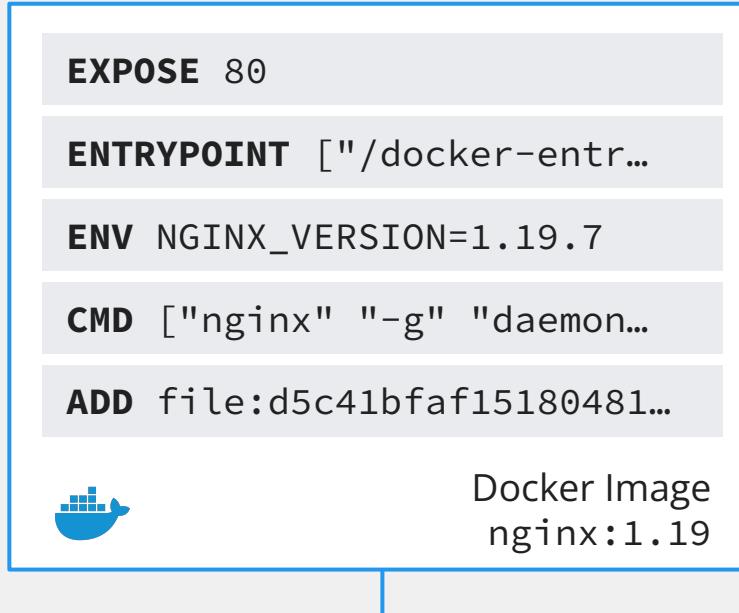
```
bash  
  
# Missing -it parameters cause container to run and exit  
$ docker run --name alpine alpine:3.9  
$ docker run --name alpine -it alpine:3.9  
docker: Error response from daemon: Conflict. The container name  
"/alpine" is already in use by container "3be5c4e2b0...". You have to  
remove (or rename) that container to be able to reuse that name.  
See 'docker run --help'.  
$ docker rm alpine
```



Running multiple Docker containers



Docker image instance

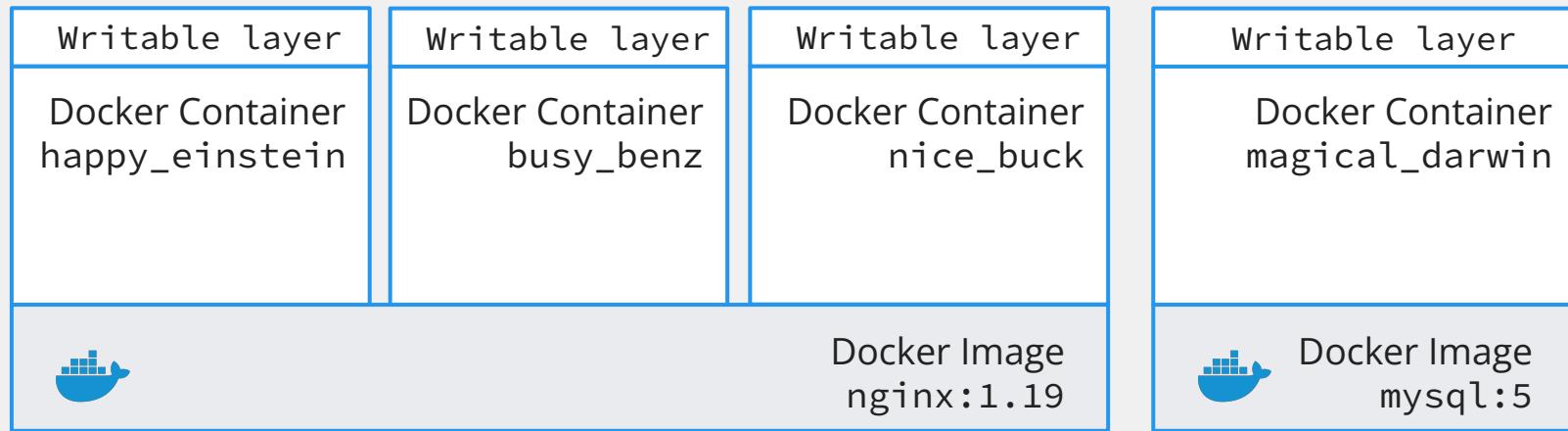


docker run nginx:1.19

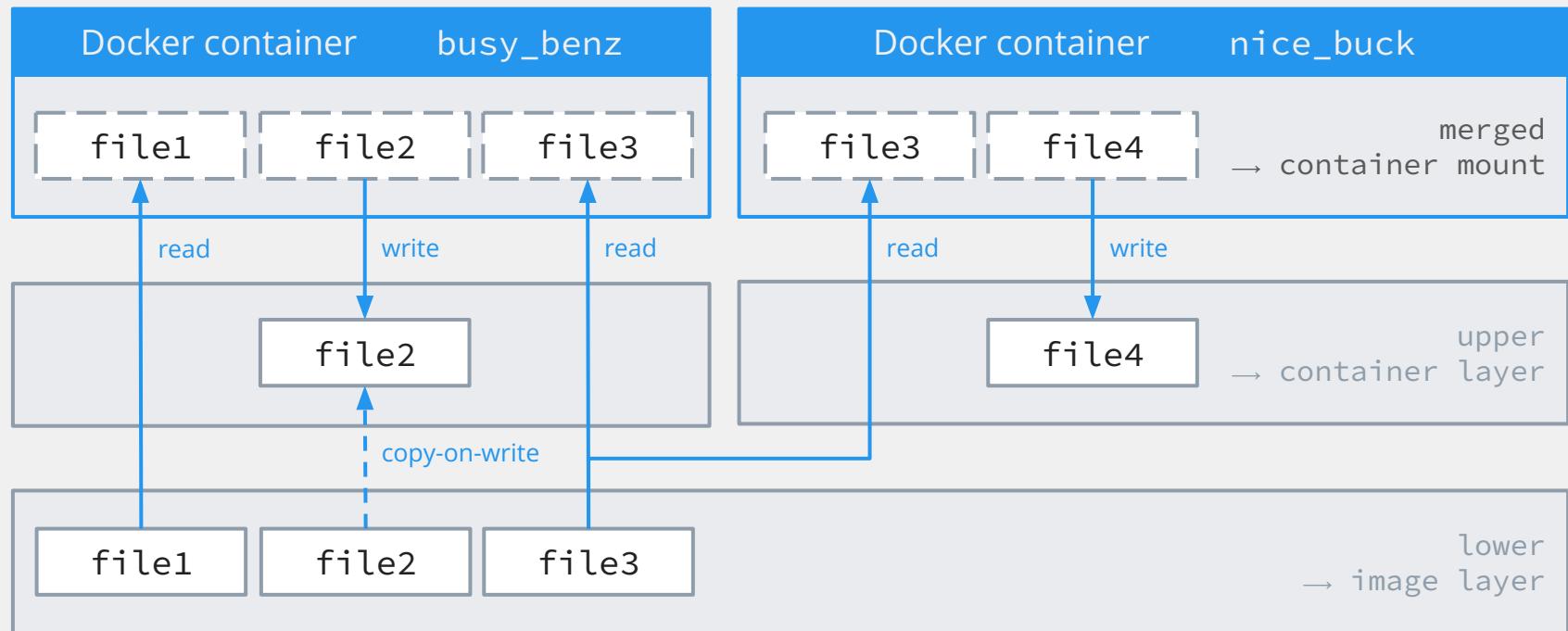


Multiple containers from the same image

Many containers ran from the same image use the read-only filesystem provided by it, each container has its own writable layer.



Container filesystem



Try it out

Run many web server instances with published port starting from 80 up to the total number of containers set in the COUNT variable

```
bash

# Mind CPU and RAM resources you have available
$ COUNT=20
$ for i in $(seq 0 $COUNT); do \
    docker run --rm -d \
    -p "$(((80+$i))):80" \
    --name "nginx_${i}" \
    nginx:1.19; \
done
$ docker stop $(docker ps -f ancestor=nginx:1.19 -q)
```



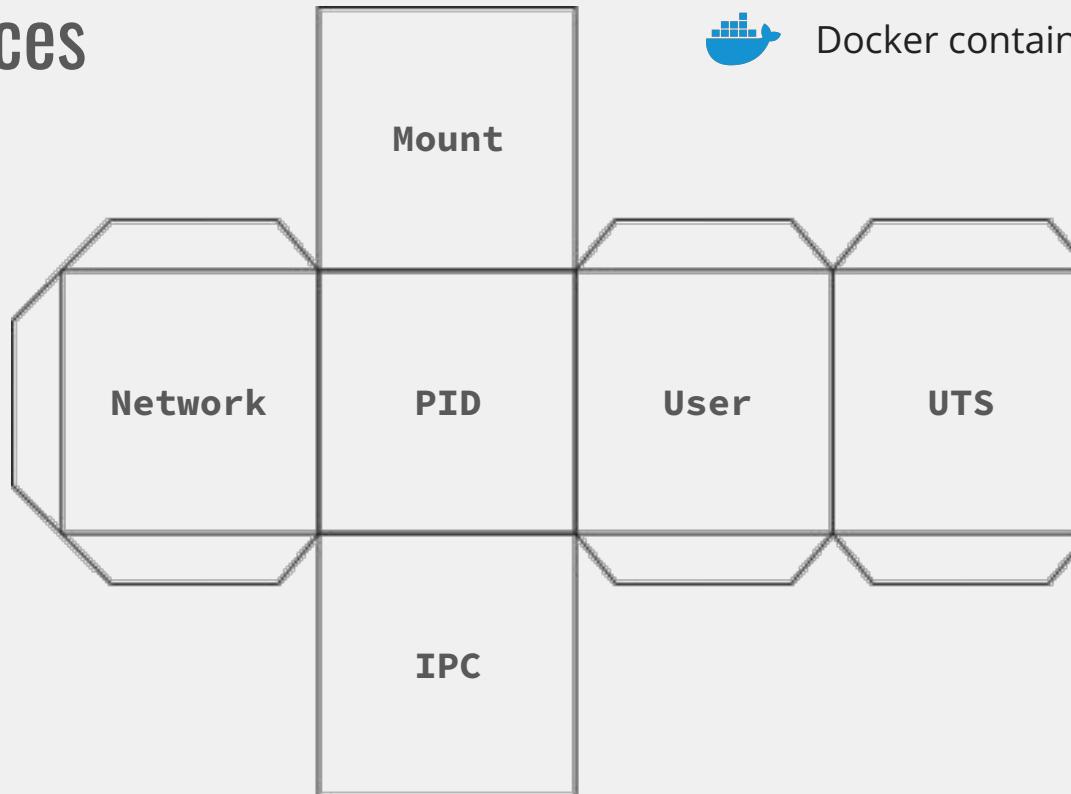
Communication between containers - network, disk, shared memory, unix sockets



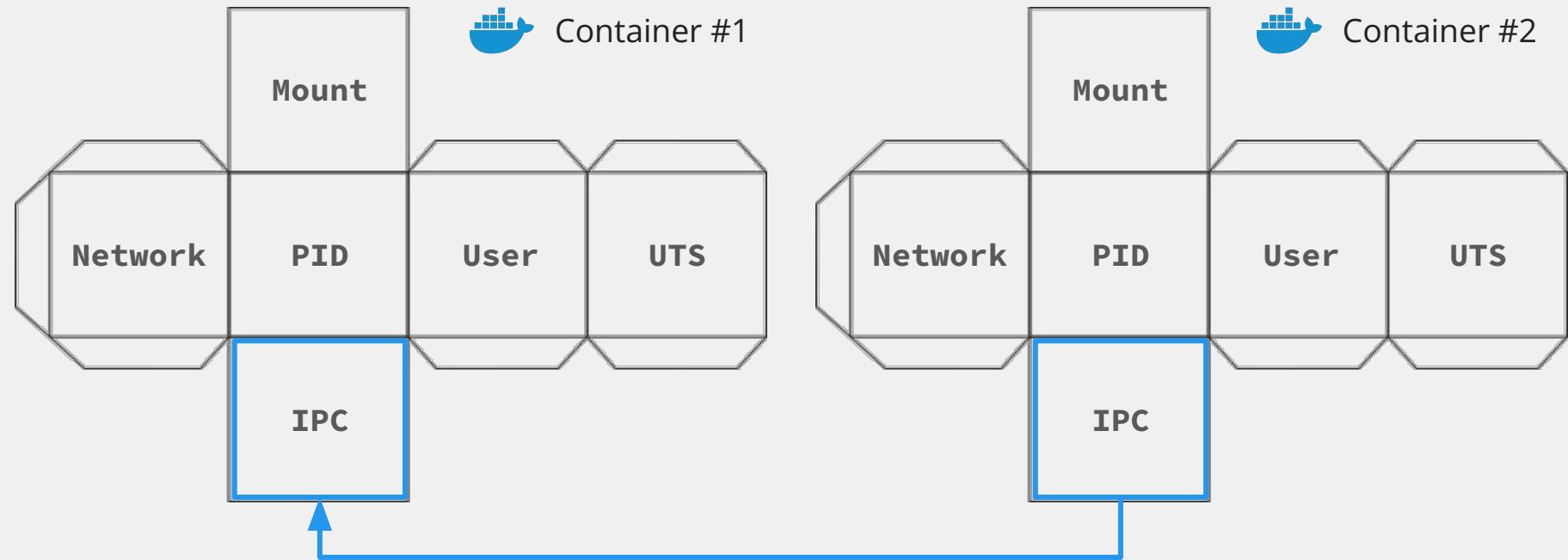
Linux namespaces



Docker container



Linux namespaces



>_

ipcs

Display information about resources used in IPC (Inter-process Communication).

Maintainer: LaMont Jones
<lamont@debian.org>

Section: utils



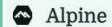
OS X `brew install util-linux`
your system



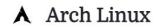
Debian `apt-get install util-linux`



Ubuntu `apt-get install util-linux`



Alpine `apk add util-linux`



Arch Linux `pacman -S util-linux`



Kali Linux `apt-get install util-linux`



CentOS `yum install util-linux`



Fedora `dnf install util-linux`



Raspbian `apt-get install util-linux`



Docker `docker run cmd.cat/ipcs ipcs`
powered by [Commando](#)

Display information about
resources used in IPC (Inter-
process Communication).

Specific information about the Message Queue
which has the id 32768:

`ipcs -qi 32768`

General information about all the IPC:

`ipcs -a`

© tl;dr; authors and contributors

<https://command-not-found.com/ipcs>



Using Docker client to monitor containers



Run a container

Start a `lukaszlach/clock` container in the background.



bash

```
$ docker run --name clock -d lukaszlach/clock  
009ffca7b10ea401b10e3161959f992974e37bec35ca0184d29c1c6c7c...
```



List running containers

List running containers with docker ps command.

Docker returns the short ID, image name and status information.

```
bash  
$ docker ps  
CONTAINER ID        IMAGE               ...      CREATED  
009ffca7b10e        lukaszlach/clock   ...      3 seconds ago
```



Output of the docker ps command is
always sorted by container creation time,
newest first.



List all containers

List all containers including the exited ones.

```
● ● ● bash  
$ docker ps -a  
CONTAINER ID IMAGE ... STATUS  
6e494a61169d lukaszlach/clock Exited (137) 9 minutes ago  
6d6f20e9e926 alpine:3 Exited (0) 6 minutes ago  
009ffca7b10e busybox Exited (0) 6 minutes ago
```



Run more containers

Start two more containers by running the `lukaszlach/clock` image. Verify that `docker ps` correctly reports all three containers.

```
● ● ● bash  
$ docker run -d lukaszlach/clock  
6d6f20e9e9265c59faaa7aac2eed205c399df1b21fa915bc8f4416900...  
$ docker run -d lukaszlach/clock  
2e494a61169d2a58c7003e685ffbd9c270f7def3a40fa9a9f4214ed8d...  
$ docker ps
```



View the last container

Call docker ps with the -l flag to show only the last created container.

```
bash  
$ docker ps -l  
CONTAINER ID      IMAGE          ...      CREATED  
6e494a61169d    lukaszlach/clock ...  2 minutes ago
```



Short container ID

Some Docker commands like stop, kill or rm can work on a list of container IDs. To fetch only IDs of all running containers pass the -q flag.

```
$ docker ps -q  
6e494a61169d  
6d6f20e9e926  
009ffca7b10e
```



Command-line inception

Include -q parameter when calling docker ps with any other options to return only container IDs matching the query.

You can directly pipe this output to any other docker command and trigger an action on all containers at once.

```
$ docker kill $(docker ps -q --filter name=web*)  
$ docker inspect $(docker ps -qf exited=7)  
$ docker stop $(docker ps -q)
```

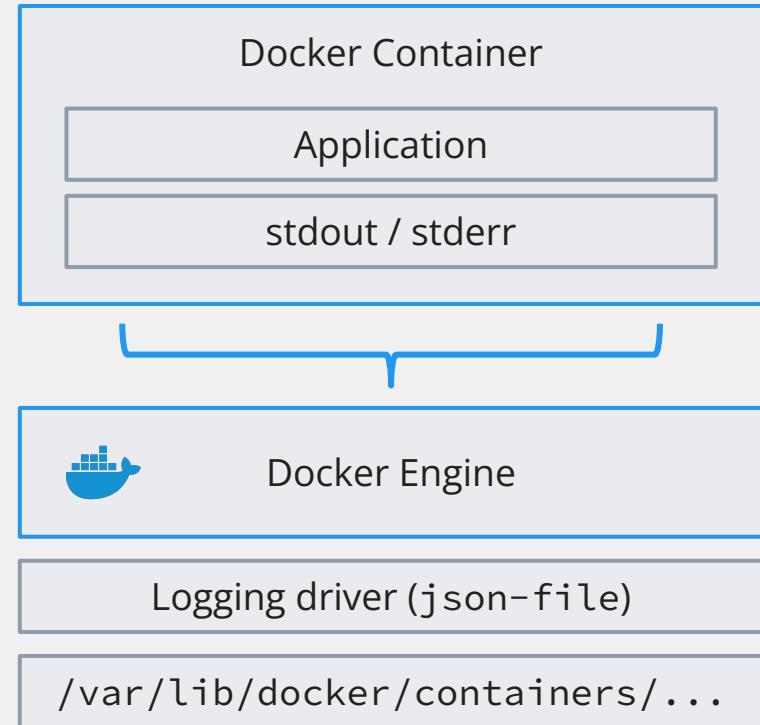


Logging internals

By default, Docker shows the command's output just as it would appear if you ran the command interactively in a terminal, meaning both STDOUT and STDERR are displayed.

These two streams also used to gather logs from the container and are forwarded to remote destinations or files.

The default logging driver is json-file. It stores container logs in JSON format on local disk.



Container logs

Use the `docker logs` command to show output of the detached container. The output will display all logs starting from the beginning, which may be a huge payload.



bash

```
$ docker logs clock
Wed Jul 18 18:51:50 UTC 2018
Wed Jul 18 18:51:51 UTC 2018
Wed Jul 18 18:51:52 UTC 2018
...

```



Most recent logs

Use the `--tail` parameter to display only selected number of lines that you want to fetch.

```
bash
$ docker logs --tail 3 clock
Wed Feb  3 18:39:16 UTC 2021
Wed Feb  3 18:39:17 UTC 2021
Wed Feb  3 18:39:18 UTC 2021
...
```



Follow container logs

Use the `-f` parameter to "follow" the logs of your container, just like in the UNIX `tail` command. This will display the last line in the log file and will continue to display the logs in the real time.

```
bash
$ docker logs --tail 1 -f clock
Wed Feb  3 18:39:16 UTC 2021
Wed Feb  3 18:39:17 UTC 2021
Wed Feb  3 18:39:18 UTC 2021
...
^C
```



Fetch the filtered logs

Container logs can be filtered:

\$ docker logs

--timestamps

Show timestamp on every log line

--details

Show extra details provided to logs

--since 5m

Show logs since timestamp or relative

--until 2019-06-02

Show logs before a timestamp or relative

-f --since 1m

Show logs from last minute and follow



Inspect the container

Display all the information Docker stores about the container in any state, in JSON format (by default).

```
bash
$ docker inspect clock
[
  {
    "Id": "0b0041369c83f5a330d3453b35b1f2ac03c331554876d57a90ab2",
    "Created": "2021-02-03T18:31:59.833063021Z",
    "Path": "sh",
    "Args": [
      "-c",
      "while true; do date; sleep 1; done"
    ],
    ...
  }
]
```



Fetch a single information

Pass the --format parameter to extract only selected information.

```
bash  
$ docker inspect \  
    --format '{{ json .State.Status }}' clock  
"running"  
  
$ docker inspect \  
    --format '{{ print .Path }} {{ join .Args " " }}' clock  
sh -c while true; do date; sleep 1; done  
  
$ docker inspect --format \  
    '{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}' \  
clock  
172.17.0.3
```



Query all containers

Format string can contain advanced logic of filtering all containers by any custom value.

```
bash

# Find the names of all containers that exited with a non-zero code
$ docker inspect -f \
    '{{if ne 0 .State.ExitCode}}{{.Name}} {{.State.ExitCode}}{{end}}' \
$(docker ps -aq) \
| grep .

/nostalgic_blackwell 137
/frosty_gauss 127
/clock 137
/elegant_jones 127
...
```



Containers resource usage

Display a live stream of containers resource usage statistics, including CPU, RAM, network and block I/O, in a terminal.

```
bash
$ docker stats
CONTAINER ID  NAME          CPU %     MEM USAGE / LIMIT   MEM % ...
940a6aca2aab reverent_brahmagupta  0.00%    616KiB / 1.904GiB  0.03%
0b0041369c83  clock          0.00%    604KiB / 1.904GiB  0.03%
0d9cc89dale3  nervous_satoshi  0.08%    628KiB / 1.904GiB  0.03%

$ docker stats --no-stream \
--format '{{.Name}}\t{{.CPUPerc}}'
reverent_brahmagupta 0.00%
clock                0.01%
nervous_satoshi      0.08%
```



Raw container logs

With the docker inspect command you can fetch information about logging driver used and a log file localization.

```
bash  
$ docker inspect \  
    --format '{{.HostConfig.LogConfig.Type}}' clock  
json-file  
  
$ docker inspect --format='{{.LogPath}}' clock  
/var/lib/docker/containers/371ff3cf8d8cde71ca546f0fe6175af233  
f08911b3be0d89309f2a097be52738/371ff3cf8d8cde71ca546f0fe6175a  
f233f08911b3be0d89309f2a097be52738-json.log
```



Container processes

Display the running processes of a container with no external dependencies.

```
● ● ● bash
$ docker top clock
PID      USER      TIME      COMMAND
8598      root      0:00      sh -c while true; do date; ...
14450      root      0:00      sleep 1
```



Container size on disk

To view the approximate size of a running container, use the docker ps -s command. Two different columns relate to size:

size the amount of data (on disk) that is used for the writable layer of each container.

(virtual size) the amount of data used for the read-only image data used by the container plus the container's writable layer size.



Localize the files

Inspect changes to files or directories on a container's filesystem.

```
bash
$ docker diff clock
$ docker exec clock \
    touch /new.file
$ docker diff clock
A /new.file
```



Graceful shutdown of a container



Terminate the container

There are few ways to terminate the detached container:

stop Gracefully stop it using the docker stop command

kill Forcefully kill it using the docker kill command

rm Forcefully kill and remove it with a single
docker rm command with -f ("force") parameter



Signal handling

STOP

```
$ docker stop  
$ docker-compose stop  
$ docker-compose down
```

SIGTERM



10 seconds (by default)

SIGKILL



Stop the container

Use the docker stop command to gracefully stop the container.

```
● ● ● bash  
$ docker run --name clock -d lukaszlach/clock  
009ffca7b10ea401b10e3161959f992974e37bec35ca0184d29c1c6c7c...  
$ time docker stop clock  
clock  
docker stop clock 0.03s user 0.02s system 0% cpu 10.215 total
```



In Kubernetes the graceful stop timeout delay equals to terminationGracePeriod of the pod and defaults to 30 seconds.



Controlling the time

Option 1

Set the time to wait for stop
before killing the container.

```
$ docker stop -t 3
```

```
$ docker-compose stop -t 3
```

stop_grace_period: 3s

Option 2

Change the signal sent
to stop the container.

stop_signal: SIGKILL



Stop all containers

Utilize a subshell to stop all running containers at once.

```
bash
$ docker stop $(docker ps -q)
dd1c733721bf
940a6aca2aab
0d9cc89dale3
```



Forceful shutdown of a container



Terminate the container

There are few ways to terminate the detached container:

stop Gracefully stop it using the docker stop command

kill Forcefully kill it using the docker kill command

rm Forcefully kill and remove it with a single
docker rm command with -f ("force") parameter



Signal handling

KILL

```
$ docker kill  
$ docker-compose kill
```

SIGKILL (by default)



Sending signals

The docker `kill` command is capable of sending any signal.

For some applications, instead of restarting a container you can call docker `kill` it to reload the process and its configuration by sending a `SIGHUP` signal.

```
$ docker kill --signal=HUP <container>  
$ docker exec <container> kill -SIGHUP 1
```



Kill the container

Use the docker kill command to terminate the container.
The stop and kill commands can take multiple container IDs
in parameters.

```
● ● ● bash
$ docker run --name clock -d lukaszlach/clock
009ffca7b10ea401b10e3161959f992974e37bec35ca0184d29c1c6c7c...
$ docker kill clock
clock
docker kill clock 0.03s user 0.02s system 30% cpu 0.152 total
```



Custom signal

Use the `docker kill` command to manually send the HUP signal to the container process running under PID 1.

```
● ● ● bash  
$ docker run --name nginx -d nginx  
7b10ea401b10e3161959f9ffcaca0184d29c1c6c7c92974e37bec35009...  
$ docker kill --signal HUP nginx  
nginx
```



Ignored signal

Run a sleep command in a container. It does not respond to the SIGTERM signal, so a 10 second delay is in effect.

```
bash
$ docker run -d --name sleep alpine:3 \
    sleep 1d

$ docker kill --signal TERM sleep
sleep
$ time docker kill sleep
sleep
docker stop sleep 0.03s user 0.02s system 0% cpu 0.215 total
```



Kill all containers

Use a docker rm command to terminate and remove all remaining containers with a single command.

```
bash | Training

$ docker rm -f \
$(docker ps -f ancestor=lukaszlach/clock -q)
6e494a61169d

$ docker kill $(docker ps -q)
dd1c733721bf
0d9cc89dale3
```



One technology giving hundreds of different technologies and tens of programming languages



Restarting and attaching to a container



Run a container

An interactive mode (-it parameters) is a must to be able to detach from a container and attach back later.

```
bash  
$ docker run -it --name alpine alpine:3  
/ #
```



Detach from a container

Container can be detached from (and later attached back), leaving it running in the background.

```
bash

# Detach from the running container (Ctrl+P+Q)
/ # ^P^Q
$

# Attach back to the same shell session
$ docker attach alpine
/ #
```



Commands combined

docker run

 docker create →

 docker start →

 docker attach (if not running in the daemon mode with -d)



Commands combined

```
$ docker run -it --name alpine alpine:3
```

```
● ● ● bash  
$ docker create -it --name alpine alpine:3  
aa361f978b4296918801e0f2d15dc2a89ef795bd84a2946677677f28cbae  
$ docker start alpine  
alpine  
$ docker attach alpine  
/ #
```



Extend the container

Create some state in the container,
install additional packages with the apk package manager.

```
bash
/ # apk update
/ # apk add figlet
/ # exit
$
```



Respawn the container

restart the container and attach to the stdin of the main process.

● ● ● bash

```
$ docker restart alpine  
alpine
```

```
$ docker attach alpine  
/ # figlet Works
```

```
--\ \ / / -- _ - _ | | _  
\\ \\ / / / _ \ | ' _ | | / / _ |  
\\ v v / ( ) | | | | < \ _ \ \  
\\ / / \ _ / | _ | | _ | \ _ \ _ /
```



Container startup

Remember to "use the force" (-f parameter) when calling commands that fail when called twice in a row (`mkdir`, `cp`, `mv`, `ln`, ...).

```
bash

$ docker run -it --name alpine alpine:3 \
    mkdir /test

$ docker restart alpine
alpine
$ docker inspect -f '{{.State.ExitCode}}' alpine
1
$ docker logs alpine
mkdir: can't create directory '/test': File exists
```



Executing commands inside the container and obtaining shell access



The docker exec command runs
a new process inside a running container.



Run a command in a running container

\$ docker exec	Run a new process in a running container.
\$ docker exec -it	Bind to the container's standard input and allocate a pseudo-TTY (terminal).
\$ docker exec -u	Run as specified username or UID.

Enter the container shell as the root user:

```
$ docker exec -it -u 0 <container> sh
```



Run a command in the container

Use the docker exec command to run any command available inside the selected container.

```
bash  
$ docker run -d --name clock lukaszlach/clock  
$ docker exec clock ps aux  
PID   USER      TIME  COMMAND  
    1  root      0:00  sh -c while true; do date; sleep 1; done  
   21  root      0:00  sh  
   68  root      0:00  sleep 1  
   69  root      0:00  ps aux
```



Enter the container shell

Use the docker exec command to access the shell of the container. This creates a new process running /bin/sh inside the container.

```
bash

$ docker run -d --name clock lukaszlach/clock
$ docker exec -it clock sh

/ # ps aux
PID  USER      TIME  COMMAND
  1 root      0:00  sh -c while true; do date; sleep 1; done
 21 root      0:00  sh
 68 root      0:00  sleep 1
 69 root      0:00  ps aux
```



Chaining commands

You can chain several commands to execute them all at once, to do this you need to wrap them with shell.

```
bash

$ docker run -d --name clock lukaszlach/clock
$ docker exec -it clock sh -c "ls && ps"
bin    dev    etc    home   proc   root   sys    tmp    usr    var

PID   USER      TIME  COMMAND
      1 root      0:00  sh -c while true; do date; sleep 1; done
      21 root     0:00  sleep 1
      22 root     0:00  ps
```



The nsenter command is available by default on most of Linux distributions and allows to enter any container filesystem, network and IPC namespaces.



Enter the namespace

The only thing you need is a PID of **any** process running inside the chosen container from the host system perspective.

```
bash  
$ sudo su  
# ps aux | grep apache  
root      13896  0.0  0.4 351588 32572 ? Ss 2020 8:39 apache2 -DFOREGROUND  
www-data  14141  0.0  0.4 351588 32572 ? Ss 2020 8:39 apache2 -DFOREGROUND  
www-data  14154  0.0  0.4 351588 32572 ? Ss 2020 8:39 apache2 -DFOREGROUND  
...  
# nsenter -m -u -p -n -p -t 13896 bash  
root@a21ffa3b8dc5:/#
```



Container namespaces



bash

```
$ export PID=$(docker inspect -f '{{.State.Pid}}' clock)
```

```
$ nsenter -h
```

Usage:

```
nsenter [options] <program> [<argument>...]
```

Run a program with namespaces of other processes.

Options:

-t, --target <pid>	target process to get namespaces from
-m, --mount[=<file>]	enter mount namespace
-u, --uts[=<file>]	enter UTS namespace (hostname etc)
-i, --ipc[=<file>]	enter System V IPC namespace
-n, --net[=<file>]	enter network namespace
-p, --pid[=<file>]	enter pid namespace
-C, --cgroup[=<file>]	enter cgroup namespace
-U, --user[=<file>]	enter user namespace
-S, --setuid <uid>	set uid in entered namespace



Container namespaces

```
bash

# Network (with host tools)
$ sudo nsenter -t $PID -n tcpdump

# Process
$ sudo nsenter -t $PID -m ps aux

# Mount (only container)
$ sudo nsenter -t $PID -m ls /
# Mount (both container and host filesystems and host tools)
$ sudo ls -la /proc/${PID}/root/

# Shell access
$ sudo nsenter -t $PID -n -p -m -u -i sh
```



cntr <https://github.com/Mic92/cntr>

Use the cntr tool for ultimate access to a container.

Access network and processes, both container (/var/lib/cntr) and host (/) filesystems and all the tools installed on the host system.

```
● ● ● bash  
$ ngrep  
-bash: ngrep: command not found  
$ apt-get update  
$ apt-get install -y ngrep  
$ sudo cntr attach orca  
orca@20bac2ffc478:/var/lib/cntr$
```



Introduction to building own container image



Build own container image

Interactive mode

```
$ docker run -it  
$ docker commit
```

Automated mode

```
$ docker build
```



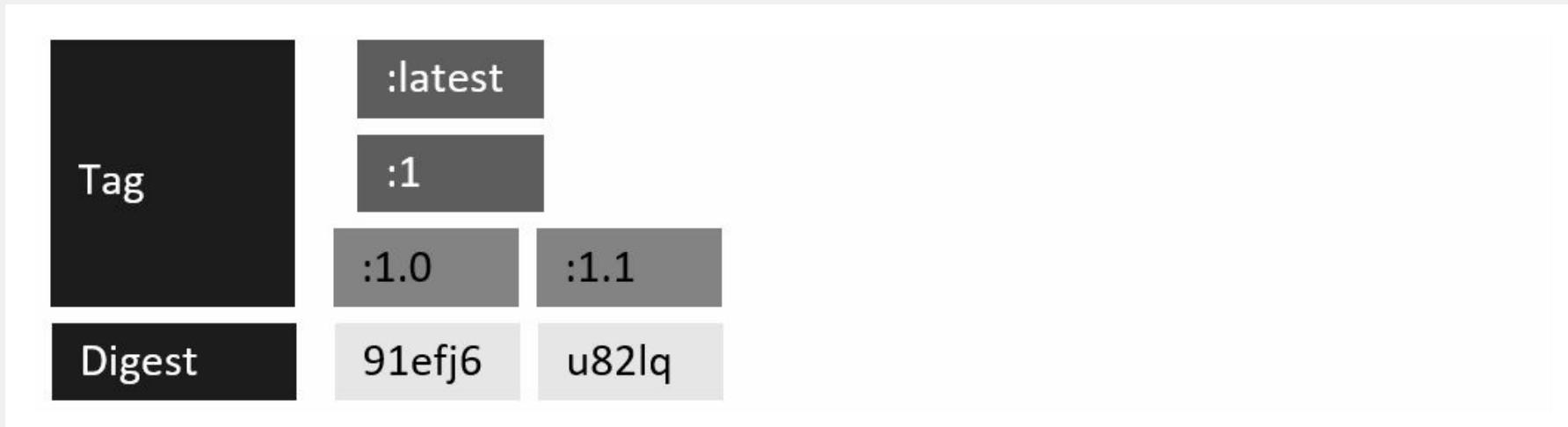
Versioning strategies

Stable tags

Unique tags



Stable tagging



Source: [Microsoft](#)



Stable tags

`:1` a stable tag for the major version. 1 means the "newest" or "latest" `1.*` version.

`:1.0` a stable tag for version `1.0`, allowing to bind to this minor version and not be rolled forward to `1.1`. Note this tag can also point `1.0.2` version.

`:latest` points to the latest stable tag, no matter what the current major version is.



Unique tagging

Tag	:12204
Digest	91efj6

Source: [Microsoft](#)



Unique tags

git commit

```
$ docker tag image image:$(git rev-parse --short HEAD)
```

date-time-stamp

```
$ docker tag image image:$(date +%y%m%d-%H%M)
```

build ID – incremental, gives you correlation back to the information in your build and CI/CD systems, including all the artifacts and logs

```
$ docker tag image image:$CI_BUILD_ID
```



Good practices

Use stable tagging for base images.

Do not deploy with stable tags as it leads to instability.



Good practices

Use unique tags for deployments.

You should stick to an exact version when deploying to production,
so hosts would not accidentally pull a newer version
causing inconsistency with the other nodes.



Building Docker images in an interactive mode



Run an interactive container

Run the alpine-figlet container from the alpine:3 image and alter it however it is needed.

```
bash  
$ docker run -it --name alpine-figlet alpine:3  
/ # apk update  
/ # apk add figlet  
/ # figlet Works  
/ # exit
```



The docker commit command saves all the changes made to the container into a new layer and creates a new image.



Convert container to image

Create a new image from the changes made to the alpine-figlet container.

```
bash  
$ docker commit alpine-figlet alpine:3-figlet  
$ docker run alpine:3-figlet \  
    figlet Still works
```

```
   _ _ _ | - | _ ( ) | | | _ _  
 \_ _ \ | _ | | | | \ \ / \ / / _ \ | ' _ | | / / _ |  
   _ _ ) | | _ | | | | \ V V / ( ) | | | | < \ _ \ \ _ |  
   _ _ / \ _ | _ | _ | | \ _ / \ _ / \ _ / | _ | | \ _ \ _ /
```



The new top layer

Use the docker history command to display the image layers.

```
bash  
$ docker history alpine:3  
IMAGE          CREATED        CREATED BY  
82f67be598eb  2 months ago   /bin/sh -c #(nop) CMD ["/bin/sh"]  
<missing>      2 months ago   /bin/sh -c #(nop) ADD file:f4f85ec73d7c...  
  
$ docker history alpine:3-figlet  
IMAGE          CREATED        CREATED BY  
4ecd28323dd1  12 seconds ago /bin/sh  
82f67be598eb  2 months ago   /bin/sh -c #(nop) CMD ["/bin/sh"]  
<missing>      2 months ago   /bin/sh -c #(nop) ADD file:f4f85ec73d7c...
```



Building Docker images from a Dockerfile



Ways to create a new image

`$ docker commit`

Saves all the changes made to the container
into a new layer and creates a new image.

`$ docker build`

Runs a repeatable image build process
basing on the recipe from the Dockerfile.



The `docker build` command builds Docker images from a Dockerfile and a “context”.

A build context is the set of files located in the specified PATH or URL. This is an automated and repeatable process.



Dockerfile

A Dockerfile is a build recipe for a Docker image.

It contains step-by-step instructions telling how an image is constructed. Use the docker build command to build an image from a Dockerfile.



Build targets

No build context

```
$ docker build - < Dockerfile
```

```
$ Get-Content Dockerfile | docker build -
```

Build context stored in an archive

```
$ docker build - < context.tar.gz
```

Build GIT repository directly

```
$ docker build https://github.com/sclevine/yj.git
```

Build with locally stored build context

```
$ docker build .
```



Build the image

The `docker build` command builds Docker images from a `Dockerfile` and a “context”. A build context is the set of files located in the specified path or URL.

```
$ docker build -t tag context
```

-t tag
context

tag of the new image
location of the build context



Example Dockerfile

Create a file called `Dockerfile` in the project directory.



Dockerfile

```
FROM golang:1.12-alpine
WORKDIR /project
ENV GOPATH=/project
COPY main.go .
RUN go build .
CMD ["/project/project"]
```



Build the image

Use the `docker build` command to convert the Dockerfile to a Docker image.

```
bash
$ docker build .
Sending build context to Docker daemon 6.656kB
Step 1/6 : FROM golang:1.12-alpine
...
Successfully built 473d4281bb92
```



Build and tag the image

Use the `docker build` command to convert the Dockerfile to a Docker image.

```
● ● ● bash
$ docker build -t project .
Sending build context to Docker daemon 6.656kB
Step 1/6 : FROM golang:1.12-alpine
...
Successfully built 473d4281bb92
Successfully tagged project:latest
```



Build logs



bash

```
Sending build context to Docker daemon 758.8kB
Step 1/6 : FROM golang:1.12-alpine
--> e04879bf1b7f
Step 2/6 : WORKDIR /project
--> Running in 8fc25bef4be1
Removing intermediate container 8fc25bef4be1
--> 9161bb15c514
Step 3/6 : ENV GOPATH=/project
...
Successfully built 65fc013f7a9f
Successfully tagged project:latest
```



Build logs

```
Sending build context to Docker daemon 758.8kB
```

The build context is the working directory given to the docker build command, compressed and sent by the client to the Docker daemon.

Be patient if that directory is big or holds a lot of files.



Build logs

Step 5/6 : RUN go build .

---> Running in `0bc41d183e84`

Removing intermediate container `0bc41d183e84`

---> `879e4a5fb231`

1. A container (`0bc41d183e84`) is created for the base image which is the result of Step 4/6.
2. The RUN instruction is executed in this container.
3. The container is committed into an image (`879e4a5fb231`).
4. The container used to execute the instruction (`0bc41d183e84`) is removed.



Build logs

```
---> 65fc013f7a9f
```

```
Successfully built 65fc013f7a9f
```

```
Successfully tagged project:latest
```

1. The container is committed into an image ([65fc013f7a9f](#)).
2. This image is tagged as our result [project:latest](#) image.



	FROM golang:1.12-alpine	create an intermediate image	1 →
→ 1	WORKDIR /project	create a directory and change directory	2 →
→ 2	ENV GOPATH=/project	set the environment variable	3 →
→ 3	COPY main.go .	create a new layer with copied files	4 →
→ 4	RUN go build .	run defined command on top of new layer	5 →
→ 5	CMD ["..."]	set a default command and/or parameters	6 →
→ 6	tag the result intermediate image	project:latest	→



Rebuild the image

Build the image again using the same command.

```
bash  
$ docker build -t project .  
Sending build context to Docker daemon 6.656kB  
Step 1/6 : FROM golang:1.12-alpine  
Step 2/6 : WORKDIR /project  
    --> Using cache  
    --> 7a2a559734bb  
Step 3/6 : ENV GOPATH=/project  
    --> Using cache  
    --> 86a80294338d  
...  
...
```



Build cache

After each build step, Docker takes a snapshot of the resulting image.

Before executing a build step, Docker checks if it has already built the same sequence and therefore has it in the build cache.

You can always force a rebuild and disable the cache:

```
$ docker build --no-cache ...
```



Build cache key

Docker uses the instruction contents for a build cache key, meaning:

```
RUN apk add curl figlet
```

is different from

```
RUN apk add figlet curl
```

RUN apt-get update is neither re-executed unless forced,
whether target package mirrors are updated or not.



Build cache and build context

Docker caches steps involving COPY and ADD instructions.

Those steps **will not be executed again** if the files, recursively in all pointed directories, have not been changed.

On the other hand, when a remote resource is used the step **is never stored in cache** and it will always be executed, so all further steps as well.



Run the image

Verify the image was built correctly by running it.
You should see the output message.

```
● ● ● bash
$ docker run project
Hello World
```



Multi-stage builds



Multi-stage build

Dockerfile can be built with multiple build stages.

Use multiple FROM statements in a single Dockerfile.

Each FROM instruction can use a different base,
and each of them begins a new stage of the build.

The last FROM instruction and all steps executed further
result in the final image.



Multi-stage build benefits

- keep all the logic in a single Dockerfile
- optimized and tiny result image
- improve run-time performance
- faster deployments due to limited download size
- a standardized method of running build actions



Example Dockerfile

Create a file called `Dockerfile` in the project directory.



Dockerfile

```
FROM golang:1.12-alpine
WORKDIR /project
ENV GOPATH=/project
COPY main.go .
RUN go build .
CMD ["/project/project"]
```



Large image

The project:latest image inherited all of the golang image layers.

```
bash  
$ docker images | grep project  
REPOSITORY      TAG      IMAGE ID      CREATED      SIZE  
project          latest   5122e9f680b9   About a minute ago  348MB
```



Inherited layers



bash

```
$ docker history project
```

IMAGE	CREATED	CREATED BY	SIZE
8bb788d78609	7 hours ago	/bin/sh -c #(nop) CMD ["/project/project"]	0B
45d61990847e	7 hours ago	/bin/sh -c go build .	2MB
5739b2aababa	7 hours ago	/bin/sh -c #(nop) COPY file:1ae271c8253f218c...	73B
50ef281684e1	7 hours ago	/bin/sh -c #(nop) ENV GOPATH=/project	0B
4f3af5e445cb	7 hours ago	/bin/sh -c #(nop) WORKDIR /project	0B
76bddfb5e55e	12 months ago	/bin/sh -c #(nop) WORKDIR /go	0B
<missing>	12 months ago	/bin/sh -c mkdir -p "\$GOPATH/src" "\$GOPATH/b..."	0B
<missing>	12 months ago	/bin/sh -c #(nop) ENV PATH=/go/bin:/usr/loc...	0B
<missing>	12 months ago	/bin/sh -c #(nop) ENV GOPATH=/go	0B
<missing>	12 months ago	/bin/sh -c set -eux; apk add --no-cache --v...	340MB
<missing>	12 months ago	/bin/sh -c #(nop) ENV GOLANG_VERSION=1.12.17	0B
<missing>	13 months ago	/bin/sh -c [! -e /etc/nsswitch.conf] && ec...	17B
<missing>	13 months ago	/bin/sh -c apk add --no-cache ca-certifica...	553kB
<missing>	13 months ago	/bin/sh -c #(nop) CMD ["/bin/sh"]	0B
<missing>	13 months ago	/bin/sh -c #(nop) ADD file:e69d441d729412d24...	5.59MB



COPY --from

In a multi-stage build artifacts can be selectively copied between stages, leaving behind everything not needed in the final image.

```
COPY --from=0 /binary .
```

```
COPY --from=stage-name *.sh /bin/
```

You can also point a separate image, either using the local image name, a tag available locally or on a Docker registry.

```
COPY --from=nginx:latest /etc/nginx/nginx.conf /nginx.conf
```



Modified Dockerfile

Introduce multi-stage builds in the Dockerfile.



Dockerfile

```
FROM golang:1.12-alpine
WORKDIR /project
ENV GOPATH=/project
COPY main.go .
RUN go build .
CMD ["/project/project"]

FROM alpine:3
COPY --from=0 /project/project /project
CMD ["/project"]
```



Name your build stages

Introduce multi-stage builds in the Dockerfile.



Dockerfile

```
FROM golang:1.12-alpine AS build
WORKDIR /project
ENV GOPATH=/project
COPY main.go .
RUN go build .
CMD ["/project/project"]

FROM alpine:3
COPY --from=build /project/project /project
CMD ["/project"]
```



Optimized Dockerfile

Remove redundant instructions from the Dockerfile.

```
Dockerfile

FROM golang:1.12-alpine AS build
WORKDIR /project
ENV GOPATH=/project
COPY main.go .
RUN GOPATH=/project go build .
CMD ["/project/project"]

FROM alpine:3
COPY --from=build /project/project /project
CMD ["/project"]
```



Stop at a specific build stage

When you build your image, you do not necessarily need to build the entire Dockerfile including every stage.
You can specify a target build stage.

```
$ docker build --target <stage_name> .
```



Dockerfile CI

One Dockerfile is needed to lint, analyze, install dependencies, test and containerize your project using a single docker build command.

This example PHP project Docker image is built only when all previous stages pass successfully.

```
FROM scratch AS project
COPY . /build

FROM jakzal/phpqa:php7.2 AS qa
COPY --from=project /build /build
WORKDIR /build
RUN phplint . && \
phpa .

FROM composer AS project-vendor
COPY --from=project /build /build
WORKDIR /build
RUN composer install

FROM phpunit/phpunit AS test
COPY --from=project-vendor /build /build
WORKDIR /build
RUN phpunit test/

FROM php:7.2-fpm-alpine AS release
COPY --from=project-vendor /build /var/www

FROM release AS release-dev
RUN apk add --no-cache --update --virtual buildDeps \
    autoconf g++ make && \
    pecl install xdebug && \
    docker-php-ext-enable xdebug && \
    apk del buildDeps

FROM release
```



Dockerfile CI

Run only the stages required for tests execution

```
$ docker build --target=test .
```

Build a developer version of the image

```
$ docker build --target=release-dev -t project:dev .
```

Build the production image

```
$ docker build -t project:latest .
```



Difference between Docker images and containers



Image

An **image** is read-only.

A Docker image is built up from a series of layers.

Each layer is only a set of differences from the layer before it.

The layers are stacked on top of each other.



Container

A **container** is an instantized image, in a read-write copy of its filesystem. Copy-on-write strategy is used instead of regular copy to improve disk I/O performance.

There are two ways to create a container out of an image:

```
$ docker create      creates a container but does not run it  
$ docker run        starts a container from a given image
```



Container and image

The difference between a container and an image is the top writable layer. All writes to the container that add new or modify existing data are stored in this writable layer.

Because each container has its own writable container layer, multiple containers can share access to the same underlying image and yet have their own data state. The underlying image remains unchanged.



Container filesystem

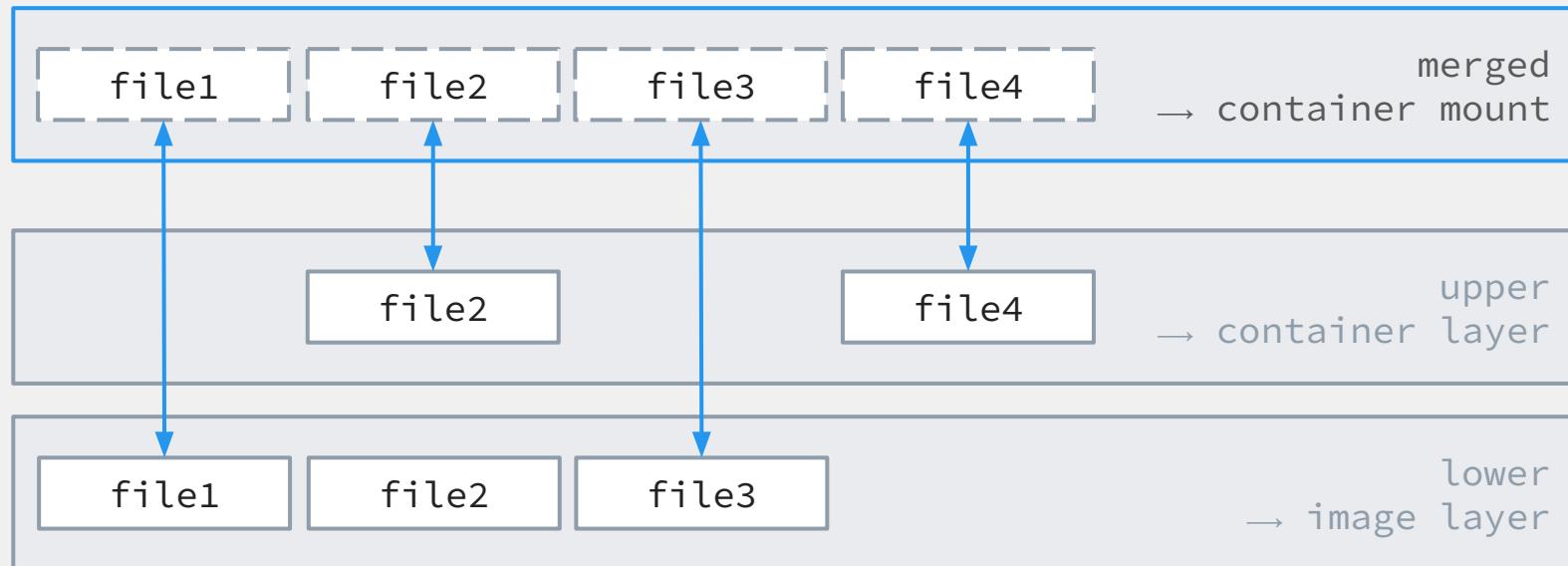
OverlayFS combines two filesystems for working with containers - "upper" filesystem and "lower" filesystem.

When a name exists in both filesystems, the object in the "upper" filesystem is visible while the object in the "lower" filesystem is either hidden or, in case of directories, merged with the "upper" object.

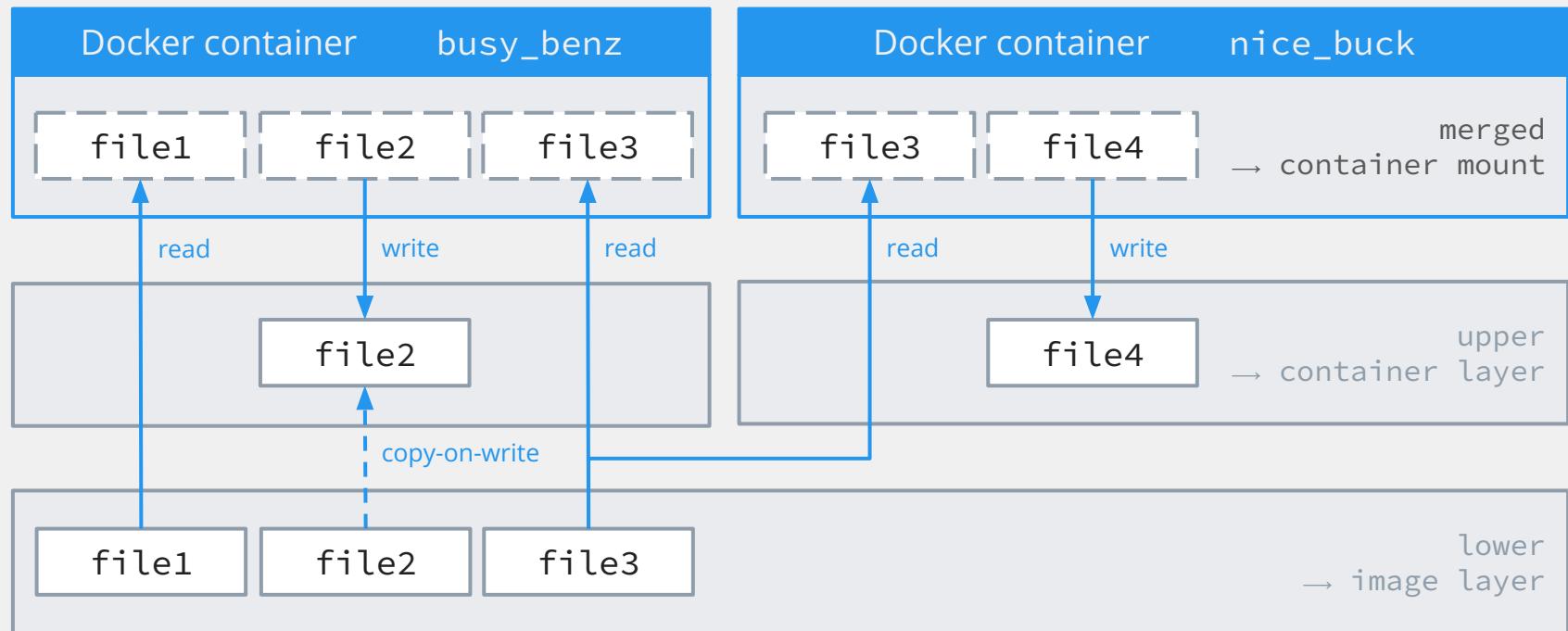
The unified view is exposed through its own directory called "merged".



Container filesystem



Container filesystem



Browse container filesystem

Browse the container filesystem directly by accessing the "merged" directory which gives a look at the root filesystem of a container being a combination of all layers.

```
● ● ● bash | State
$ docker inspect \
  --format '{{json .GraphDriver.Data}}' container
{
  "LowerDir": "/var/lib/docker/overlay2/0dad272f8c6706fb...",
  "MergedDir": "/var/lib/docker/overlay2/0dad4d523351851...",
  "UpperDir": "/var/lib/docker/overlay2/0dad4d523351851a...",
  "WorkDir": "/var/lib/docker/overlay2/0dad4d5233518c2a0..."
}
```



Accessing host files when building an image



COPY

The COPY instruction copies files or directories from <src> and adds them to the filesystem of the container at the path <dest>.

Multiple <src> resources may be specified,
also each <src> may contain wildcards.

COPY <src>[, <src>, ...] <dest>

COPY index.html /var/www

COPY *.png /var/www/static



ADD

The ADD instruction copies new files, directories or **remote file URLs** from <src> and adds them to the filesystem of the image at <dest>.

If <src> is **a local archive** in a recognized compression format (identity, gzip, bzip2 or xz) then it is unpacked as a directory.

```
ADD <src>[ , <src>, ...] <dest>
```

```
ADD *.png /var/www
```

```
ADD archive.tar /
```



.dockerignore

If a file named `.dockerignore` exists in the root directory of the context, Docker client modifies the context to exclude files and directories that match patterns stored in it.

This helps to avoid unnecessarily sending large or sensitive files and directories to the build process and potentially adding them to images using ADD or COPY.



.dockerignore syntax

The pattern matching syntax is based on Go `filepath.Match()` function and includes some additions in comparison to `.gitignore` file.

- * matches any sequence of non-separator characters
- ? matches any single non-separator character
- ** matches any number of directories (including zero)
- ! can be used to make exceptions to exclusions
- # lines starting with this character are ignored, use it for comments



List strategies

Whitelisting files

Call the COPY instruction only on selected files and directories.

This can become hard to maintain and sometimes requires several COPY instructions which create separate layers.

Greylisting files

Create a `.dockerignore` and use `COPY . . .` in your Dockerfile.

If the ignore file stops being maintained the built image size may grow and expose unexpected files.



Dockerfile structure and available instructions



Dockerfile structure



Dockerfile

```
# Comment  
INSTRUCTION arguments  
INSTRUCTION arguments more-arguments  
INSTRUCTION arguments other-argument
```



Notice on syntax

Some instructions can take JSON or string syntax.

The string syntax is executed through sh -c "...." by default, for CMD it means it will be started as a subcommand of /bin/sh, which does not pass signals. For the RUN instruction this behavior is desirable.

RUN apt-get update

String syntax specifies a command to be wrapped within /bin/sh -c "...."

CMD ["/usr/bin/daemon"]

JSON syntax specifies an exact command to execute.



FROM

The FROM instruction initializes a new build stage and sets the base image for subsequent instructions.

As such, a valid Dockerfile must start with a FROM instruction.

```
FROM [--platform=<platform>] <image>[:<tag>] [AS <name>]
```

```
FROM [--platform=<platform>] <image>[@<digest>] [AS <name>]
```

```
FROM nginx:1.19
```

```
FROM nginx@sha256:e71b1bf426bc9a277eadc1...e16c1c
```



RUN

The RUN instruction will execute any commands in a new layer on top of the current image and commit the results. The resulting committed image will be used for the next step in the Dockerfile.

```
RUN <command>
```

```
RUN ["executable", "param1", "param2"]
```

```
RUN apt-get update && apt-get install -y curl
```

```
RUN ["/bin/bash", "-c", "echo Hello"]
```



CMD

The main purpose of a CMD is to provide defaults for an executing container. These defaults can include an executable, or they can omit it, in which case you must specify an ENTRYPOINT instruction as well.

```
CMD ["executable", "param1", "param2"]
```

```
CMD ["param1", "param2"] # requires an ENTRYPOINT as well
```

```
CMD command param1 param2
```

```
CMD nginx -g 'daemon off'
```

```
CMD ["nginx", "-g", "daemon off"]
```



EXPOSE

The EXPOSE instruction informs Docker that the container listens on the specified network ports at runtime.

You can specify whether the port listens on TCP or UDP.

EXPOSE <port>[/<protocol>]

EXPOSE 80

EXPOSE 3030/udp



ENV

The ENV instruction sets the environment variable <key> to the value <value>. This value will be in the environment for all subsequent instructions and visible in a running container.

```
ENV <key>=<value> ...
```

```
ENV NAME="John Doe"
```

```
ENV ENVIRONMENT=staging
```

```
ENV FIRST=1st SECOND=2nd
```



Environment variables in RUN

If an environment variable is only needed during build, and not in the final image, consider setting a value for a single command instead. The same applies if the variable is needed only by the RUN instruction.

```
RUN DEBIAN_FRONTEND=noninteractive \
    apt-get update && apt-get install -y ...
```

```
RUN VARIABLE=value env
```

```
RUN export VARIABLE=value && env
```



COPY

The COPY instruction copies files or directories from <src> and adds them to the filesystem of the container at the path <dest>.

Multiple <src> resources may be specified,
also each <src> may contain wildcards.

COPY <src>[, <src>, ...] <dest>

COPY index.html /var/www

COPY *.png /var/www/static



COPY --chown

COPY creates all new files and directories with a UID and GID of 0 (root), unless the optional --chown flag specifies a given username, group name, or UID/GID combination to request specific ownership of the copied content.

```
COPY --chown=www-data:www-data . /var/www
```

```
COPY --chown=101 index* /srv/
```



ENTRYPOINT

An ENTRYPOINT allows you to configure a container that will run as an executable. Command line arguments to docker run <image> will be appended after all elements of the ENTRYPOINT, and will override all elements specified using CMD.

```
ENTRYPOINT ["executable", "param1", "param2"]
```

```
ENTRYPOINT command param1 param2
```

```
ENTRYPOINT ["nginx", "-g", "daemon off"]
```



Entrypoint

ENTRYPOINT

```
["/entrypoint.sh"]
```

CMD

```
["some", "default", "command"]
```

```
$ docker run <image>
```

```
/entrypoint.sh some default command
```



VOLUME

The VOLUME instruction creates a mount point with the specified name and marks it as holding externally mounted volumes from native host or other containers.

```
VOLUME ["/dir", "/path"]  
VOLUME /dir /path
```

```
VOLUME /var/www  
VOLUME /srv/project /etc/project
```



USER

The USER instruction sets the user name (or UID) and optionally the user group (or GID) to use when running the image and for any RUN, CMD and ENTRYPOINT instructions that follow it in the Dockerfile.

USER <user>[:<group>]

USER <UID>[:<GID>]

USER www-data:www-data

USER root

USER 101:101

USER 999



WORKDIR

The WORKDIR instruction sets the working directory for any RUN, CMD, ENTRYPOINT, COPY and ADD instructions that follow it in the Dockerfile.

If the WORKDIR doesn't exist, it will be created even if it is not used in any subsequent Dockerfile instruction.

WORKDIR /path/to/workdir



ARG

The ARG instruction defines a variable that users can pass at build-time to the builder with the docker build command using the --build-arg <varname>=<value> flag.

```
FROM alpine:3
ARG VERSION
RUN apk add nginx~$VERSION
$ docker build --build-arg VERSION=1.18 .
```



Instructions behaviour

Creates an image layer:

RUN, COPY, ADD

Runs in an intermediate container:

all except **COPY** and **ADD**



Build Docker images with BuildKit and BuildX



Dockerfile best practices



Dockerfile best practices

Collapsing layers

Squashing the image

Multi-stage builds

Removing
unnecessary files

Order of instructions

Understand
incremental layers



Collapsing layers

RUN instructions are executed through sh -c, one of the benefits of this is that several **commands can be chained together**.

```
FROM alpine:3.9
RUN apk --no-cache add su-exec
RUN mkdir -p /var/lib/orca
```

readable | creates 2 layers

```
FROM alpine:3.9
RUN apk --no-cache add su-exec && \
    mkdir -p /var/lib/orca
```

expensive layer | creates 1 layer



Squashing the image

After the build process is done, **the image is reduced into a single layer**. So in the end there are two of them - the FROM layer and the squashed layer.

```
● ● ● bash
$ docker build --squash -t orca .
$ docker history orca
IMAGE          ... SIZE      COMMENT
f3d87b6668d7    15.2MB    merge sha256:a3e722... to sha256:cdf98d...
```



Removing unnecessary files

Do not allow package manager cache, compiler cache or any other temporary files to be stored in a layer.

```
FROM debian:stretch
RUN apt-get update && \
    apt-get install -y curl && \
    rm -rf /var/lib/apt/lists/*
```

```
FROM alpine:3.9
RUN apk update && \
    apk add curl && \
    rm -rf /var/cache/apk/*
```

```
# syntax=docker/dockerfile:1.2
FROM ubuntu:disco
RUN --mount=type=cache,target=/var/lib/apt \
    apt update && apt install -y curl
```



Explore all of the Dockerfile 1.2
and experimental instructions under
<https://lach.dev/dockerfile-1.2>



Side challenge

`alpine:3` is a 6 MB image, `10mb` is a file of this exact size.
What are the sizes of images built with below Dockerfiles?

```
FROM alpine:3  
COPY 10mb .
```

16 MB

```
FROM alpine:3  
COPY 10mb .  
RUN rm -f 10mb
```

16 MB

```
FROM alpine:3  
RUN dd if=/dev/zero of=10mb bs=1024 count=10240  
RUN rm -f 10mb
```

16 MB



Order of instructions

1. Place static instructions higher in the order - **EXPOSE, VOLUME, CMD, ENTRYPOINT, and WORKDIR** - whose value does not change once it is set.
2. Place dynamic instruction lower in the order.
Instructions like **ENV** (when using variable substitution), **ARG**.
3. Place dependency **RUN** instructions before ADD or COPY.
4. Place **ADD** and **COPY** instructions as close as possible to the end of file.



Incremental layers

Docker image layers work similarly compared to git commits. Any new Docker image layer created with an instruction in the Dockerfile increases the size of the Docker image, even if that instruction removes files from the filesystem.

```
FROM alpine:3.9
RUN apk update && \
    apk add curl
...
RUN apk add wget && \
    rm -rf /var/cache/apk/*
```

The APK cache still exists in one of the layers

```
FROM alpine:3.9
RUN apk update && \
    apk add curl && \
    rm -rf /var/cache/apk/*
...
RUN apk update && \
    apk add wget && \
    rm -rf /var/cache/apk/*
```

No APK cache is stored in any of the layers



Incremental layers

The same applies when copying and moving files between image filesystem.

```
FROM alpine:3.9  
  
COPY . /build  
RUN mv /build /project
```

```
FROM alpine:3.9  
COPY . /build  
RUN cp -R /build /project && \  
    rm -rf /build
```

Project directory exists in two layers

```
FROM alpine:3.9  
COPY . /project
```

```
FROM alpine:3.9  
COPY . /build  
RUN ln -sf /build /project
```

Only one layer holds the project directory



Example

```
FROM alpine

COPY . /project
RUN apk update
RUN apk add nginx
ENV VARIABLE=value
EXPOSE 80
COPY ./www /usr/share/nginx/html
RUN rm -rf /var/cache/apk/*
CMD nginx -g daemon off;
```

Before (6 layers)

```
FROM alpine:3.13

EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
ENV VARIABLE=value

RUN apk update && \
    apk add nginx && \
    rm -rf /var/cache/apk/* && \
    ln -sf /project/www \
        /usr/share/nginx/html

COPY project/ /project
```

After (3 layers)



Dockerfile linter

Build best practice Docker images with a help of Dockerfile linter.
RUN instructions are checked using ShellCheck.



bash

```
$ docker run -i hadolint/hadolint < Dockerfile
/dev/stdin:7 DL3018 warning: Pin versions in apk add. Instead
of `apk add <package>` use `apk add <package>=<version>`
/dev/stdin:7 DL3019 info: Use the `--no-cache` switch to
avoid the need to use `--update` and remove
`/var/cache/apk/*` when done installing packages
```



Final example



Dockerfile

```
FROM alpine:3.13

EXPOSE 80

CMD ["nginx", "-g", "daemon off;"]

ENV VARIABLE=value

RUN apk --no-cache add nginx=1.18.0-r13 && \
    ln -sf /project/www \
        /usr/share/nginx/html

COPY project/ /project
```



Process takeover

When a shell script aims to pass a control to another command or process it calls the exec command.

When you exec a command, it replaces the shell entirely, no new process is forked, no new PID is created, and all memory controlled by the shell is destroyed and overwritten.

The new process is therefore responsible for signal handling.

```
exec command [arguments ...]
```



Docker Volumes



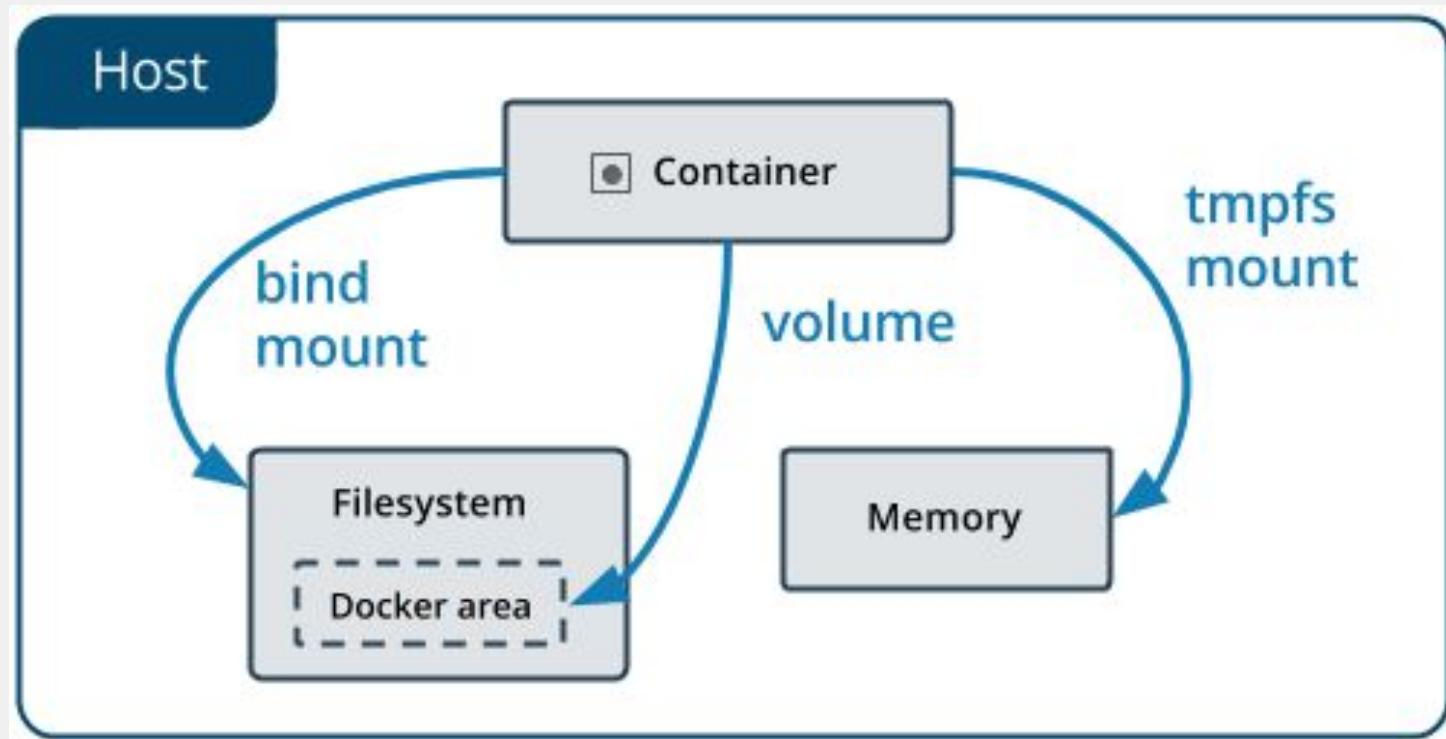
Use cases

- Sharing configuration files from the host machine to containers.
- Sharing source code or build artifacts between a development environment on the Docker host and a container.
- Sharing data among multiple running containers.
- When you want to store your container's data on a remote host or a cloud provider, rather than locally (Docker Volume drivers).



Shared filesystem in practice





<https://docs.docker.com/storage>



Mount types

Volumes

Created and managed by Docker. You can create a volume explicitly using the `docker volume create` command, or Docker can create a volume during container or service creation.

```
$ docker volume create custom-volume
$ docker run -v custom-volume:/var/www ...
```



Docker Volumes

Anonymous volume is created implicitly if an image defines it.

```
bash  
$ docker run -d --name mysql \  
-e MYSQL_ROOT_PASSWORD=root \  
mysql:5.7  
$ docker inspect -f '{{ .Config.Volumes }}' mysql  
$ docker inspect -f '{{ json .Mounts }}' mysql
```



Docker Volumes

Create a named volume with a `docker volume create` command and bind it to a container.

```
bash
$ docker volume create mysql-data
$ docker run -d \
    -v mysql-data:/var/lib/mysql \
    -e MYSQL_ROOT_PASSWORD=root \
    mysql:5.7
```



Docker Volumes

Volume driver named `local` stores all the data in `/var/lib/docker`.

```
bash
$ docker volume inspect mysql-data
[
  {
    "CreatedAt": "2021-03-21T12:45:42Z",
    "Driver": "local",
    "Labels": {},
    "Mountpoint": "/var/lib/docker/volumes/mysql-data/_data",
    "Name": "mysql-data",
    "Options": {},
    "Scope": "local"
  }
]
```



Mount types

Bind mounts

Available since the early days of Docker. When you use a bind mount, a file or directory on the host machine is mounted into a container.

```
$ docker run -v /home/ll/project:/var/www ...
$ docker run \
  --mount type=bind,source=/home/ll/project,target=/var/www ...
$ docker run -v /project/index.html:/var/www/index.html ...
```



Docker Volumes

Use an absolute host path and bind it to a container.

```
bash
$ mkdir website/
$ echo "Hello Docker" > website/index.html
$ docker run -d -p 80:80 \
             -v $(pwd)/website:/usr/share/nginx/html \
             nginx:1.19
```



Mount types

tmpfs mounts

A tmpfs mount is not persisted on disk. It can be used by a container during the lifetime of the container, to store non-persistent state or sensitive information.

```
$ docker run --mount type=tmpfs,destination=/app ...
$ docker run --tmpfs /app ...
$ docker run --mount type=tmpfs,destination=/app,tmpfs-size=1024
```



Docker Networks



Docker Network internals



Docker Engine appends iptables rules for forwarding and isolation. For this, it uses a filter chain named DOCKER.



Docker installs two custom iptables chains named DOCKER-USER and DOCKER, and it ensures that incoming packets are always checked by these two chains first.

All of Docker's iptables rules are added to the DOCKER chain. Do not manipulate this chain manually.

If you need to add rules, add them to the DOCKER-USER chain. These rules are applied before any rules Docker creates automatically.



Networking internals

Use netstat and iptables system tools to display container traffic rules.

```
bash  
$ docker run -d -p 8081:80 nginx:1.19  
  
$ netstat -an | grep 8081  
tcp        0      0  0.0.0.0:8081          0.0.0.0:*          LISTEN  
  
$ iptables-save | grep 8081  
-A DOCKER ! -i docker0 -p tcp -m tcp --dport 8081  
-j DNAT --to-destination 172.17.0.3:80
```



Networking internals

Use netstat and iptables system tools to display container traffic rules.

```
bash  
$ docker run -d -p 127.0.0.1:8082:80 nginx:1.19  
  
$ netstat -an | grep 8082  
tcp        0      0  127.0.0.1:8082          0.0.0.0:*          LISTEN  
  
$ iptables-save | grep 8082  
-A DOCKER -d 127.0.0.1/32 ! -i docker0 -p tcp -m tcp --dport 8082  
-j DNAT --to-destination 172.17.0.4:80
```



Control Docker network access

By default, all external source IPs are allowed to connect to the Docker host. To allow only a specific IP or network to access the containers, insert a negated rule at the top of the DOCKER-USER filter chain.

```
# allow all source IPs except 192.168.1.1
$ iptables -I DOCKER-USER -i ext_if ! -s 192.168.1.1 -j DROP

# allow access only from the subnet 192.168.1.0/24
$ iptables -I DOCKER-USER -i ext_if ! -s 192.168.1.0/24 -j DROP
```



Docker network drivers



Docker Networks

Docker's networking subsystem is pluggable, using drivers.

Several drivers exist by default, and provide core networking functionality:

- bridge (default)
- none
- host
- container
- overlay



List available network drivers

List available network drivers using docker info command.



bash

```
$ docker info | grep Network  
Network: bridge host ipvlan macvlan null overlay
```



List available networks

List available networks using docker network ls command.

```
bash  
$ docker network ls  
NETWORK ID      NAME      DRIVER      SCOPE  
a06fce24a24    bridge    bridge      local  
c8335fd39cee    host      host      local  
72aea92dae69    none      null      local  
  
$ docker network inspect bridge
```

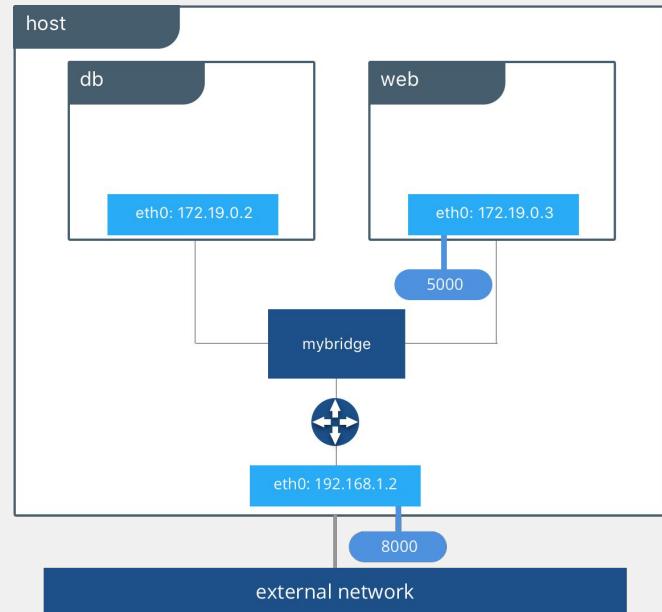


Docker Network - bridge

The bridge driver creates a private network internal to the host so containers on this network can communicate using container names * as hostnames. External access is granted by exposing ports to containers.

Docker Engine creates the necessary Linux bridges, internal interfaces, iptables rules, and host routes to make this connectivity possible.

```
$ docker run --net bridge
```



Docker Network - none

For this network mode the container gets the `lo` loopback interface and no `eth0` or other network interface is present.

As it is not able to send nor receive network traffic, this mode effectively isolates the container completely from the network.

```
$ docker run --net none
```



Docker Network - host

In this network mode the container is able to reach and access all the network interfaces of the host. Network traffic does not go through NAT or bridge, giving performance.

It can bind to any address and any port and reach services on host that are bind to 127.0.0.1 only.

```
$ docker run --net host
```



Container network mode

In this mode, the newly created container will share the same network namespace with an existing container. New container will not create its own NIC and allocate new IP, it shares IP address, hostname and network interfaces with the existing container.

```
docker run \
--net container:<container_id|container_name>
```



Netshoot

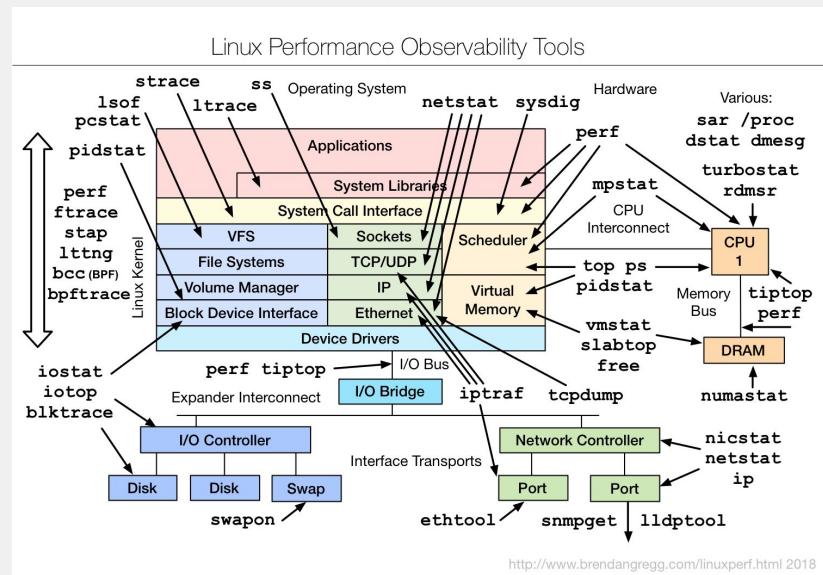
The netshoot is Docker and Kubernetes network troubleshooting swiss-army container that contains a set of powerful networking tools that can be used to troubleshoot Docker networking issues.

Container Network Namespace

```
$ docker run -it \
--net container:<id|name> \
nicolaka/netshoot
```

Host Network Namespace

```
$ docker run -it --net host \
nicolaka/netshoot
```



<http://www.brendangregg.com/linuxperf.html> 2018



Monitor shared network stack

Run the network monitoring tool in the other container to observe the traffic from the application container.

```
bash  
$ docker run -it --net container:application cmd.cat/ngrep \  
    ngrep -d eth0 -x  
interface: eth0 (172.24.0.0/255.255.0.0)  
T 172.19.0.1:56742 -> 172.19.0.3:8080 [AP]  
  50 4f 53 54 20 2f 20 48      54 54 50 2f 31 2e 31 0d      POST / HTTP/1.1.  
  0a 48 6f 73 74 3a 20 30      3a 38 30 38 30 0d 0a 55      .Host: 0:8080..U  
  73 65 72 2d 41 67 65 6e      74 3a 20 63 75 72 6c 2f      ser-Agent: curl/  
  37 2e 35 30 2e 30 0d 0a      41 63 63 65 70 74 3a 20      7.50.0..Accept:  
...
```



Enabling connectivity between containers



Remember that the default bridge network named bridge **does not** provide service-discovery, although containers still see each other using IP addresses.



Run a container

Run the server container in the `custom-network` network.

```
bash  
$ docker network create custom-network  
$ docker run -d \  
  --name server --net custom-network \  
  nginx:1.19
```



Inspect network environment

Use docker inspect to get containers connected to the selected network and the other way round.

```
bash
$ docker container inspect \
    -f '{{json .NetworkSettings.Networks}}' server
{"custom-network":{"IPAMConfig":null,"Links":null,"Aliases": [
    "d2dc032d5143"], "NetworkID":"3371930d81b1d95e7923507bcc242948
    b873220d4474c405896cc0bbaa109959", "EndpointID":"ec8f2717dd895
    80126a76c19f32b6f48e2d6d3526d32da6e01dc7febb7bebe3c", "Gateway"
    :"172.23.0.1", "IPAddress":"172.23.0.2", "IPPrefixLen":16, "IPv
    6Gateway": "", "GlobalIPv6Address": "", "GlobalIPv6PrefixLen":0, "
    MacAddress":"02:42:ac:17:00:02", "DriverOpts":null}}
```



Inspect network environment

Use docker inspect to get containers connected to the selected network and the other way round.

```
bash
$ docker network inspect \
    -f '{{json .Containers}}' custom-network
{
  "d2dc032d51436dc5895fa5064ac53cedd349c88c68c1d359e796b24b5bd
  bd397": {
    "Name": "server",
    "EndpointID": "ec8f2717dd89580126a76c1
    9f32b6f48e2d6d3526d32da6e01dc7febb7bebe3c",
    "MacAddress": "02:4
    2:ac:17:00:02",
    "IPv4Address": "172.23.0.2/16",
    "IPv6Address": ""
  }
}
```



Run a container

Run the client container in the `custom-network` network and utilize a service-discovery mechanism.

```
bash
$ docker run --rm \
  --net custom-network \
  cmd.cat/curl curl server
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
...

```



Connect and disconnect

Containers can join and part networks at runtime using `docker network connect` and `disconnect` commands.

```
bash  
$ docker network create custom-network  
$ docker run --name client -it alpine:3  
# Connect the client container to the custom-network  
$ docker network connect custom-network client  
# Disconnect the client container from the custom-network  
$ docker network disconnect custom-network client
```



Host network mode



Docker Network - host

In this network mode the container is able to reach and access all the network interfaces of the host. Network traffic does not go through NAT or bridge, giving performance.

It can bind to any address and any port and reach services on host that are bind to 127.0.0.1 only.

```
$ docker run --net host
```



Open ports directly on host

Web server opens port 80 directly on the host network interfaces making it available for curl without port publishing.

```
bash
$ docker run -d --net host nginx:1.19
$ curl localhost

$ docker run -d -p 80:80 --net host nginx:1.19
WARNING: Published ports are discarded when using host
network mode
```



Use the host network mode to troubleshoot host network interfaces using command-line (and desktop) tools running in containers.



Troubleshoot the host network

Either manually install desired tools in an interactive shell, find matching Docker image, build your own or let Commando handle it.

```
bash

# Enter container shell and install tools manually
$ docker run --net host -it alpine:3
/ # apk -U add ngrep

# Use Commando to access command-line HTTP client tools
$ docker run --net host cmd.cat/curl/wget

# Network troubleshooting tools
$ docker run --net host \
    cmd.cat/ngrep/tcpdump/ip/ifconfig/netstat
```



Docker Compose



Docker Compose

Docker Compose allows to store the whole application stack including services (containers), networks and volumes in a single YAML file.

docker-compose command maintains the container stack, automatically builds and pulls the images and allows to monitor their state, combines multiple actions in a single commands.



Command-line

```
docker pull  
docker run  
docker run -d  
docker ps →  
docker stop  
docker rm  
docker build
```

```
docker-compose pull  
docker-compose run  
docker-compose up -d  
docker-compose ps  
docker-compose stop  
docker-compose rm  
docker-compose build
```



Commands combined

`docker-compose up -d`

`docker pull` (if not exists) →

`docker stop` →

`docker build` (if not exists or forced by `--build`) →

`docker rm` (if changed or forced by `--force-recreate`) →

`docker network create` (if not exists) →

`docker volume rm` (if forced by `--renew-anon-volumes`) →

`docker volume create` (if not exists) →

`docker run` (in a foreground or in a detached mode)



Commands combined

`docker-compose down`

`docker stop` →

`docker rm` →

`docker network rm` (if not exists or forced by `--build`) →

`docker volume rm` (anonymous volumes) →

`docker volume rm` (named volumes when forced by `--volumes`) →

`docker image rm` (when forced by `--rmi`)



A native look and feel

Docker Compose project can be easily converted to a systemd service in which case this process manager manages container state, logs, utilizing the docker-compose command underneath.

```
[Unit]
Description=My Docker Project
After=docker.service
BindsTo=docker.service

[Service]
WorkingDirectory=/srv/project
ExecStartPre=/usr/local/bin/docker-compose down
ExecStart=/usr/local/bin/docker-compose up
--force-recreate
ExecStop=/usr/local/bin/docker-compose stop

[Install]
WantedBy=multi-user.target
```



Get Docker Compose

Docker Compose is a part of Docker Desktop platform on Windows and Mac but on Linux it needs a manual installation.

Follow the instructions on <https://docs.docker.com/compose/install/> to install Compose on Windows, Windows Server 2016, Mac, or Linux systems.



Docker Compose YAML file structure and syntax



The Compose file

A Docker Compose YAML file has several sections:

- version **required**, this is a Compose file version not the Docker version
- services **required**, a service is one or more replicas of the same container
- networks optional, defines the networks that containers should use
 by default a per-compose bridge network is created
- volumes optional, defines volumes to be used by the containers



The Compose file

Each service in the YAML file **must** be defined by either:

`build` points a path with the Dockerfile

`image` indicates an image name

if both are specified an image will be built from
the specified build context and then properly tagged.



Service configuration

```
build    command   container_name depends_on  
entrypoint env_file environment extra_hosts  
image     network_mode networks ports restart  
volumes
```



Project

We will base our work on a simple PHP project.

```
php  
<?php  
    echo "Hello ".$_ENV['NAME']."\n";
```



Service configuration - build

Builds the service image if needed using passed build configuration.

```
build: ./dir
```

```
build: .
```

If you specify `image` as well as `build`, then Compose names the built image with the `webapp` and optional tag specified in `image`.

```
build: ./dir
```

```
image: webapp:tag
```



Service configuration - image

Specify the image to start the container from.

Can either be a repository/tag or a partial image ID.

image: redis

image: ubuntu:18.04

image: tutum/influxdb

image: example-registry.com:4000/postgresql



Service configuration - command

Override the default command.

```
command: bundle exec thin -p 3000
```

The command can also be a list, similar to Dockerfile syntax.

```
command: ["bundle", "exec", "thin", "-p", "3000"]
```



Service configuration - container_name

Specify a custom container name,
rather than a generated default name.

container_name: my-web-container



Service configuration - depends_on

Express dependency between services.

```
services:  
  web:  
    build: .  
    depends_on: ["db", "redis"]  
  redis:  
    image: redis  
  db:  
    image: postgres
```



Service configuration - entrypoint

Override the default entrypoint.

```
entrypoint: /code/entrypoint.sh
```

The entrypoint can also be a list, similar to Dockerfile syntax.

```
entrypoint: ["php", "vendor/bin/phpunit"]
```



Service configuration - env_file

Add environment variables from a file. Can be a single value or a list.

env_file: .env

env_file:

- ./common.env
- ./apps/web.env
- /opt/runtime_opts.env



Service configuration - environment

Add environment variables.

You can use either an array or a dictionary.

environment:

RACK_ENV: development

SHOW: 'true'

environment:

- RACK_ENV=development
- SHOW=true



Service configuration - extra_hosts

Add hostname mappings.

Use the same values as the docker client --add-host parameter.

`extra_hosts:`

- "some.host.com:162.242.195.82"
- "otherhost:50.31.209.229"



Service configuration - network_mode

Network mode.

Use the same values as the docker client --network parameter.

network_mode: "host"

network_mode: "none"

network_mode: "service:[service name]"

network_mode: "container:[container name/id]"



Service configuration - networks

Networks to join.

services:

 some-service:

networks:

- some-network
- other-network



Service configuration - networks / aliases

Aliases (alternative hostnames) for this service on the network.

services:

 some-service:

networks:

 some-network:

aliases:

- my-service
- service.com



Service configuration - restart

Specify the restart policy.

restart: "no"

restart: always

restart: on-failure

restart: unless-stopped



Service configuration - volumes

Mount host paths or named volumes.

volumes:

- /var/lib/mysql
- /opt/data:/var/lib/mysql
- ./cache:/tmp/cache
- ~/configs:/etc/configs/:ro
- datavolume:/var/lib/mysql



Service configuration - ports

Publish ports.

ports:

- "8000:8000"
- "9090-9091:8080-8081"
- "127.0.0.1:8001:8001"
- "127.0.0.1:5000-5010:5000-5010"
- "6060:6060/udp"
- "12400-12500:1240"



Migrating

Convert docker run command into a docker-compose.yml file with a help of the Composerize project.

```
bash
$ docker run lukaszlach/composerize \
    docker run --name web nginx:1.19
version: 3
services:
    nginx:
        container_name: web
        image: 'nginx:1.19'
```



Create a Compose file

Create a file called docker-compose.yml.

```
docker-compose.yml

version: '3'

services:
  nginx:
    image: nginx:1.19
    container_name: web
```



Run in daemon mode

Run the image in the background using docker-compose.

```
bash  
$ docker-compose up -d  
Creating network "project_default" with the default driver  
Creating web ... done  
  
$ docker ps
```



Docker Compose defining multi container application stack



Dependent containers

Compose does not wait for containers to be “ready” before starting dependent services, but only until they have been started. There are ways some to control the startup order, although...

- DO** use it for development, research and running side tasks.
- DO NOT** use it for deployments, as it is valid only at startup time and does not behave properly when dependent containers are restarted or crashing at run time.



Service healthy

A healthcheck indicates that you want a dependency to wait for another container to be “healthy” before starting.

In the example, Compose waits for the db and api services to be healthy before starting web.

```
version: "2.4"
services:
  web:
    image: project/web
    depends_on:
      db:
        condition: service_healthy
      api:
        condition: service_healthy
  db:
    image: project/db
    healthcheck:
      test: nc -z localhost 3306
  api:
    image: project/api
    healthcheck:
      test: curl -f localhost:80
```



Adaptive health

Dockerize gives you the ability to wait for services on a specified protocol (file, tcp, tcp4, tcp6, http, https and unix) before starting your application.

github.com/jwilder/dockerize

```
version: "3"
services:
  web:
    image: project/web
    command: >
      dockerize
      -wait tcp://db:3306
      -wait http://api:80
      web-project -start
  db:
    image: project/db
  api:
    image: project/api
```



Dependent containers

Each container should include mechanism
to **retry** and **reconnect** to dependent services
when the connection is dropped.

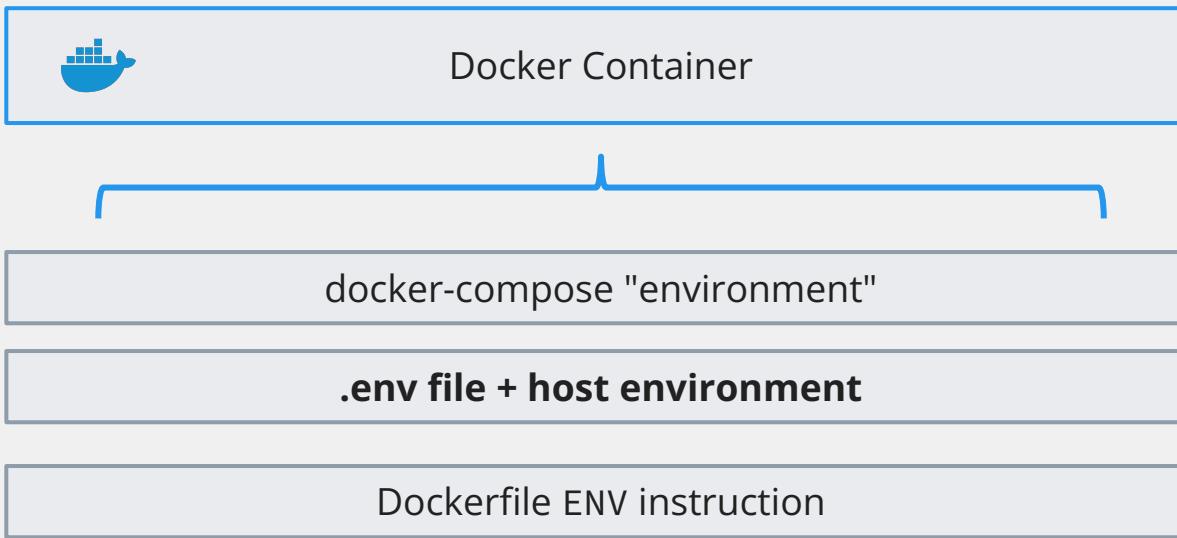
The solution should be implemented in the application logic,
container health status is meant only for orchestration.



Docker Compose .env file



Passing the environment



The .env file

Compose supports declaring default environment variables in an environment file named `.env` placed in the folder where the `docker-compose` command is executed.

- Compose expects each line to be in `VARIABLE=value` format.
- Lines beginning with `#` are processed as comments and ignored.



Example

```
● ● ● bash  
$ cat .env  
TAG=v1.5  
  
$ cat docker-compose.yml  
version: '3'  
services:  
  web:  
    image: "web:${TAG}"
```



Proof

The docker-compose config command prints resolved application config.

```
bash
$ docker-compose config
version: '3'
services:
  web:
    image: 'web:v1.5'
```



Duplicated environment

When you set the same environment variable in multiple files,
here is the priority used by Compose to choose which value to use:

1. Compose file
2. Shell environment variables
- 3. Environment file**
4. Dockerfile
5. Variable is not defined



Overwritten value

Values in the shell take precedence over those specified in the .env file.

```
bash
$ export TAG=v2.0
$ docker-compose config
version: '3'
services:
  web:
    image: 'web:v2.0'
```



Custom environment file

You can use a custom file with environment variables when using either docker and docker-compose commands.

```
bash  
$ cat ./custom.env  
VARIABLE=custom-value  
$ docker run \  
    --env-file ./custom.env \  
    alpine:3 env  
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:...  
HOSTNAME=7cf3147a6f1e  
VARIABLE=custom-value  
HOME=/root
```



Custom environment file

You can use a custom file with environment variables when using either docker and docker-compose commands.

```
bash
version: '3'
services:
  api:
    image: alpine:3
    env_file:
      - ./custom.env
    environment:
      - ENV=production
      - CACHE_LEVEL
```



Advanced docker-compose.yml



Services well organized

Organize docker-compose.yml with extensions, environment variables, YAML anchors and aliases.

```
x-project-vars:  
  - &project-name "my-docker-project"  
  - &project-version "${VERSION:-dev}"  
  - &syslog-host "tcp://127.0.0.1:1514"  
  - &web-replicas 3  
  
x-logger-syslog: &syslog  
  logging:  
    driver: "syslog"  
    options:  
      syslog-address: *syslog-host  
      tag: *project-name  
  
version: "3.4"  
services:  
  web:  
    image: project/web  
    environment:  
      project_name: *project-name  
      project_version: *project-version  
    deploy:  
      replicas: *web-replicas  
  db:  
    <<: *syslog  
    image: mysql:5.7
```



Exporting Docker image to file



Export Docker image to file

Docker images can be exported into an archive and imported later on.

```
bash

# Store all image tags in a single archive
$ docker save alpine > alpine.tar
# Store a single image tag in an archive
$ docker save -o alpine-3.9.tar alpine:3.9

$ docker load -i alpine-3.9.tar
Loaded image: alpine:3.9

$ docker save alpine | \
    ssh user@remote-server.com docker load
```



Running a local image registry



Docker Registry

The Registry is a stateless, highly scalable server side application that stores and lets you distribute Docker images.

The Registry is open-source.

<https://docs.docker.com/registry/>

https://hub.docker.com/_/registry



Run local Docker Registry

Run local Docker Registry to persist the image and be able to share it.

```
bash  
$ docker volume create registry_data  
$ docker run -d \  
  -v registry_data:/var/lib/registry \  
  -p 127.0.0.1:5000:5000 \  
  registry:2
```



Persist the image

Tag the image to point the local Docker Registry and push it.

```
bash  
$ docker tag nginx:1.19 localhost:5000/nginx:1.19  
$ docker push localhost:5000/nginx:1.19  
The push refers to repository [localhost:5000/nginx:1.19]  
1335c04e27fc: Pushed  
b1b32e60a368: Pushed  
b8a1f9c66f90: Pushed  
f1b5933fe4b5: Pushed  
latest: digest: sha256:9a04bce134177a28b71010cd... size: 1155
```



Proof

Remove the local images and try to run any, it should be pulled automatically from the Docker Registry.

```
● ● ● bash
$ docker rmi -f nginx:1.19 localhost:5000/nginx:1.19
Untagged: nginx:1.19
Untagged: localhost:5000/nginx:1.19
Deleted: sha256:d5238c5102f639d4d1eeb7ecf3db26c4b8cb25771443
$ docker run localhost:5000/nginx:1.19
```



Storing Docker images



Image registry



A registry is a storage and content delivery system, holding named Docker images, available in different tagged versions.



Image registry use cases

- store and share Docker images with co-workers inside a company, working on the same project or even publicly
- integrate with and complement your CI/CD system
- distribute a new image over a large cluster of machines
- distribute images inside an isolated network



Image registry solutions

Self-hosted:

- Docker Registry
- Harbor
- GitLab Registry

Cloud-based:

- Docker Hub
- GitHub Packages
- GitLab
- Google Container Registry
- Amazon ECR



Image namespaces

Root images

e.g. debian, mysql, busybox

The root namespace is for official images.

They are put there by Docker Inc., but they are generally authored and maintained by third parties.



Image namespaces

User and organizations images

e.g. `lukaszlach/clock`

This namespace holds user and organization images who expose their Docker images and projects publicly.



Image namespaces

Self-hosted images

e.g. `registry.com:5000/project/image`

Dedicated to images which are not hosted on Docker Hub, but on third party registries. They contain the hostname or IP address, optionally the port, of the registry server.



GitHub Packages



Image registry configuration and security



Run local Docker Registry

Run local Docker Registry to persist the image and be able to share it.

```
bash  
$ docker volume create registry_data  
$ docker run -d \  
  -v registry_data:/var/lib/registry \  
  -p 127.0.0.1:5000:5000 \  
  registry:2
```



Running a Docker Hub mirror



Run local Docker Registry

Run a Docker Hub mirror.

```
bash  
$ docker volume create registry_data  
$ docker run -d \  
  -v registry_data:/var/lib/registry \  
  -e REGISTRY_PROXY_REMOTEURL=https://registry-1.docker.io \  
  -e REGISTRY_STORAGE_DELETE_ENABLED=true \  
  -p 127.0.0.1:5000:5000 \  
  registry:2
```



Internals of communication with a Docker Registry



Docker image internals



Challenge

Explore chosen image internals.

Explore contents of a Docker image -
configuration, manifest, and the image layers themselves.



Image archive

Explore the image contents without extracting it.

```
bash  
$ docker save alpine:latest | tar -tf -  
34c6804a20d163e5a63e7e2b2ca404bbea473a00b59362a329eb14474daaf/  
34c6804a20d163e5a63e7e2b2ca404bbea473a00b59362a329eb14474daaf/VERSION  
34c6804a20d163e5a63e7e2b2ca404bbea473a00b59362a329eb14474daaf/json  
34c6804a20d163e5a63e7e2b2ca404bbea473a00b59362a329eb14474daaf/layer.tar  
d4ff818577bc193b309b355b02ebc9220427090057b54a59e73b79bdfe139.json  
manifest.json  
repositories  
  
$ docker save alpine:latest | \  
tar -xOf - manifest.json
```



Image archive

Extract `redis:latest` Docker image to `/tmp/image` directory.

```
bash  
$ mkdir /tmp/image  
$ docker save redis:latest | tar -C /tmp/image/ -xv  
$ ls /tmp/image  
08502081bff61084d64fc76f0f90ea39b89935cd071d9e12c5374ae191ff53c0.json  
374663a8f0988a631966c00ddc24b66048985be4e2cf3c2af8a03b93943f084  
...  
f7d5800201179f62c95c94d3cb1d447df28e6b621ecb9a248ed21a4395673f19  
manifest.json  
repositories
```



Image internals

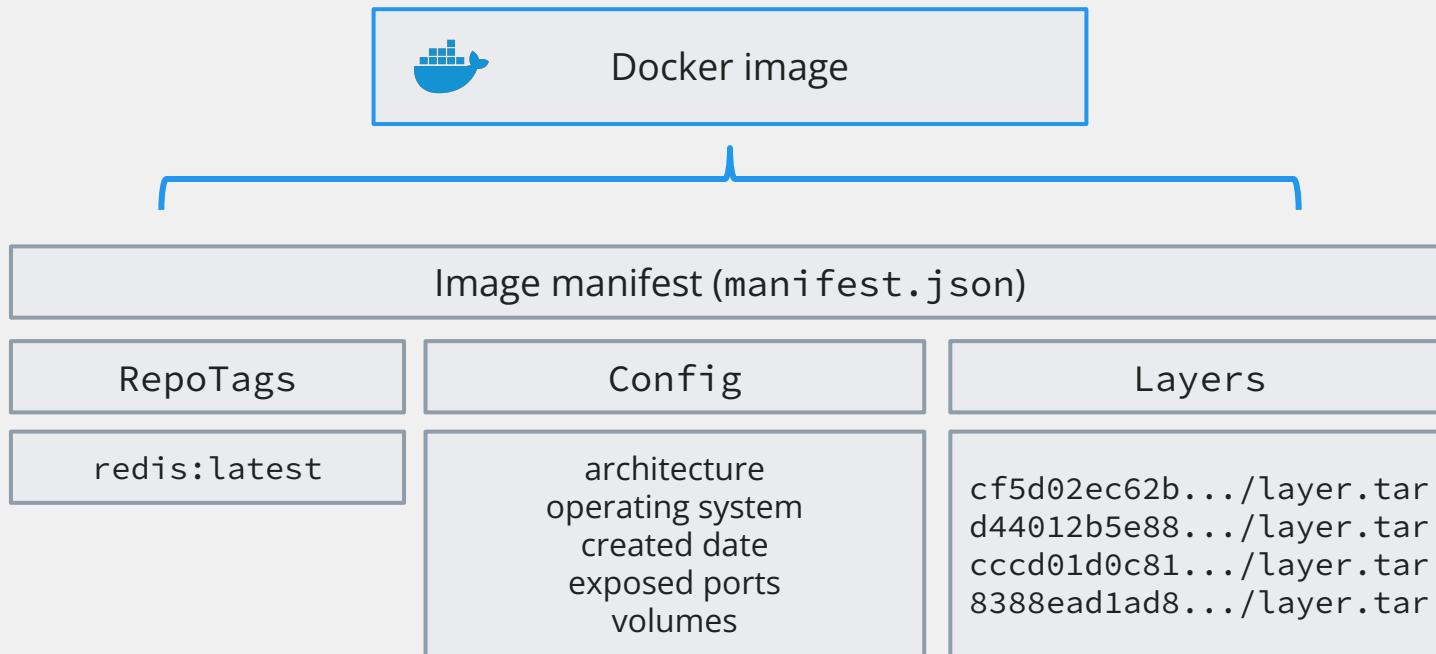


Image manifest

The Docker image has six layers pointed by tar archives.

```
bash
$ jq '. .' < /tmp/image/manifest.json
[
  {
    "Config": "08502081bff61084d64fc76f0f90ea39b89935cd071d9e1274ae191ff53c0.json",
    "RepoTags": [
      "redis:latest"
    ],
    "Layers": [
      "99822641346ab3f492352d3e866bee8d4f7a555fce6f7dc216f7ec24fa6f19d/layer.tar",
      "6c5d0141f27d6c48bf6001257ee5638be14fe1051a28b2743658b1e8bb6c29ef/layer.tar",
      "87c8d9d768ec8a6ef7f7b7aad708bf0b451b08b661431000a82cb873e53cea6c/layer.tar",
      "9760352c46cd6de0fcb5848f72e3802f31cebae8fc388a5cd59e489516348160/layer.tar",
      "f7d5800201179f62c95c94d3cb1d447df28e6b621ecb9a248ed21a4395673f19/layer.tar",
      "374663a8f0988a631966c00ddc24b66048985be4e2cf3c2af8a03b93943f084/layer.tar"
    ]
  }
]
```



Image configuration

```
bash  
$ jq -r '.[0].Config' < /tmp/image/manifest.json  
$ jq '.history' < \  
  /tmp/image/$(jq -r '.[0].Config' < /tmp/image/manifest.json)  
[  
  {  
    "created": "2020-04-24T01:05:35.458457398Z",  
    "created_by": "/bin/sh -c #(nop) ADD file:a0afdb0b0db7f9ee949618c00 in / "  
  },  
  {  
    "created": "2020-04-24T01:05:35.807646609Z",  
    "created_by": "/bin/sh -c #(nop)  CMD [\"/bin/sh\"]",  
    "empty_layer": true  
  },  
  ...
```



Image layer

Pick the second layer of the image
and extract the layer archive file to /tmp/image/layer.

```
bash

$ jq -r '.[0].Layers[1]' < /tmp/image/manifest.json
d44012b5e880363c488d909bcc7932645111e83660ec2d3dd25/layer.tar

$ mkdir /tmp/image/layer
$ tar -C /tmp/image/layer -xvf \
    /tmp/image/$(jq -r '.[0].Layers[1]' < /tmp/image/manifest.json)
etc/
etc/.pwd.lock
etc/group
etc/group-
```



Image layer

This is the layer created by a RUN instruction
that creates orca user and group and installs su-exec.

 bash

```
$ find /tmp/image/layer
etc/
etc/.pwd.lock
etc/group
etc/group-
etc/gshadow
etc/gshadow-
etc/passwd
etc/passwd-
etc/shadow
etc/shadow-
```



Image filesystem

Extract all image layers to a single directory.

```
bash  
$ mkdir /tmp/image/filesystem  
$ jq -r '.[0].Layers | .[]' < /tmp/image/manifest.json | \  
  xargs -n1 -I{} \  
    tar -C /tmp/image/filesystem -xvf "/tmp/image/{}"  
bin/  
bin/bash  
bin/cat  
bin/chgrp  
bin/chmod  
bin/chown  
bin/cp
```



Image filesystem

This is how the container filesystem looks like after creation and before running the main process.

```
bash
$ cd /tmp/image/filesystem
$ ls
bin    data   etc    lib     media   opt     root   sbin   sys    usr
boot   dev    home   lib64   mnt    proc    run    srv    tmp    var

$ find . -name redis-server
./usr/local/bin/redis-server

$ ./usr/local/bin/redis-server
```



Modify a Docker image without rebuilding



Automatic completion of Docker command-line



Run Microsoft VS Code in a Docker container



BACKEND



Python in Docker



Python in Docker

Python is an interpreted, interactive, object-oriented, open-source programming language. It incorporates modules, exceptions, dynamic typing, very high level dynamic data types, and classes. Python combines remarkable power with very clear syntax

Image

python - official Docker image

https://hub.docker.com/_/python

Ports

none

Volumes

none

Other

- Lightweight Alpine version of the image.
- Slim version of the image containing the minimal packages needed to run python.
- Windows-based image variants.



Ruby in Docker



Ruby in Docker

Ruby is a dynamic, reflective, object-oriented, general-purpose, open-source programming language. It supports multiple programming paradigms, including functional, object-oriented, and imperative. It also has a dynamic type system and automatic memory management.

Image

ruby - official Docker image

https://hub.docker.com/_/ruby

Ports

none

Volumes

none

Other

- Lightweight Alpine version of the image.
- Slim version of the image containing the minimal packages needed to run Ruby.
- Windows-based image variants.



Node.JS in Docker



Node.JS in Docker

Node.js is a software platform for scalable server-side and networking applications.

Node.js applications are written in JavaScript and can be run within the Node.js runtime on Mac OS X, Windows, and Linux without changes.

Image

node - official Docker image

https://hub.docker.com/_/node

Ports

none

Volumes

none

Other

- Lightweight Alpine version of the image.
- Slim version of the image containing the minimal packages needed to run Node.JS.



GoLang in Docker



GoLang in Docker

Go (a.k.a., Golang) is a programming language first developed at Google. It is a statically-typed language with syntax loosely derived from C, but with additional features such as garbage collection, type safety, some dynamic-typing capabilities, additional built-in types (e.g., variable-length arrays and key-value maps), and a large standard library.

Image

golang - official Docker image

https://hub.docker.com/_/golang

Ports

none

Volumes

none

Other

- Lightweight Alpine version of the image.
- Windows-based image variants.



PHP in Docker



PHP in Docker

PHP is a server-side scripting language designed for web development, but which can also be used as a general-purpose programming language. PHP can be added to straight HTML or it can be used with a variety of templating engines and web frameworks.

Image

php - official Docker image

https://hub.docker.com/_/php

Ports

none

Volumes

none

Other

- Lightweight Alpine version of the image.
- FPM version of the image to be used with external reverse-proxy
- Apache version of the image with a built-in web server



Running an entire application stack in Docker



Memcached in Docker



Memcached in Docker

Memcached is a general-purpose distributed memory caching system. It is often used to speed up dynamic database-driven websites by caching data and objects in RAM to reduce the number of times an external data source must be read.

Image

memcached - official Docker image

https://hub.docker.com/_/memcached

Ports

11211 / tcp

Volumes

none

Other

- Lightweight Alpine version of the image.
- Limiting memory supports native Docker mechanism.



MySQL in Docker



MySQL in Docker

MySQL is the world's most popular open source database. With its proven performance, reliability and ease-of-use, MySQL has become the leading database choice for web-based applications, covering the entire range from personal projects and websites.

Image

mysql - official Docker image

https://hub.docker.com/_/mysql

Ports

3306 / tcp

33060 / tcp

Volumes

/var/lib/mysql

Other

- Requires an environment variable describing administration user creation strategy, used only once on first start



Postgres in Docker



Postgres in Docker

PostgreSQL, often simply "Postgres", is an object-relational database management system (ORDBMS) with an emphasis on extensibility and standards-compliance. As a database server, its primary function is to store data, securely and supporting best practices, and retrieve it later.

Image

postgres - official Docker image

https://hub.docker.com/_/postgres

Ports

5432 / tcp

Volumes

/var/lib/postgresql/data

Other

- Lightweight Alpine version of the image.
- Requires an environment variable describing administration user creation strategy, used only once on first start



MariaDB in Docker



MariaDB in Docker

MariaDB Server is one of the most popular database servers in the world. It's made by the original developers of MySQL and guaranteed to stay open source. The intent is also to maintain high compatibility with MySQL, ensuring a library binary equivalency and exact matching with MySQL APIs and commands.

Image

mariadb - official Docker image

https://hub.docker.com/_/mariadb

Ports

3306 / tcp

Volumes

/var/lib/mysql

Other

- Compatible with MySQL API, the image can be run with the exact same options.
- Requires an environment variable describing administration user creation strategy, used only once on first start



MongoDB in Docker



MongoDB in Docker

MongoDB is a free and open-source cross-platform document-oriented database program. Classified as a NoSQL database program, MongoDB uses JSON-like documents with schema.

Ports

27017 / tcp

Volumes

/data/db

Image

mongo - official Docker image

https://hub.docker.com/_/mongo



ElasticSearch in Docker



ElasticSearch in Docker

Elasticsearch is a distributed, RESTful search and analytics engine capable of solving a growing number of use cases. As the heart of the Elastic Stack, it centrally stores your data so you can discover the expected and uncover the unexpected.

Image

elasticsearch - official Docker image
https://hub.docker.com/_/elasticsearch

Ports

9200 / tcp
9300 / tcp

Volumes

none



Apache Solr in Docker



Apache Solr in Docker

Solr is highly reliable, scalable and fault tolerant, providing distributed indexing, replication and load-balanced querying, automated failover and recovery, centralized configuration and more. Solr powers the search and navigation features of many of the world's largest internet sites.

Image

solr - official Docker image

https://hub.docker.com/_/solr

Ports

8983

Volumes

/var/solr

Other

- Slim versions of the image.



RabbitMQ in Docker



RabbitMQ in Docker

RabbitMQ is open source message broker software that implements the Advanced Message Queuing Protocol (AMQP). The RabbitMQ server is written in the Erlang. Client libraries to interface with the broker are available for all major programming languages.

Image

rabbitmq - official Docker image

https://hub.docker.com/_/rabbitmq

Ports

4369	5671	5672
15691	15692	25672

Volumes

/var/lib/rabbitmq

Other

- Management versions of the image providing a web interface.



Redis in Docker



Redis in Docker

Redis is an open-source, networked, in-memory, key-value data store with optional durability. It is written in ANSI C. Redis is the most popular key-value store. The name Redis means REmote DIctionary Server.

Image

redis - official Docker image

https://hub.docker.com/_/redis

Ports

6379 / tcp

Volumes

/data

Other

- Lightweight Alpine version of the image.



Stateful applications in Docker



Reverse-proxy load-balancer in Docker



nginx load-balancer in Docker



nginx in Docker

Nginx (pronounced "engine-x") is an open source reverse proxy server for HTTP, HTTPS, SMTP, POP3, and IMAP protocols, as well as a load balancer, HTTP cache, and a web server (origin server). The nginx project started with a strong focus on high concurrency, high performance and low memory usage.

Image

nginx - official Docker image
https://hub.docker.com/_/nginx

Ports

80 / tcp

Volumes

none

Other

- Lightweight Alpine version of the image.
- Image variant with Perl pre-installed.





bash

```
$ cat nginx.conf
worker_processes 1;
pid /run/nginx.pid;
events {
    worker_connections 1024;
}
http {
    include mime.types;
    default_type application/octet-stream;
    sendfile on;
    keepalive_timeout 0;
    gzip on;

    access_log /proc/1/fd/1;
    error_log /proc/1/fd/1;

    include /etc/nginx/conf.d/*.conf;
}
```





bash

```
$ cat conf.d/web.conf
upstream php-upstream {
    server php:9000;
}
server {
    listen 80;
    server_name _;
    root /var/www;
    location / {
        try_files $uri @rewriteapp;
    }
    location @rewriteapp {
        rewrite ^(.*)$ /index.php/$1 last;
    }
    location ~ ^/(index)\.php(/|$) {
        fastcgi_pass php-upstream;
        fastcgi_split_path_info ^(.+\.php)(/.*)$;
        include fastcgi_params;
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
    }
}
```



Varnish caching load-balancer in Docker



Varnish

Varnish is a caching reverse-proxy that can:

- pick a HTTP backend using one of available load-balancing strategies
- pass the request to a backend and nothing more
- additionally cache the response if you sent a valid caching header or used VCL for it
- serve a stalled response from cache while still updating it in background with request coalescing
- serve a stalled response when backend servers are down



Varnish in Docker

Varnish is an HTTP accelerator designed for content-heavy dynamic web sites as well as APIs. In contrast to other web accelerators, such as Squid, which began life as a client-side cache, or Apache and nginx, which are primarily origin servers, Varnish was designed as an HTTP accelerator.

Image

varnish - official Docker image
https://hub.docker.com/_/varnish

Ports

80 / tcp

8443 / tcp

Volumes

none



VCL

VCL is a programming language

- Varnish translates VCL into binary code which is then executed when requests arrive
- VCL files are organized into subroutines
- if you do not call an action in your subroutine and it reaches the end Varnish will execute a built-in VCL
- does not have any loops, jump statements, custom variables, classes or functions





bash

```
$ cat default.vcl
vcl 4.0;
import directors;

backend application_1 {
    .host = "application_1";
    .port = "80";
}

sub vcl_init {
    new application = directors.round_robin();
    application.add_backend(application_1);
}

sub vcl_recv {
    set req.backend_hint = application.backend();
}
```



Accessing the container shell and resources



Docker image weight classes



Lightweight Image

Base OS and core tools

Application and libraries



Flyweight Image

Application and libraries



Heavyweight Image

Base OS and core tools

Application and libraries

Network tools ping, curl, wget

Process tools ps, top, htop

Debug tools strace, gdb

Other bash, vim



Let's try

Run nginx web server container from the official image.



bash | Challenge

```
# The container does not publish any port
$ docker run -d --name nginx nginx
```



Command not found

Additional tools should not be a part of an image,
and are not in this case.

```
● ● ● bash  
$ docker exec nginx ps aux  
OCI runtime exec failed: exec failed: container_linux.go:344:  
starting container process caused "exec: \"ps\": executable  
file not found in $PATH": unknown  
$ docker exec nginx curl localhost  
OCI runtime exec failed: exec failed: container_linux.go:344:  
starting container process caused "exec: \"curl\": executable  
file not found in $PATH": unknown
```



Obtaining access to host environment, shell and tools in any container



cntr

Use the cntr tool (Linux only) - github.com/Mic92/cntr - to access both network and process namespaces of the other container and use all the tools installed on the host system.

```
● ● ● bash
$ sudo cntr attach nginx
/ # curl
curl: try 'curl --help' or 'curl --manual' for more
information
/ # http localhost
/ # tcpdump
```



cntr

Access both container (/var/lib/cntr) and host (/) filesystems.



bash

```
$ sudo cntr attach nginx
# Edit container file using vim from the host
/ # vim /var/lib/cntr/etc/nginx/nginx.conf
# Synchronize host and container project directory
# using rsync from the host
/ # rsync -avh \
    /home/ll/project/etc/nginx \
    /var/lib/cntr/etc/nginx
```



No shell change

As a side effect when taking all your host tools with you to the container and because of the filesystem structure, cntr provides you your shell environment.

Your zsh, oh-my-zsh and fuzzy finder, shell aliases and history will be available inside the attached container.



Testing an application with different versions of programming languages and used technologies



Docker image and container security



Container CPU and RAM resource usage



Limit resources

Containers will automatically have access to the **entire range of RAM and CPU** processing power of its host. When you start running multiple containers, each one can start stepping on each other.

```
$ docker run --memory="2.5g" --cpus="1.5" <image>  
$ docker update --memory="2.5g" --cpus="1.5" <container>
```

<https://lach.dev/docker-limit-resources>



Container restart policy



Restart policy

To configure the restart policy for a container,
use the `--restart` flag when using the `docker run` command.

- `no` - Do not automatically restart the container (the default)
- `on-failure` - Restart the container if it exits due to an error,
which manifests as a non-zero exit code.
- `always` - Always restart the container if it stops.
- `unless-stopped` - Similar to `always`, except that when the container is stopped
(manually or otherwise), it is not restarted.
- suffixing any of the above with `:LIMIT` allows to limit number of retries



Restart policy

Container restart policy can be changed when its running.

```
● ● ● bash  
# Start a Redis container and configure it to never restart  
$ docker run -d --restart no redis  
  
# Change the restart policy for an already running container  
$ docker update --restart unless-stopped redis  
  
# Alter all currently running containers  
$ docker update --restart unless-stopped \  
$(docker ps -q)
```



Container read-only root filesystem



Read-only filesystem

The `--read-only` flag mounts the root filesystem as read only prohibiting writes to locations other than the specified volumes.

```
bash
$ docker run --read-only alpine:3 touch /test
touch: /test: Read-only file system
```



Read-only filesystem

Volumes and mounts can be used in combination with `--read-only` to control where a container writes files.

```
bash

# Allow write access to /write volume directory
$ docker run --read-only -v /tmp/test:/test \
    alpine:3 touch /test/file
# Mount a temporary filesystem with write access under /test
$ docker run --read-only --tmpfs /test \
    alpine:3 touch /test/file
$ docker run --read-only \
    --mount type=tmpfs,destination=/test,tmpfs-size=1024 \
    alpine:3 touch /test/file
```



Privileged containers



Privileged containers are containers that have **all of the root capabilities of a host machine**, allowing the ability to access resources which are not accessible by default.



Privileged container

Use `--cap-add` to selectively extend container capabilities or `--privileged` to give it root capabilities of a host machine.

```
bash
$ docker run cmd.cat/strace strace -p 123
strace: attach: ptrace(PTRACE_ATTACH, 123): Operation not
permitted

$ docker run --cap-add SYS_PTRACE \
    cmd.cat/strace strace -p 123
$ docker run --privileged \
    cmd.cat/strace strace -p 123
```



Privileged container



bash

```
$ docker run --privileged debian:buster bash -c 'mkdir /tmp/devices; cd /tmp/devices; for i in $(seq 1 300); do mknod device_$i b $i 0; done; fdisk -l * 2>/dev/null'
Disk device_1: 16 MiB, 16777216 bytes, 32768 sectors
...
Disk device_8: 40 GiB, 42949672960 bytes, 83886080 sectors
Disk model: QEMU HARDDISK
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0x2d062b28
Device      Boot Start      End  Sectors Size Id Type
device_8p1 *        2048 83886046 83883999  40G 83 Linux
```



Containers using a host network mode



If you use the host network mode for a container, **that container's network stack is not isolated from the Docker host**, and the container does not get its own IP-address allocated.



Netshoot

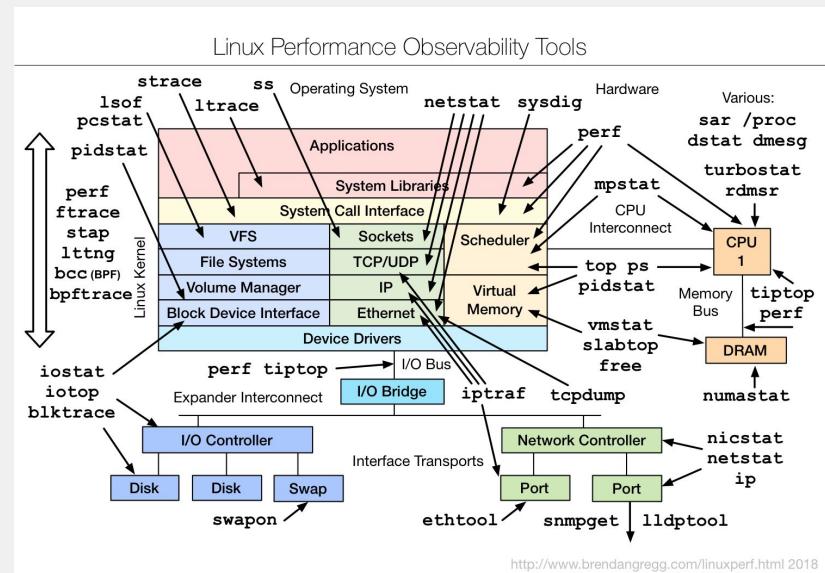
The netshoot is Docker and Kubernetes network troubleshooting swiss-army container that contains a set of powerful networking tools that can be used to troubleshoot Docker networking issues.

Container Network Namespace

```
$ docker run -it \
--net container:<id|name> \
nicolaka/netshoot
```

Host Network Namespace

```
$ docker run -it --net host \
nicolaka/netshoot
```



<http://www.brendangregg.com/linuxperf.html> 2018



Securing Docker container with AppArmor



Running a containerized application from dedicated user and group



Dedicated user

Anything running in a container should be treated with the same consideration as anything running on the host itself.

By default, root in a container is the same root (UID 0) as on the host machine.

It is good practice to create a dedicated group and user, with a scope limited to only what is needed to run an application in the container.



USER

The USER instruction sets the user name (or UID) and optionally the user group (or GID) to use when running the image and for any RUN, CMD and ENTRYPOINT instructions that follow it in the Dockerfile.

USER <user>[:<group>]

USER <UID>[:<GID>]

USER www-data:www-data

USER 101:101

USER 999



Dropping privileges at runtime

Using an `USER` instruction in Dockerfile sets the same user and group for following `CMD` and `ENTRYPOINT` instructions.

Both `docker run` and `docker exec` commands accept `-u 0` parameter which allows to switch back to the root account.

By using tools like `gosu` or `su-exec`, Docker can run the container entrypoint as root which drops the privilege when it is no longer needed.

```
$ su-exec user[:group] command      $ gosu user[:group] command
```



COPY --chown

COPY creates all new files and directories with a UID and GID of 0 (root), unless the optional --chown flag specifies a given username, group name, or UID/GID combination to request specific ownership of the copied content.

```
COPY --chown=www-data:www-data . /var/www
```

```
COPY --chown=101 index* /srv/
```



Exploring Docker image and container filesystems



Docker security scanners - Aqua Microscanner



Aqua Microscanner



A free tool that scans images for vulnerabilities.

<https://github.com/aquasecurity/microscanner>

If the image contains any packages with vulnerabilities, it will display a detailed summary with the average CVE score. Scan will stop the build if it finds a high or critical vulnerability.



Get the access token

To use MicroScanner you first need to register a token.



Scan any Docker image

Microscanner is designed to be run as part of building a Docker image, if high severity vulnerabilities are found, will fail the image build.

```
$ docker build --no-cache -t security-scan - <<'EOF'  
FROM any:docker-image  
ADD https://get.aquasec.com/microscanner .  
RUN apk add --no-cache ca-certificates && \  
    chmod +x ./microscanner && \  
    ./microscanner YourAccessToken  
EOF
```



ADD

The ADD instruction copies new files, directories or **remote file URLs** from <src> and adds them to the filesystem of the image at <dest>.

If <src> is **a local archive** in a recognized compression format (identity, gzip, bzip2 or xz) then it is unpacked as a directory.

```
ADD <src>[ , <src>, ...] <dest>
```

```
ADD *.png /var/www
```

```
ADD archive.tar /
```



Docker security scanners - Anchore



Docker security scanners - Trivy



Trivy

Trivy is a simple and complex vulnerability scanner for Docker images.
It scans OS packages (Alpine, RHEL, CentOS, Debian, ...)
and application dependencies (Composer, npm, yarn, pip, gem, ...).

```
$ docker volume create trivy-cache  
$ docker run  
  -v /var/run/docker.sock:/var/run/docker.sock \  
  -v trivy-cache:/root/.cache/ \  
  aquasec/trivy alpine:3.9
```



bash-3.2\$ trivy -h
NAME:
trivy - A simple and comprehensive vulnerability scanner for containers
USAGE:
trivy [options] image_name
VERSION:
0.0.11
OPTIONS:
--format value, -f value format (table, json) (default: "table")
--input value, -i value input file path instead of image name
--severity value, -s value severities of vulnerabilities to be displayed (comma separated) (default: "UNKNOWN,LOW,MEDIUM,HIGH,CRITICAL")
--output value, -o value output file name
--exit-code value Exit code when vulnerabilities were found (default: 0)
--skip-update skip db update
--clean, -c clean all cache
--quiet, -q suppress progress bar
--ignore-unfixed display only fixed vulnerabilities
--refresh refresh DB (usually used after version update of trivy)
--debug, -d debug mode
--help, -h show help
--version, -v print the version
bash-3.2\$]

<https://aquasecurity.github.io/trivy/latest/>



Orchestration introduction



Docker Swarm



Docker Swarm - setup a cluster



Local cluster

Initialize one-node swarm on your machine making it a master of the newly created cluster.

```
● ● ● bash | Orchestrate  
$ docker swarm init --advertise-addr eth1  
Swarm initialized: current node (o0x9it2xenrsq31s1geq41a4n)  
is now a manager.  
To add a worker to this swarm, run the following command:  
  
docker swarm join --token  
SWMTKN-1-2z3fkalzpbvirb24mgpamy479247wc96cr70hwixc9rofnqn mh-3tpex63g79z  
0keqtusz1p7gj 172.11.0.101:2377  
...
```



Overlay network

Overlay networks connect multiple Docker daemons together and enable swarm services to communicate with each other. Docker transparently handles routing of each packet to and from the correct Docker daemon host and the correct destination container.

Overlay network can encrypt data passing through in which case Docker creates IPSEC tunnels between all the nodes where tasks are scheduled.

```
$ docker run --net ingress
```



Docker Swarm - deploy an application stack



Deploy a service stack

Use the existing **docker-compose.yml** file
and with no changes deploy it as a Swarm stack.

```
● ● ● bash
$ docker-compose down
$ docker stack deploy -c docker-compose.yml orca
Ignoring unsupported options: build
Ignoring deprecated options:
container_name: Setting the container name is not supported.
Creating network orca_default
Creating service orca_app
Creating service orca_mysql
```



Deploy a service stack

Use the existing **docker-compose.yml** file
and with no changes deploy it as a Swarm stack.

```
bash
$ docker-compose down
$ docker-compose config | \
    docker stack deploy -c - orca
Ignoring unsupported options: build
Ignoring deprecated options:
container_name: Setting the container name is not supported.
Creating network orca_default
Creating service orca_app
Creating service orca_mysql
```



Limitations

Not everything supported in docker-compose.yml
is covered when deploying to Docker Swarm.

lach.dev/compose-swarm-limitations



Display running services

Check if the containers are running as Swarm services.

```
bash  
$ docker service ls  
ID          NAME      MODE      REPLICAS  IMAGE      PORTS  
j1ib998dxt85  orca_mysql  replicated  1/1       mysql:5.7  
yp4g36d4qe0k  orca_app    replicated  1/1       orca:latest  *:8080...  
  
$ docker service ps orca_app  
ID          NAME      IMAGE      NODE      DESIRED STATE  ...  
cdha541wlrfk  orca_app.1  orca:latest  docker-desktop  Running  ...  
  
$ docker ps
```



Scale the service

Scale the service up to 3 replicas in total.

Docker Swarm is now load-balancing the traffic across all replicas.

```
bash
$ docker service scale orca_app=3
orca_app scaled to 3
overall progress: 3 out of 3 tasks
1/3: running  [=====>]
2/3: running  [=====>]
3/3: running  [=====>]
verify: Service converged
```

```
$ watch curl -v localhost
```



Container health

The **HEALTHCHECK** instruction allows to set a command that determines container health status. Only when the status changes to "healthy" the orchestrator starts routing traffic to container instances.

```
HEALTHCHECK --interval=1m --timeout=10s CMD ( \
    # check if HTTPS port is opened
    lsof -i :443 && \
    # send HTTP request
    curl -sSf localhost:80/status \
) || exit 1
```



Remove the service stack

This will remove the stack with the name orca. Services, networks, and secrets associated with the stack will be removed.



bash

```
$ docker stack rm orca
Removing service orca_mysql
Removing service orca_app
Removing network orca_default
```



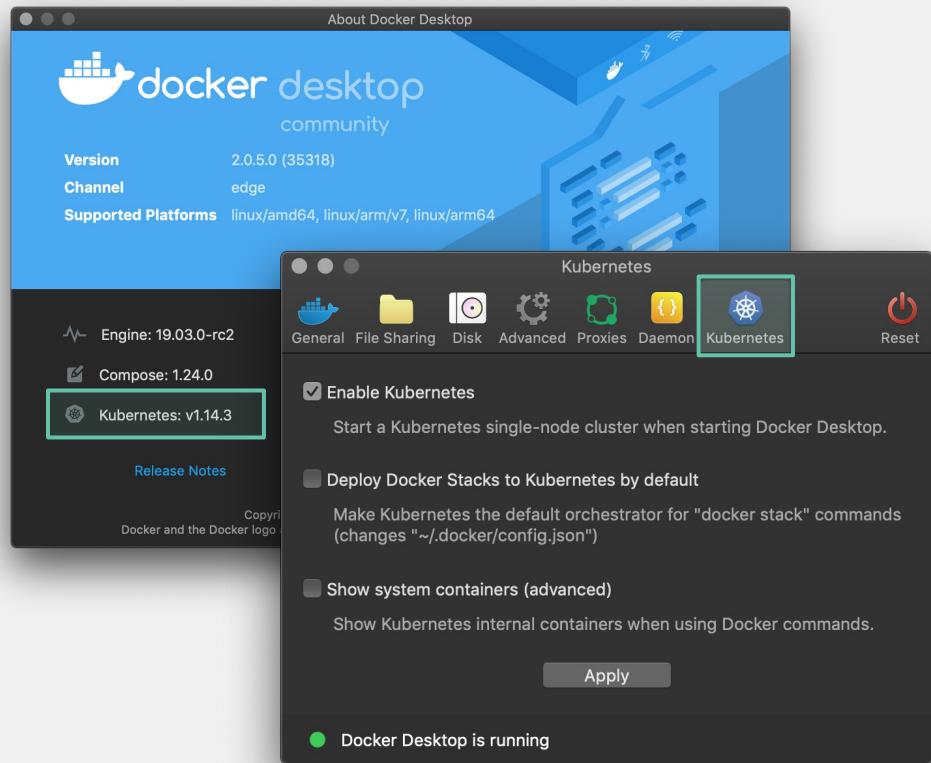
Kubernetes introduction



Local Kubernetes cluster

When Kubernetes support is enabled in Docker Desktop, you can deploy your workloads, in parallel, on Kubernetes, Swarm, and as standalone containers. Enabling or disabling Kubernetes does not affect your other workloads.

```
$ docker stack deploy \
--orchestrator kubernetes \
-c docker-compose.yml orca
```



Kubernetes in Docker



Kubernetes-in-Docker

kind works similar to docker:dind but aims to make a full Kubernetes cluster available to test against in a CI pipeline.

```
$ docker run -d -it --privileged -p 8443:8443 -p 10080:10080 \
  bsycorp/kind:latest-1.19
$ until curl -sf 127.0.0.1:10080/kubernetes-ready; do
  sleep 1s;
done

$ ./test-kubernetes-integration.sh
```



Run Kubernetes in Docker

Start a single-node Kubernetes cluster as a Docker container. Except for management ports, the application port 8080 is also published.

```
bash  
$ docker run --rm --name kind -it \  
  --privileged \  
  -p 8443:8443 -p 10080:10080 \  
  -p 8080:30080 \  
  bsycorp/kind:latest-1.19
```

^PQ



Cuddle the cluster

Container publishes port 10080 which can be used to retrieve kubectl configuration and certificates needed to access the cluster locally. You can also make an alias to kubectl inside the container.

```
bash  
$ alias kubectl='docker exec -it kind kubectl'  
$ kubectl get nodes  
NAME      STATUS    ROLES      AGE      VERSION  
minikube  Ready     master, node  12d      v1.19.1  
$ unalias kubectl
```



We need to go deeper

Docker-in-Docker

A container has its own Docker Engine installed and running.

```
RUN curl -sSfL get.docker.com | bash
```

Shared Docker socket

A container, with Docker installed, which does not run its own Docker daemon, but connects to the Docker daemon of the host system.

```
$ docker run -v /var/run/docker.sock:/var/run/docker.sock
```



Transfer the Docker image

As Kubernetes-in-Docker runs Docker-in-Docker and the Docker image is not available in any registry so it needs to be transferred manually.

```
bash
$ docker save app:1.0 | \
    docker exec -i kind docker load
Loaded image: orca:latest

$ docker exec kind docker images | grep orca
app          1.0      c4a0274b8d17  2 hours ago  75.1MB
```





app.yml

```
---  
apiVersion: apps/v1  
kind: Deployment  
metadata:  
  name: app  
spec:  
  selector:  
    matchLabels:  
      app: app  
  replicas: 3  
  template:  
    metadata:  
      labels:  
        app: app  
    spec:  
      containers:  
      - name: app  
        image: nginx:1.19  
      ports:  
      - containerPort: 80  
---  
apiVersion: v1  
kind: Service  
metadata:  
  name: app  
spec:  
  ports:  
  - name: http  
    protocol: TCP  
    port: 80  
    targetPort: 80  
    nodePort: 30080  
  selector:  
    app: app  
  type: LoadBalancer
```



Deploy

Create and update Resources in a cluster by running `kubectl apply` on a Resource Config. Apply manages complexity such as ordering of operations and merging user defined and cluster defined state.

```
● ● ● bash  
$ docker cp app.yml kind:/  
$ docker exec -it kind \  
    kubectl apply -f /app.yml  
deployment.apps/app created  
service/app created
```



Connection established

```
● ● ● bash  
$ curl -v 0:8080  
> GET / HTTP/1.1  
> Host: 0:8080  
> User-Agent: curl/7.50.0  
> Accept: */*  
>  
< HTTP/1.1 200 OK  
< Server: nginx/1.19.10  
< Content-Type: text/html  
< Content-Length: 612  
< Connection: keep-alive  
<
```



Inspect the cluster status



bash

```
$ docker exec kind kubectl get all
```

NAME	READY	STATUS	RESTARTS	AGE
pod/app-6cfdd85dbd-258jf	1/1	Running	0	8m29s
pod/app-6cfdd85dbd-5rfc5	1/1	Running	0	8m31s
pod/app-6cfdd85dbd-n6cdj	1/1	Running	0	8m40s

NAME	TYPE	CLUSTER-IP	PORT(S)	AGE
service/app	LoadBalancer	10.96.33.180	80:30080/TCP	9m22s
service/kubernetes	ClusterIP	10.96.0.1	443/TCP	22d

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
deployment.apps/app	3/3	3	3	9m22s

NAME	DESIRED	CURRENT	READY	AGE
replicaset.apps/app-6cfdd85dbd	3	3	3	8m40s



Stop the cluster

Stop the Kubernetes cluster container
and remove it entirely from the disk.

```
● ● ● bash
$ docker rm -f kind
kind
```



Kubernetes	Docker Swarm
Developed by Google	Developed by Docker Inc.
Large open-source community	Smaller community
Extensive and customizable	Less extensive and less customizable
Requires heavy setup	Easy to set up and fits well into Docker ecosystem
Has high fault tolerance	Has lower fault tolerance
Provides strong guarantees to cluster states, at the expense of speed	Facilitates for quick container deployment and scaling even in very large clusters
Enables load balancing when container pods are defined as services	Features automated internal load balancing through any node in the cluster



From Docker Compose to Kubernetes



Variety of Linux distributions in Docker



Debian in Docker

Debian is an operating system which is composed primarily of free and open-source software, most of which is under the GNU General Public License, and developed by a group of individuals known as the Debian project.

Image

debian - official Docker image

https://hub.docker.com/_/debian

Image size
up to 50 MB

Package manager
apt / apt-get

Supported architectures
amd64, arm, arm64, 386,
mips64le, ppc64le, s390x



Ubuntu in Docker

Ubuntu is a Debian-based Linux operating system that runs from the desktop to the cloud, to all your internet connected things. It is the world's most popular operating system across public clouds and OpenStack clouds. It is the number one platform for containers.

Image

ubuntu - official Docker image

https://hub.docker.com/_/ubuntu

Image size
up to 60 MB

Package manager
apt / apt-get

Supported architectures
amd64, arm, arm64,
ppc64le, s390x



Fedora in Docker

Fedora is a Linux distribution developed by the community-supported Fedora Project which is sponsored primarily by Red Hat, a subsidiary of IBM, with additional support from other companies.

Image

fedora - official Docker image
https://hub.docker.com/_/fedora

Image size
up to 90 MB

Package manager
dnf

Supported architectures
amd64, arm, arm64,
ppc64le, s390x



CentOS in Docker

CentOS Linux is a community-supported distribution derived from sources freely provided to the public by Red Hat for Red Hat Enterprise Linux (RHEL). As such, CentOS Linux aims to be functionally compatible with RHEL. CentOS Linux is no-cost and free to redistribute.

Image

centos - official Docker image
https://hub.docker.com/_/centos

Image size
up to 75 MB

Package manager
yum

Supported architectures
amd64, arm64, ppc64le



Alpine Linux in Docker

Alpine Linux is a Linux distribution built around musl libc and BusyBox. The image is only 5 MB in size and has access to a package repository that is much more complete than other BusyBox based images. This makes Alpine Linux a great image base for utilities and even production applications.

Image

alpine - official Docker image
https://hub.docker.com/_/alpine

Image size

5 MB

Package manager

apk

Supported architectures

amd64, arm, arm64,
386, ppc64le, s390x



Favourite shell and tools in a container



Challenge

Access the container shell with Bash.



bash

```
$ docker run -d --name php php:8-fpm-alpine
$ docker exec -it php bash
OCI runtime exec failed: exec failed: container_linux.go:345:
starting container process caused "exec: \"bash\": executable
file not found in $PATH": unknown
```



Solution

Solution 1: Enter sh shell and install bash.

Non-repeatable and requires altering the container which can make it unstable.

```
● ● ● bash  
$ docker exec -it php sh  
/ # apk --no-cache add bash  
/ # bash  
bash-5.1#
```



Solution

Solution 2: Run other container in pid and net container mode.
~~Does not allow to access the target container filesystem,~~
still requires manual installation of additional tools.

```
bash  
$ docker run -it \  
  --net container:php \  
  --pid container:php \  
  debian:stretch bash  
root@20bac2ffc478:/#
```



Solution

Solution 3: Run cmd.cat container in pid and net container mode.
~~Does not allow to access the target container filesystem.~~
Include additional tools just by altering the image name.

```
bash
$ docker run -it \
  --net container:php \
  --pid container:php \
  cmd.cat/bash/ngrep/htop bash
bash-5.0#
```



Container filesystem

When you hook into other container using pid container mode, there is a way to access it's filesystem through /proc/1/root.

```
● ● ● bash
$ docker run -it --pid container:php bash
root@20bac2fffc47:/# ls /proc/1/root/
bin      etc      lib      mnt      proc      run      srv      tmp      var
dev      home     media    opt      root      sbin     sys      usr

root@20bac2fffc47:/# find /proc/1/root/ | grep php.ini
/proc/1/root/usr/local/etc/php/php.ini-production
/proc/1/root/usr/local/etc/php/php.ini-development
/proc/1/root/usr/local/include/php/main/php_ini.h
```



Solution

Solution 4: Use cntr.

Access network and processes, both container (/var/lib/cntr) and host (/) filesystems and all the tools installed on the host system.

```
● ● ● bash  
$ ngrep  
-bash: ngrep: command not found  
$ apt-get update  
$ apt-get install -y ngrep  
$ sudo cntr attach php  
root@docker-masters:/var/lib/cntr#
```



Container namespaces



bash

```
$ export PHP_PID=$(docker inspect -f '{{.State.Pid}}' php)
```

```
$ nsenter -h
```

Usage:

```
nsenter [options] <program> [<argument>...]
```

Run a program with namespaces of other processes.

Options:

-t, --target <pid>	target process to get namespaces from
-m, --mount[=<file>]	enter mount namespace
-u, --uts[=<file>]	enter UTS namespace (hostname etc)
-i, --ipc[=<file>]	enter System V IPC namespace
-n, --net[=<file>]	enter network namespace
-p, --pid[=<file>]	enter pid namespace
-c, --cgroup[=<file>]	enter cgroup namespace
-U, --user[=<file>]	enter user namespace
-S, --setuid <uid>	set uid in entered namespace



Container namespaces

```
● ● ● bash  
# Network (with host tools)  
$ sudo nsenter -t $PHP_PID -n ngrep -d eth0  
  
# Process  
$ sudo nsenter -t $PHP_PID -m ps aux  
  
# Mount (only container)  
$ sudo nsenter -t $PHP_PID -m ls /  
# Mount (both container and host filesystems and host tools)  
$ sudo ls -la /proc/${PHP_PID}/root/  
  
# Shell access  
$ sudo nsenter -t $PHP_PID -n -p -m -u -i sh
```



Command-line tools running in containers text processing and manipulation



Command-line tools running in containers monitoring and debugging processes



Command-line tools running in containers monitoring and debugging network



PID 1



PID 1

A process running as PID 1 inside a container is treated specially by Linux - it **ignores any signal with the default action.**

This means that the process will **not terminate** on SIGINT or SIGTERM or handle any other signal unless it is coded to do so.

The SIGKILL signal cannot be ignored by the process and it never gets the opportunity to catch the signal and act on it.



Process takeover

When a shell script aims to pass a control to another command or process it calls the exec command.

When you exec a command, it replaces the shell entirely, no new process is forked, no new PID is created, and all memory controlled by the shell is destroyed and overwritten.

The new process is therefore responsible for signal handling.

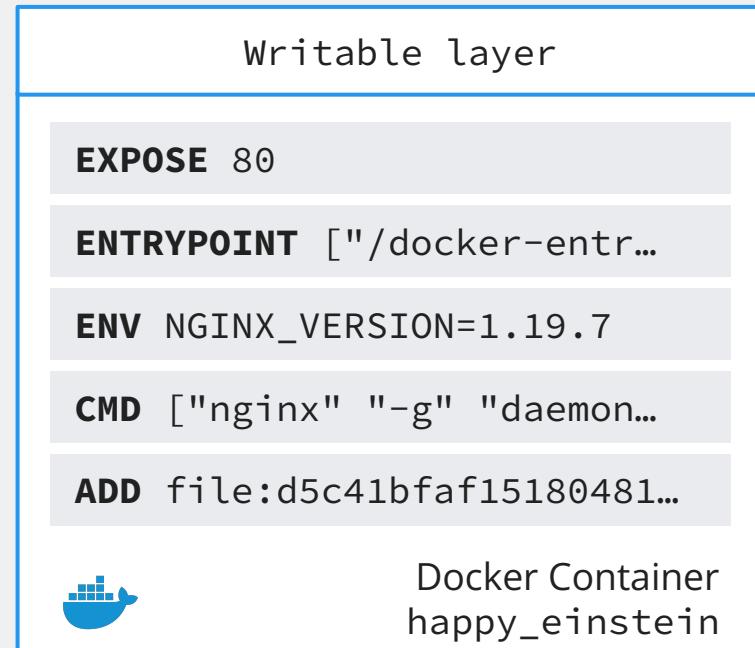
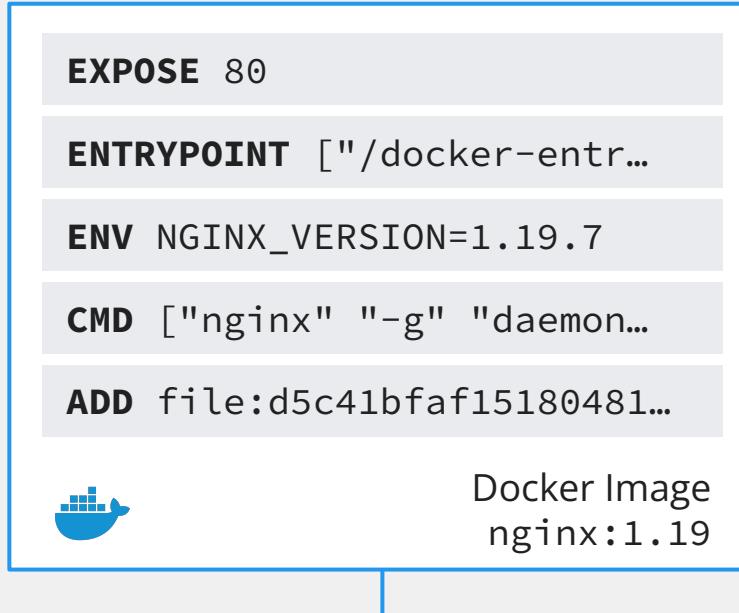
```
exec command [arguments ...]
```



Container filesystem



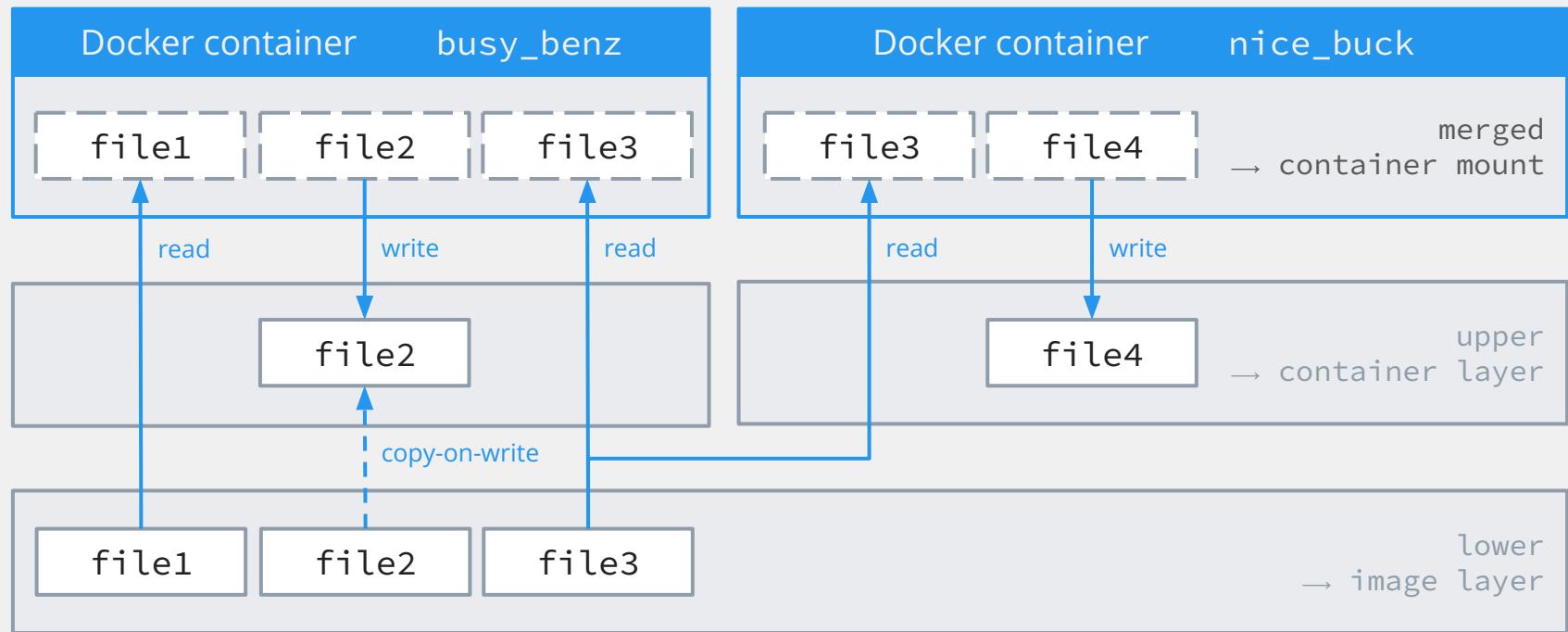
Docker image instance



docker run nginx:1.19



Container filesystem



Browse container filesystem

Browse the container filesystem directly by accessing the "merged" directory which gives a look at the root filesystem of a container being a combination of all layers.

```
● ● ● bash | State
$ docker inspect \
  --format '{{ json .GraphDriver.Data }}' nginx
{
  "LowerDir": "/var/lib/docker/overlay2/0dad272f8c6706fb...",
  "MergedDir": "/var/lib/docker/overlay2/0dad4d523351851...",
  "UpperDir": "/var/lib/docker/overlay2/0dad4d523351851a...",
  "WorkDir": "/var/lib/docker/overlay2/0dad4d5233518c2a0..."
}
```



Docker Engine metrics



Collecting and exploring Docker Engine metrics with Prometheus



CI/CD introduction



Docker in Continuous Integration (CI)



Dockerfile CI

One Dockerfile is needed to lint, analyze, install dependencies, test and containerize your project using a single docker build command.

This example PHP project Docker image is built only when all previous stages pass successfully.

```
FROM scratch AS project
COPY . /build

FROM jakzal/phpqa:php7.2 AS qa
COPY --from=project /build /build
WORKDIR /build
RUN phplint . && \
phpa .

FROM composer AS project-vendor
COPY --from=project /build /build
WORKDIR /build
RUN composer install

FROM phpunit/phpunit AS test
COPY --from=project-vendor /build /build
WORKDIR /build
RUN phpunit test/

FROM php:7.2-fpm-alpine AS release
COPY --from=project-vendor /build /var/www

FROM release AS release-dev
RUN apk add --no-cache --update --virtual buildDeps \
    autoconf g++ make && \
    pecl install xdebug && \
    docker-php-ext-enable xdebug && \
    apk del buildDeps

FROM release
```



Dockerfile CI

Run only the stages required for tests execution

```
$ docker build --target=test .
```

Build a developer version of the image

```
$ docker build --target=release-dev -t project:dev .
```

Build the production image

```
$ docker build -t project:latest .
```



Docker on GitLab.com



CI/CD

The continuous methodologies of software development are based on automating the execution of scripts to minimize the chance of introducing errors while developing applications.

It involves continuously building, testing, and deploying code changes at every small iteration, reducing the chance of developing new code based on bugged or failed previous versions.



.gitlab-ci.yml

GitLab CI/CD pipelines are configured using a YAML file called `.gitlab-ci.yml` within each project.

This file defines the structure and order of the pipelines and determines what to execute using GitLab Runner and what decisions to make when specific conditions are encountered.



.gitlab-ci.yml

stages is used to define steps that can be used by jobs and is defined globally.

Jobs of the same stage are run in parallel.

stages:

- build



.gitlab-ci.yml

Define a CI job of a defined stage,
script is the only required keyword that a job needs.

build-docker:

 stage: build

script:

 - docker build --no-cache -t application .





.gitlab-ci.yml

```
image: docker:20.10
services:
  - docker:20.10-dind

stages:
  - build

build:
  stage: build
  image: docker:20.10-dind
  script:
    - docker pull hello-world
    - docker run --rm hello-world
```



Commit your changes

Commit all changes and push them to the local repository.

```
bash | Automate  
$ git add .  
$ git commit -m "ci/cd"  
$ git push origin master  
Enumerating objects: 85, done.  
Counting objects: 100% (85/85), done.  
Delta compression using up to 4 threads  
Compressing objects: 100% (80/80), done.  
Writing objects: 100% (85/85), 108.25 KiB | 2.85 MiB/s, done.  
...
```



.gitlab-ci.yml

[artifacts](#) is used to specify a list of files and directories which should be attached to the job after success.

```
build:  
  stage: build  
  artifacts:  
    untracked: true
```



GitLab and GitLab Runner in Docker



Containerizing a static website



nginx in Docker

Nginx (pronounced "engine-x") is an open source reverse proxy server for HTTP, HTTPS, SMTP, POP3, and IMAP protocols, as well as a load balancer, HTTP cache, and a web server (origin server). The nginx project started with a strong focus on high concurrency, high performance and low memory usage.

Image

nginx - official Docker image
https://hub.docker.com/_/nginx

Ports

80 / tcp

Volumes

none

Other

- Lightweight Alpine version of the image.
- Image variant with Perl pre-installed.



Apache in Docker

The Apache HTTP Server, colloquially called Apache, is a Web server application notable for playing a key role in the initial growth of the World Wide Web. Originally based on the NCSA HTTPd server, development of Apache began in early 1995 after work on the NCSA code stalled.

Image

httpd - official Docker image
https://hub.docker.com/_/httpd

Ports

80 / tcp

Volumes

none

Other

- Lightweight Alpine version of the image.



ngrok

ngrok exposes local servers behind NATs and firewalls to the public internet over secure tunnels.

Client connects to the ngrok cloud service which accepts traffic on a public address and relays that traffic through to the ngrok process running in Docker and then on to the container and port specified.

<https://ngrok.com> | <https://github.com/wernight/docker-ngrok>



Cloud tunnel



ngrok cloud

6bf28057.ngrok.io (HTTP, HTTPS)



wernight/ngrok

tunnel to app:80 (HTTP)



app

serve on :8080 (HTTP)



```
3. docker run -it --net orca_default -p 4040:4040 wernight/ngrok http eu (docker)
× docker (docker) ⌘1
ngrok by @inconschreveable (Ctrl+C to quit)

Session Status          online
Session Expires         7 hours, 59 minutes
Update                  update available (version 2.3.29, Ctrl-U to update)
Version                 2.3.25
Region                  Europe (eu)
Web Interface           http://0.0.0.0:4040
Forwarding              http://550572fd.eu.ngrok.io -> http://orca:8080
Forwarding              https://550572fd.eu.ngrok.io -> http://orca:8080

Connections             ttl     opn      rt1      rt5      p50      p90
                        0       0       0.00    0.00    0.00    0.00
```



Working with extended CSS files in Docker - Sass and SCSS



Using Docker for static analysis and prettier of project files



Containerizing a React.JS project



Containerizing an Angular project



QA and Docker



Selenium in Docker



Selenium architecture



Source: laptrinhx.com



PlantUML in Docker

