Introducción a Containers

Container Engines - ECS

Mauricio Améndola / Sebastián Orrego – Profesor Adjunto
Escuela de Tecnología – Facultad de Ingeniería

UNIVERSIDAD ORT
Uruguay

AGENDA

1. El problema
2. Una posible solución
3. Qué es un container
4. Containers y DevOps
5. Container Engines
    a. Docker
    b. Podman
6. Construyendo imágenes

Una mirada al problema

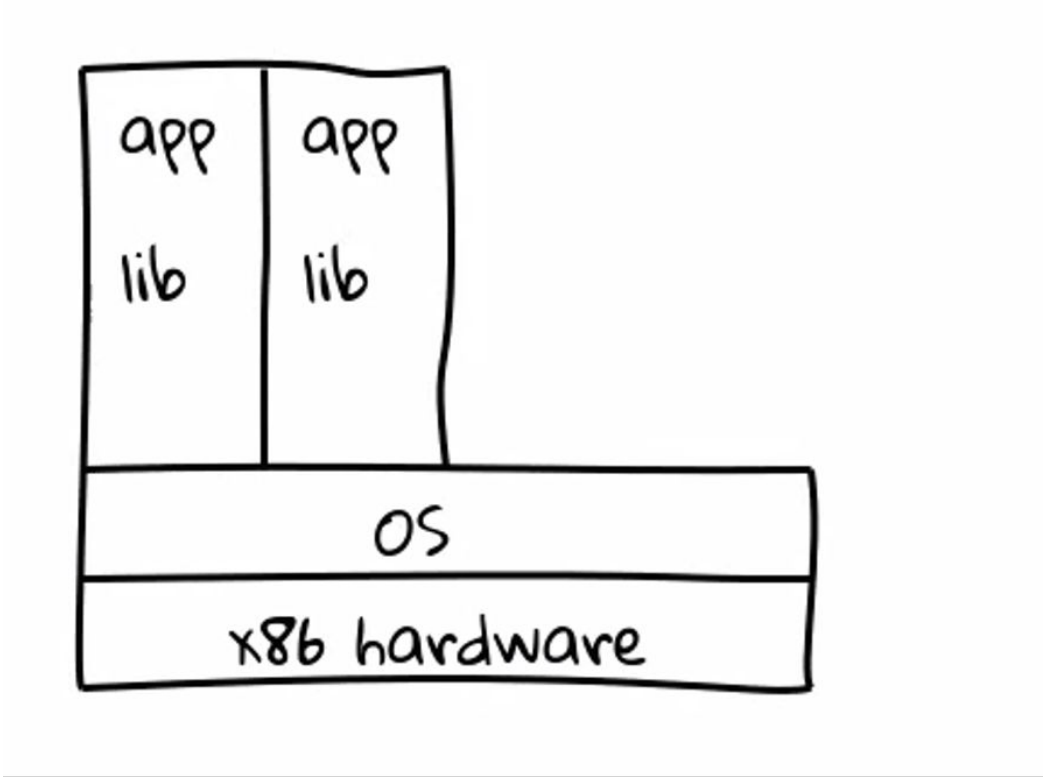
Una mirada al problema

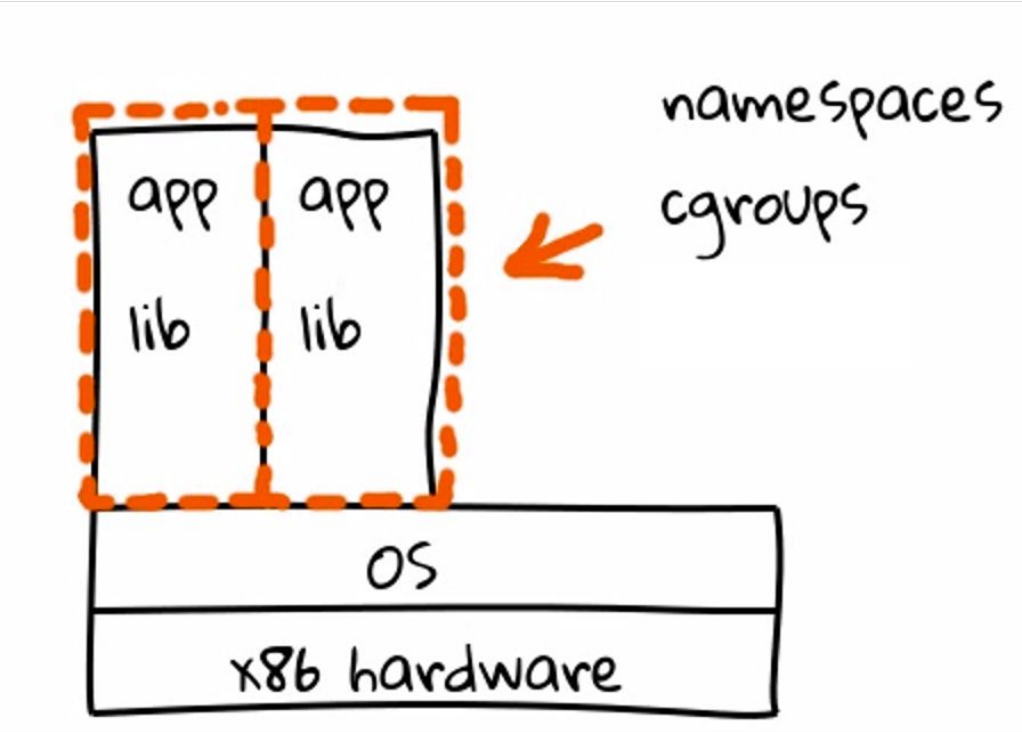
- ➔ Arquitecturas no escalables
- ➔ Aplicaciones modernas -> problemas complejos
- ➔ Necesidad de portabilidad
- ➔ Delivery -> time-to-market
- ➔ Aplicaciones más confiables / robustas
- ➔ Drift de ambientes
- ➔ DevOps KPI
  - ◆ Mean time to recovery
  - ◆ Mean time to failure

¿La solución?

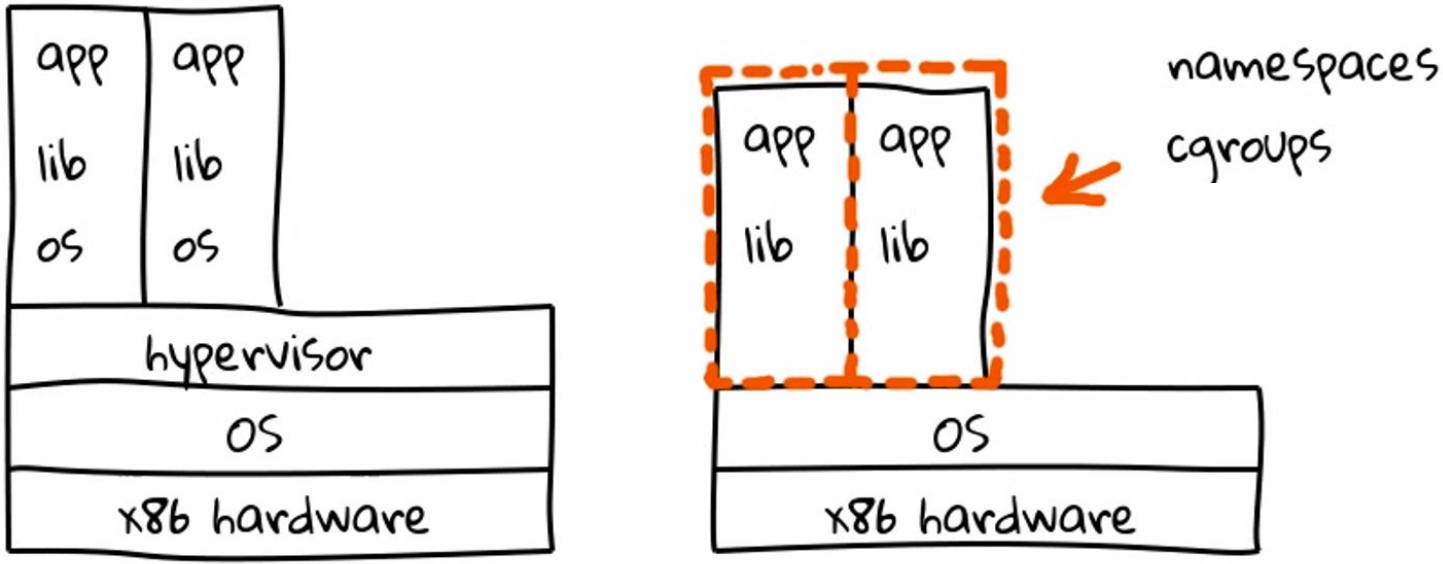

CONTAINERS...
CONTAINERS EVERYWHERE
imgflip.com

¿Qué es un Container?

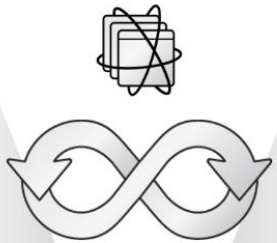

app
lib
app
lib
OS
x86 hardware


app
app
lib
lib
namespaces
cgroups
OS
x86 hardware

cgroups + namespaces + selinux = CONTAINERS


app
app
lib
lib
os
os
hypervisor
OS
x86 hardware
app
app
lib
lib
namespaces
cgroups
OS
x86 hardware

Containers para DevOps



DESARROLLO

- ➔ App empaquetada junto con sus dependencias
- ➔ Objetos transportables entre ambientes
- ➔ El MVP para los microservicios
- ➔ Simplemente Apps

OPERACIONES DE TI

- ➔ Sandboxed process
- ➔ Más livianos y densos que las VMs
- ➔ Portable entre ambientes
- ➔ Infraestructura as Code

Casos de Uso de Containers

Casos de Uso

- ➔ Aplicaciones desplegadas a partir de Código
- ➔ Portabilidad
- ➔ Infraestructura inmutable
- ➔ Aplicaciones dinámicas (versiones nuevas)
- ➔ Aplicaciones basadas en microservicios
- ➔ Recovery rápido / Tiempo promedio de recovery
- ➔ Ambientes de desarrollo, sandbox, etc
- ➔ Y mucho, mucho más...

Containers Engines

Container Engines

- ➔ Cloud / On-premises
  - ◆ Docker
  - ◆ Podman
  - ◆ Otros que ya pasaron de moda
- ➔ Cloud
  - ◆ AWS ECS
  - ◆ Azure Containers

Docker

Docker Engine

- ➔ Docker Server
- ➔ APIs
- ➔ Cliente de línea de comandos

Docker Server

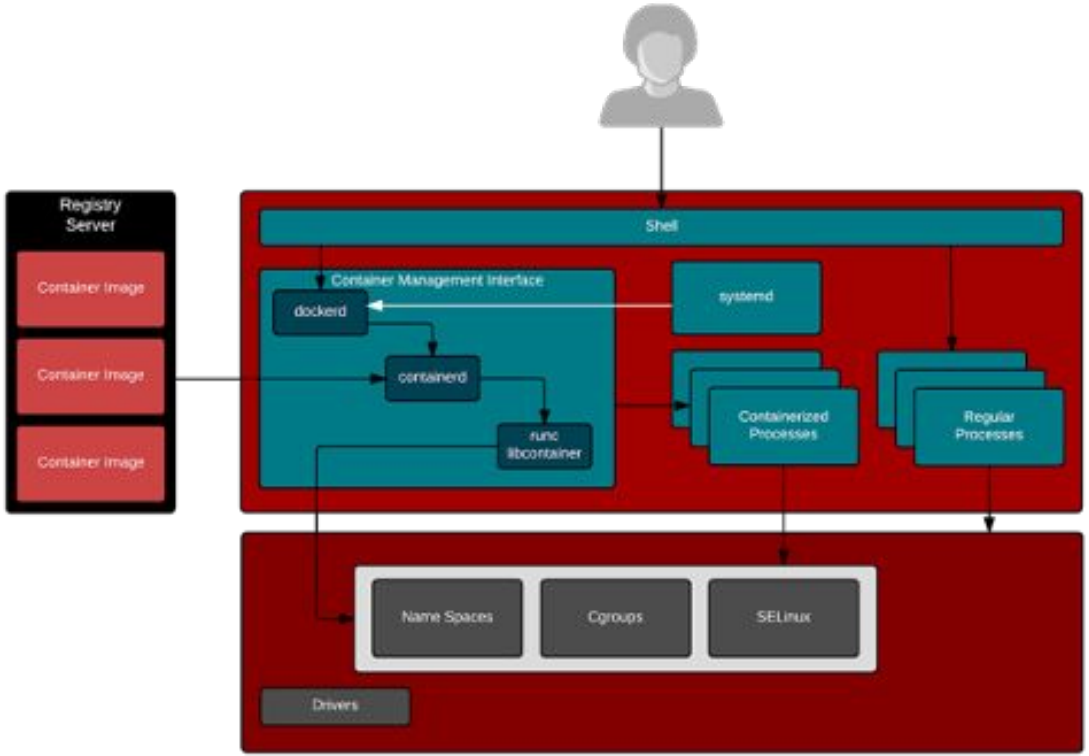
Proveé de un daemon llamado ***dockerd***

APIs

Interfaces APIs que pueden ser consumidas mediante scripting o CLI para instruír al docker daemon

Cli

Interfaz cliente de línea de comandos.



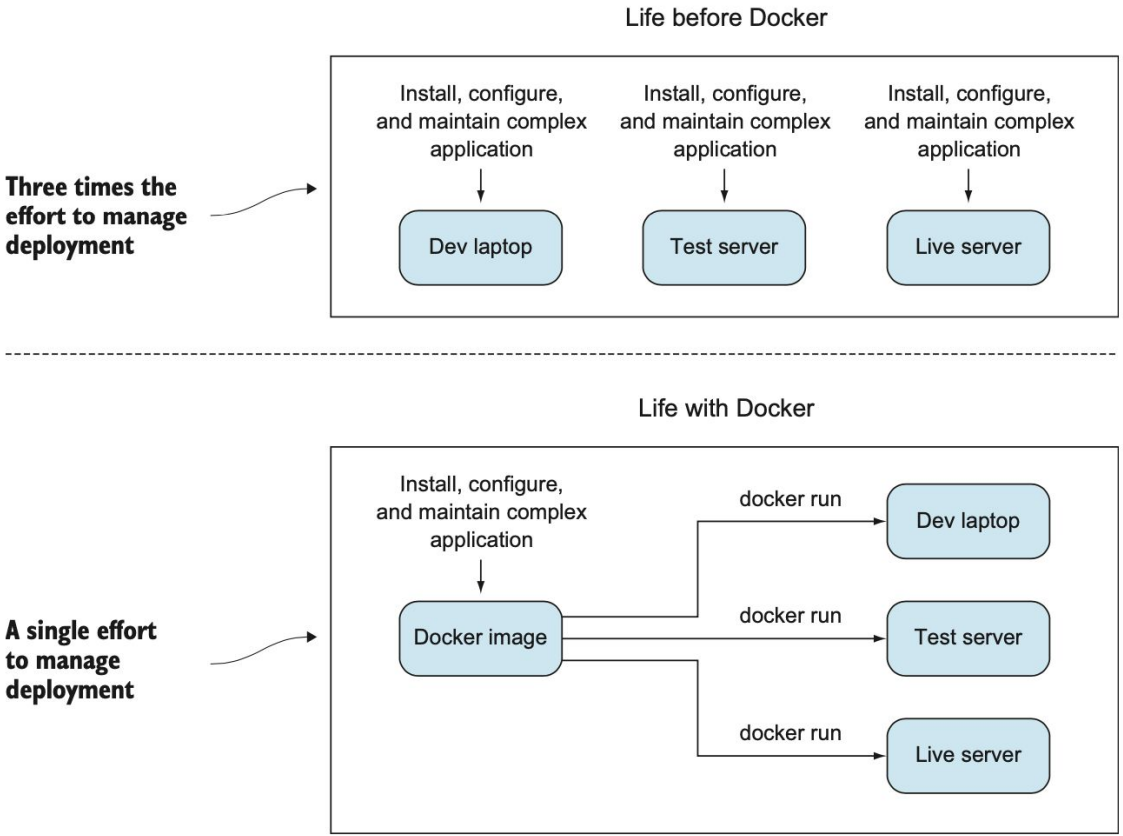
User
(Developer & SysAdmin)

User Space
(Processes & Libraries)

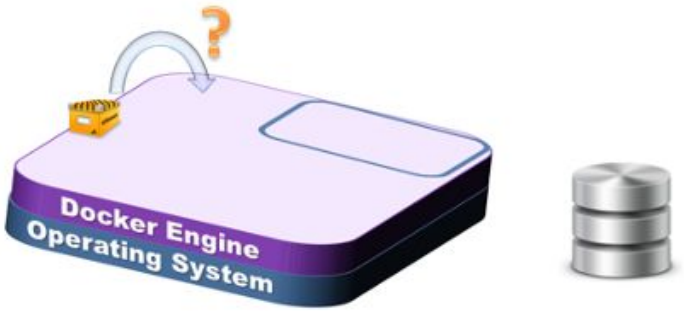
Kernel
(Data Structures)

Cómo era la vida antes de Docker?


IT WORKS ON MY MACHINE
THEN WE'LL SHIP YOUR MACHINE
AND THAT IS HOW DOCKER WAS BORN

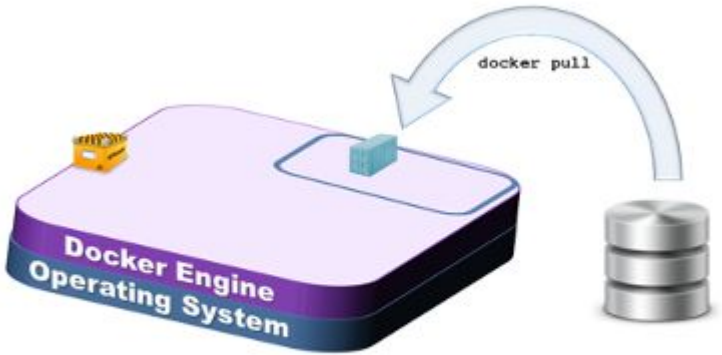

Life before Docker
Three times the effort to manage deployment
Install, configure, and maintain complex application
Install, configure, and maintain complex application
Install, configure, and maintain complex application
Dev laptop
Test server
Live server
Life with Docker
A single effort to manage deployment
Install, configure, and maintain complex application
Docker image
docker run
docker run
docker run
Dev laptop
Test server
Live server

Cómo funciona Docker?



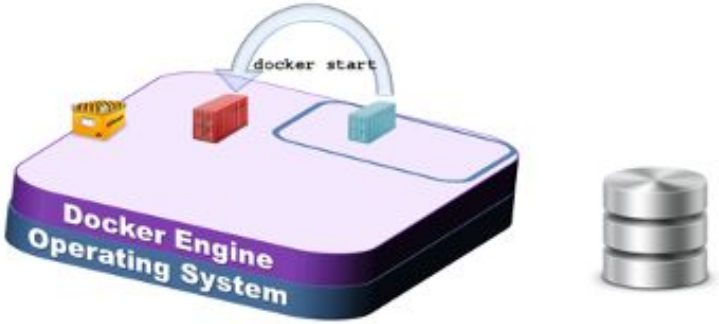
$ docker run -d -p 8080:80 httpd

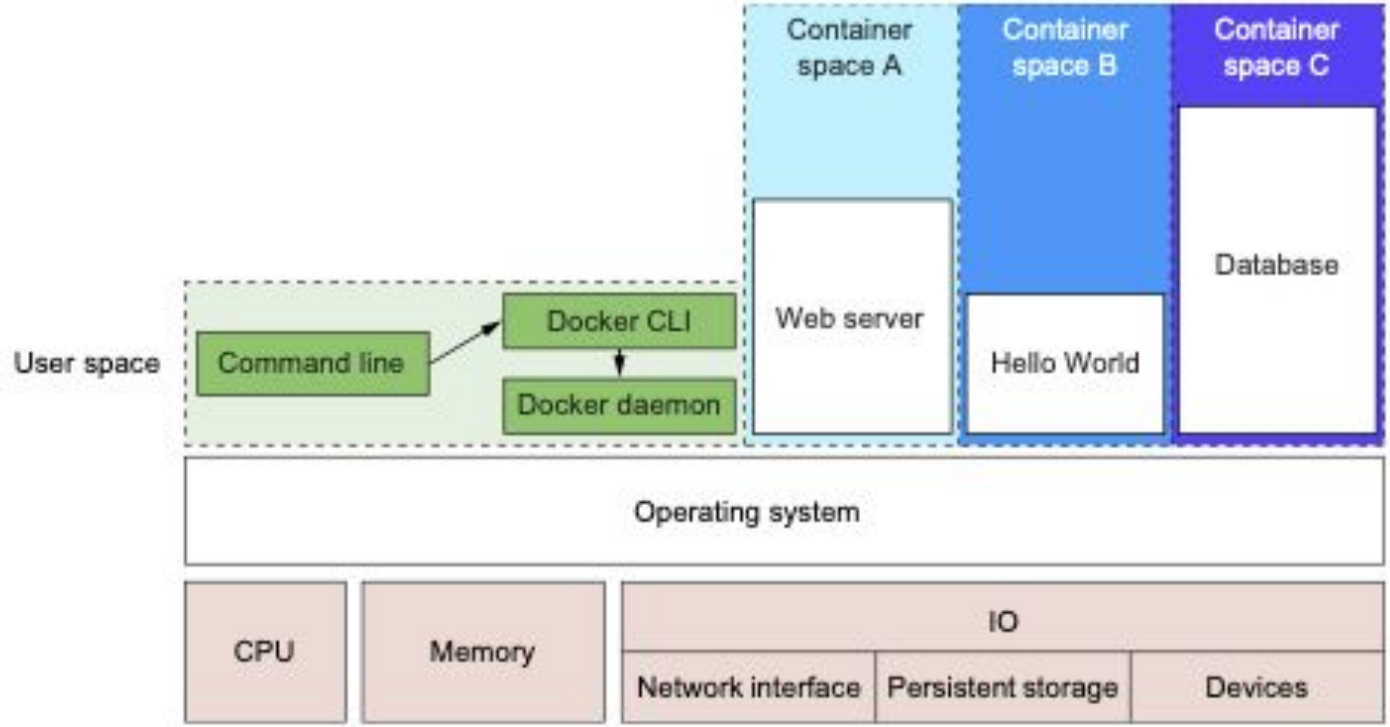
Si la imagen (httpd) no existe localmente, el daemon la va a buscar a una container Registry



Si la imagen (httpd) no existe localmente, el daemon la va a buscar a una container Registry

Luego de la descarga, se inicia el contenedor




Container space A
Container space B
Container space C
Database
Web server
Hello World
User space
Command line
Docker CLI
Docker daemon
Operating system
CPU
Memory
IO
Network interface
Persistent storage
Devices

Networking en Docker

Networking en Docker

Existen 5 drivers por default:

- ➔ Default docker bridge / User defined bridge
- ➔ Overlay Network
- ➔ Host only Network
- ➔ Macvlan Network
- ➔ None

Networking en Docker

➔ Default docker bridge / User defined bridge

Es el driver por default. Establece una interfaz de bridge que comunica al servicio dentro del container con la interfaz física del host donde corre dicho container.

Networking en Docker

➔ Overlay Network

Sirve para conectar dos o más containers que están en distintos hosts, ya sea que estos containers corren en standalone o bajo el servicio de Swarm (orquestador de container de Docker Inc.)

Networking en Docker

➔ Host only

Se elimina por completo la aislación del networking entre el container y el host, por lo que se consume directamente el servicio dentro del container, usando la interfaz física del host donde corre.

Networking en Docker

➔ Macvlan Network

Permite asignarle una dirección MAC a un container haciendo que el docker daemon rutee el tráfico a través de la MAC address. Este driver es comúnmente usado en aplicaciones legacy o que necesitan simular una interfaz física.

Networking en Docker

➔ None

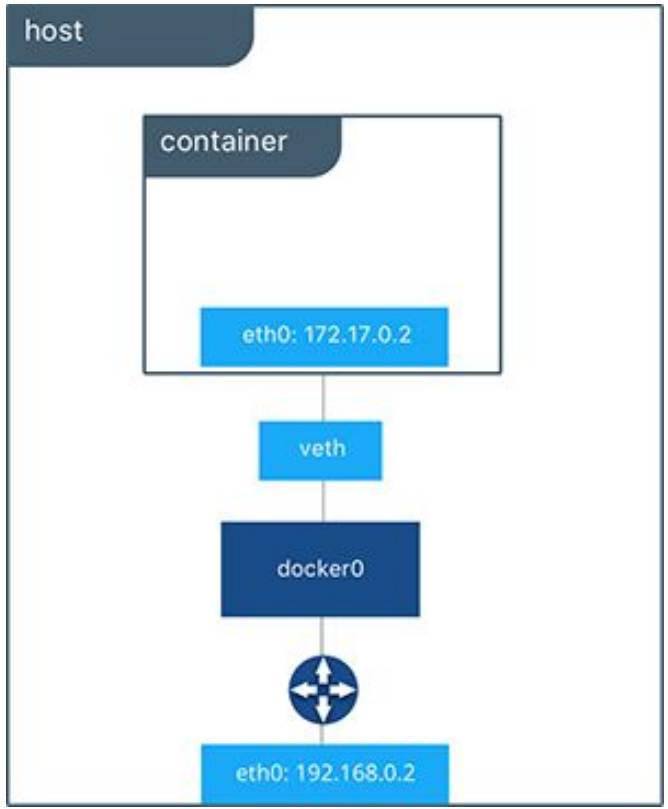
Deshabilita por completo el networking en un container. Dicho container corre aislado y sin posibilidad de consumir el servicio.

Networking en Docker

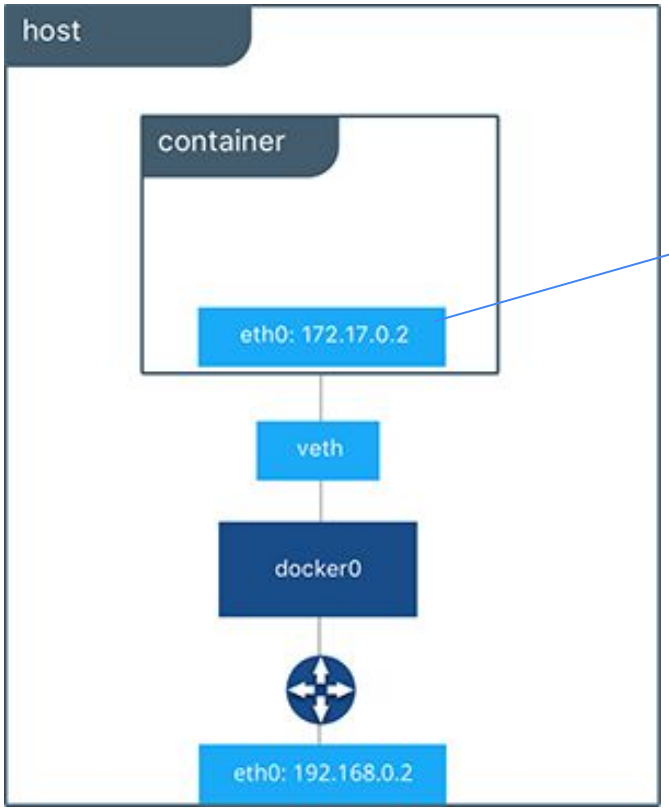
Existen 5 drivers por default:

- ➔ Default docker bridge / User defined bridge
- ➔ Overlay Network
- ➔ Host only Network
- ➔ Macvlan Network
- ➔ None

Networking en Docker

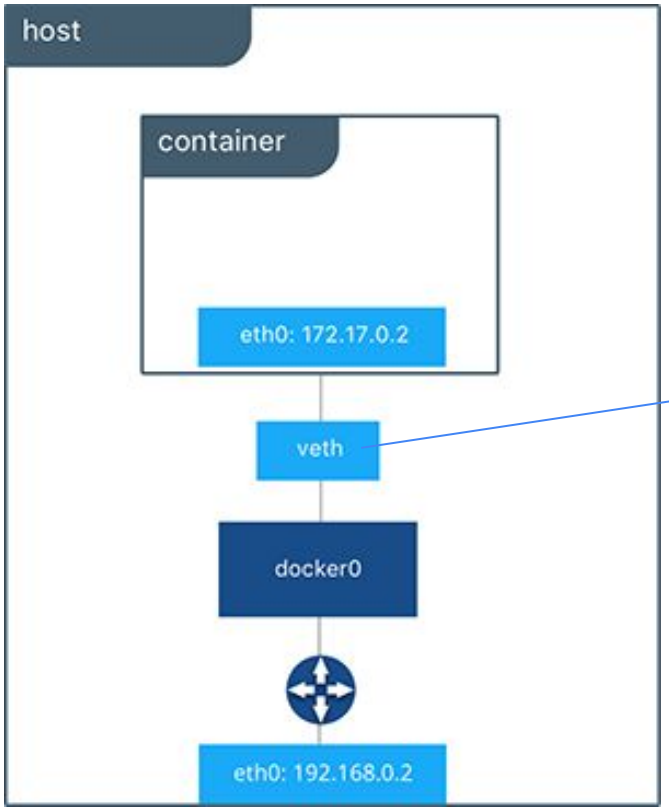


Networking en Docker



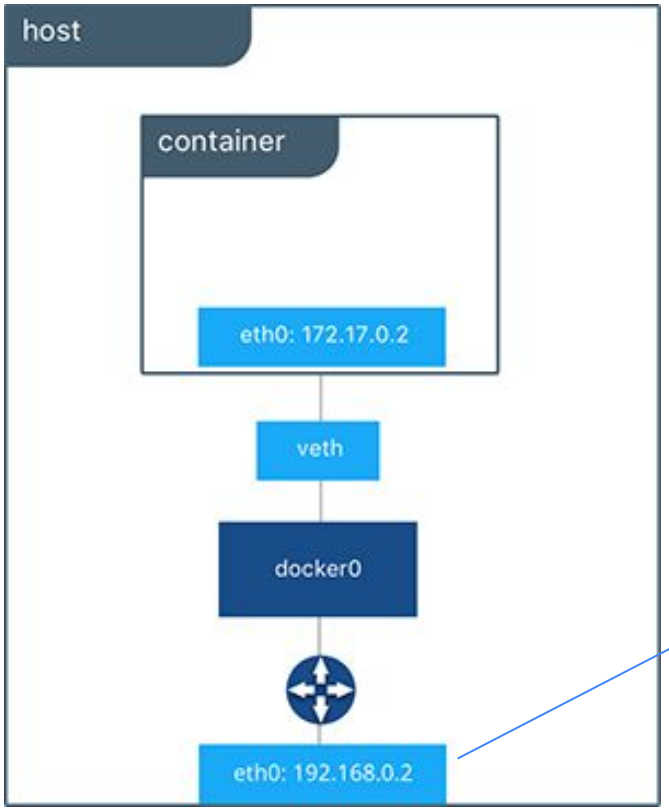
Dirección IP asignada automáticamente

Networking en Docker



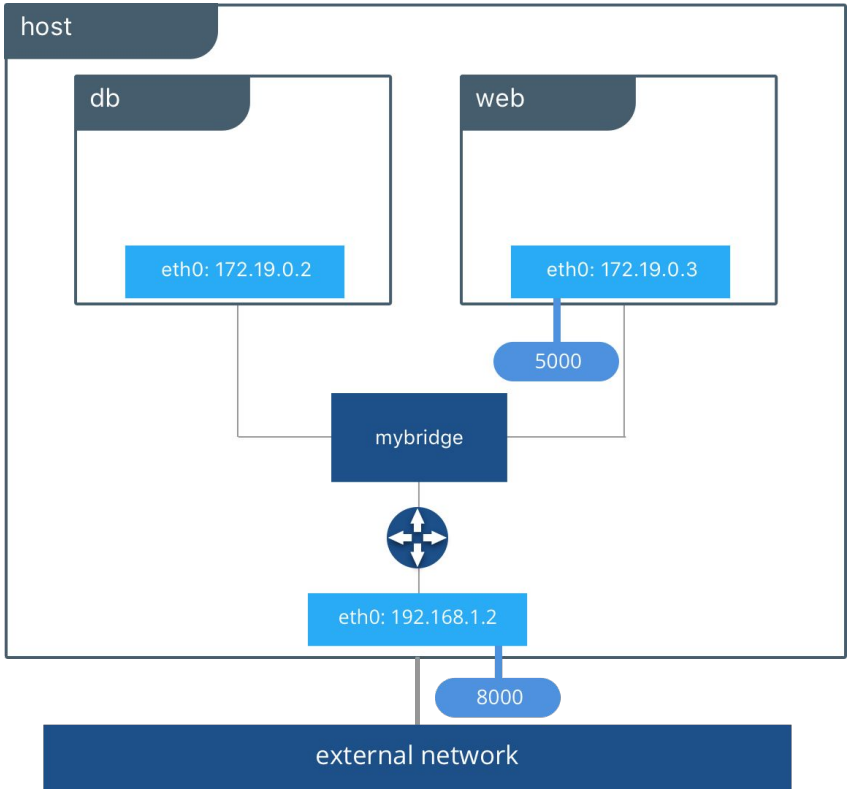
Interfaz virtual que conecta la interfaz del container con la interfaz de bridge.

Networking en Docker



Interfaz “física” del host donde corre Docker.

Networking en Docker



Volúmenes en Docker

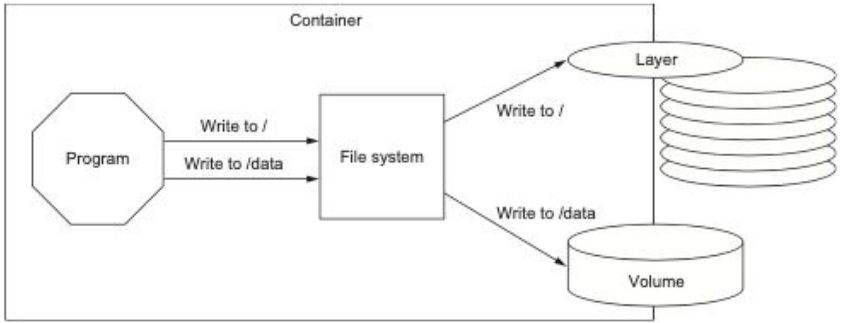
Volúmenes en Docker

Hasta ahora hemos visto cómo instalar y ejecutar programas. Los programas trabajan con datos (bases de datos, logs, archivos, etc). ¿Cómo nos aseguramos que esos datos sobrevivan a la ejecución de un container?

Volúmenes en Docker

¿Qué son?

Puntos de montaje en el árbol de directorios del contenedor donde se monta una porción del árbol de directorios del host



Volúmenes en Docker

¿Qué son?

Conceptualmente son una herramienta para segmentar y compartir datos cuyo alcance sea independiente al de un único contenedor.

- ➔ Db software vs. Db data
- ➔ App Server vs code (o logs)
- ➔ Procesamiento de datos vs. datos entrada salida
- ➔ etc.

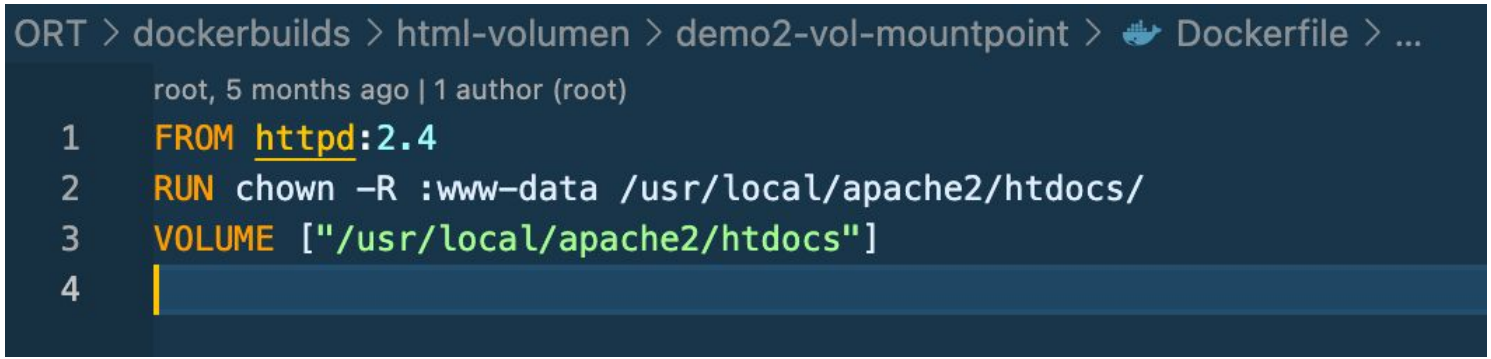
Volúmenes en Docker

Hay varias formas de presentarle un volumen a un container y en particular analizaremos dos de ellas:

- ➔ Usando la instrucción VOLUME en un archivo *Dockerfile*
- ➔ Usando el flag *-v* en tiempo de ejecución

Volúmenes en Docker

➔ Usando la instrucción VOLUME en un archivo *Dockerfile*



Volúmenes en Docker

➔ Usando la instrucción VOLUME en un archivo *Dockerfile*

```
[root@ip-172-31-78-62 demo1-vol-default]# docker volume ls
DRIVER    VOLUME NAME
local     d29dac67ccfe6c9cce8dc4cc0830a96a12d4fb6d0c74a4105f3ebfc53112415e
[root@ip-172-31-78-62 demo1-vol-default]# docker inspect d29dac67ccfe6c9cce8dc4cc0830a96a12d4fb6d0c74a4105f3ebfc53112415e
[
    {
        "CreatedAt": "2021-05-11T14:18:02Z",
        "Driver": "local",
        "Labels": null,
        "Mountpoint": "/var/lib/docker/volumes/d29dac67ccfe6c9cce8dc4cc0830a96a12d4fb6d0c74a4105f3ebfc53112415e/_data",
        "Name": "d29dac67ccfe6c9cce8dc4cc0830a96a12d4fb6d0c74a4105f3ebfc53112415e",
        "Options": null,
        "Scope": "local"
    }
]
[root@ip-172-31-78-62 demo1-vol-default]# tree /var/lib/docker/volumes/d29dac67ccfe6c9cce8dc4cc0830a96a12d4fb6d0c74a4105f3ebfc53112415e/_data
/var/lib/docker/volumes/d29dac67ccfe6c9cce8dc4cc0830a96a12d4fb6d0c74a4105f3ebfc53112415e/_data
└── index.html

0 directories, 1 file
[root@ip-172-31-78-62 demo1-vol-default]#
```

Volúmenes en Docker

➔ Usando el flag *-v* en tiempo de ejecución para montar un directorio / volumen del Host

```
[root@ip-172-31-78-62 demo2-vol-mountpoint]# docker run -d -v ~/ort-dockerbuilds/html-volumen/demo2-vol-mountpoint/app:/usr/local/apache2/htdocs -p 8080:80 webvol:modo2
dd6ce3e36e8fe3472a6ae69b6ff932382e9f47ce3f84df6c4a1c830808a7720d
[root@ip-172-31-78-62 demo2-vol-mountpoint]# docker volume ls
DRIVER    VOLUME NAME
local     d29dac67ccfe6c9cce8dc4cc0830a96a12d4fb6d0c74a4105f3ebfc53112415e
```

Práctico 1: Networking
Práctico 2: Volúmenes

PODMAN

PODMAN

Podman es un container Engine desarrollado principalmente por Red Hat junto a otros actores de la comunidad Open Source. Es producto del aprendizaje de los últimos años en el uso de tecnologías de Containers.

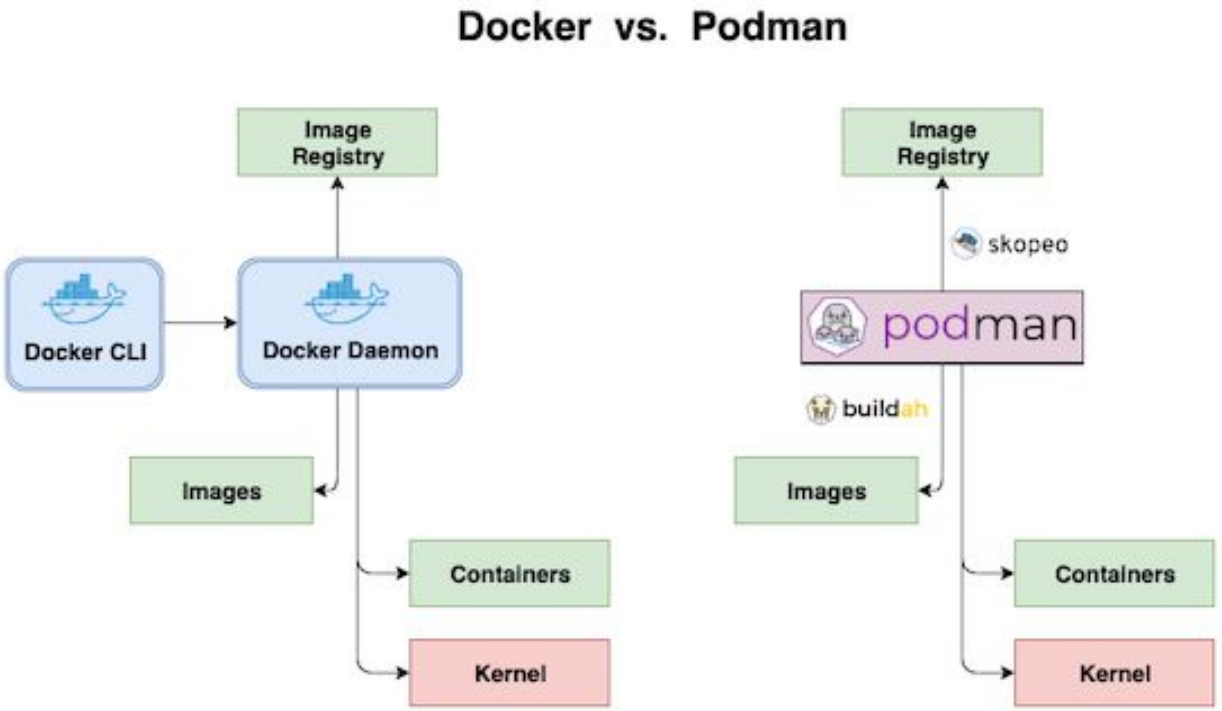
Esto no quiere decir que es mejor que Docker, aunque sí incluye algunas mejoras respecto a este. De hecho, lo aprendido en Docker, aplica a Podman, son prácticamente los mismos comandos.

PODMAN: Principales características

A diferencia de Docker...

- ➔ Es sin agente, es decir, no hay un Daemon del sistema
- ➔ Es rootless. A diferencia de Docker que el daemon requiere ser ejecutado por *root*.
- ➔ Solo corre en Linux (por ahora)
- ➔ ***Usa un modelo fork-exec, por lo que el container no depende de ningún proceso padre. En Docker, el modelo client-server implica que cada container depende del daemon, si este se cae, se terminan todos los containers.***

PODMAN: Principales características



PODMAN para Docker users

PODMAN para Docker users

- ➔ Los comandos que ya conocen de Docker, funcionan con Podman con el mismo resultado.
- ➔ Para la mayoría de los usuarios, basta con hacer un alias a docker. Incluso hay un paquete que se instala y hace eso.
- ➔ El networking ofrece las mismas características. Adicionalmente, agrega un plugin para containers rootless (que no tienen permisos para crear interfaces a nivel del host)
  - ◆ Bridge
  - ◆ Macvlan
  - ◆ Slirp4netns (similar al plugin Host Only de Docker)

Building Images

Building Images

Existen varias herramientas para crear imágenes en Docker, entre ellas:

- ➔ Dockerfile -> Docker build
- ➔ Docker-compose
- ➔ Podman (si, podman también buildea)
- ➔ Buildah
- ➔ Hashicorp Packer
- ➔ Cambio manual en un container -> Docker commit

Building Images

Existen varias herramientas para crear imágenes en Docker, entre ellas:

- ➔ Dockerfile -> Docker build
- ➔ Docker-compose
- ➔ Podman (si, podman también buildea)
- ➔ Buildah
- ➔ Hashicorp Packer
- ➔ Cambio manual en un container -> Docker commit

Building Images

Uno de los mecanismos más sencillos para crear imágenes es Dockerfile. Un archivo que contiene una serie de instrucciones para ensamblar la imagen.
Este mecanismo es el adecuado para iniciar el proceso de contenerización de una aplicación.

Building Images

Building Images

- ➔ **FROM**: instrucción para indicar cual es el repositorio e imagen base para mi nueva imagen
- ➔ **RUN**: instrucción que usaremos para instalar paquetes o ejecutar operaciones dentro de la imagen. RUN agrega una nueva capa a la imagen base.
- ➔ **WORKDIR**: Instrucción que se usa para posicionarse en un directorio
- ➔ **EXPOSE**: Indica en qué puerto/s el container escucha por nuevas conexiones. Típicamente es el puerto de la aplicación.
- ➔ **CMD**: Ejecuta un comando shell en el container. Este comando es el que se ejecuta por omisión cuando se instancia el container.

Building Images

Algunas consideraciones:

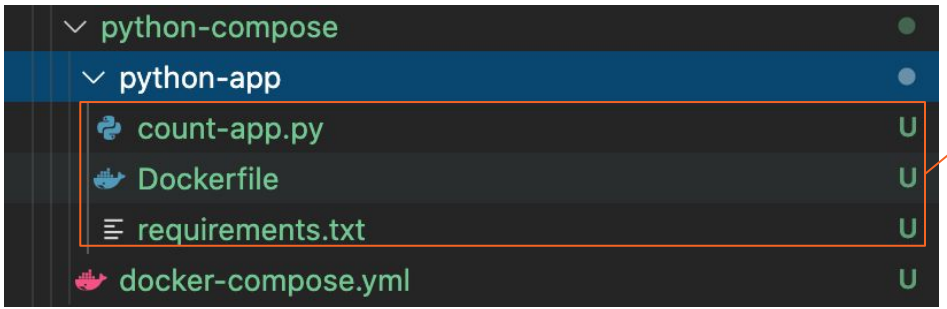
- ➔ Tratemos de generar la menor cantidad de capas. Por ejemplo, podemos agrupar las instrucciones ***RUN*** en la medida de lo posible.
- ➔ Evitemos copiar archivos grandes dentro de la imagen.
- ➔ Si instalamos paquetes / dependencias, se sugiere limpiar post-instalación.
- ➔ Tenemos que tener claro cómo “inicia” el servicio dentro del container o que proceso debe ejecutar al instanciar el container. De eso dependerá la instrucción a usar (CMD o ENTRYPOINT).

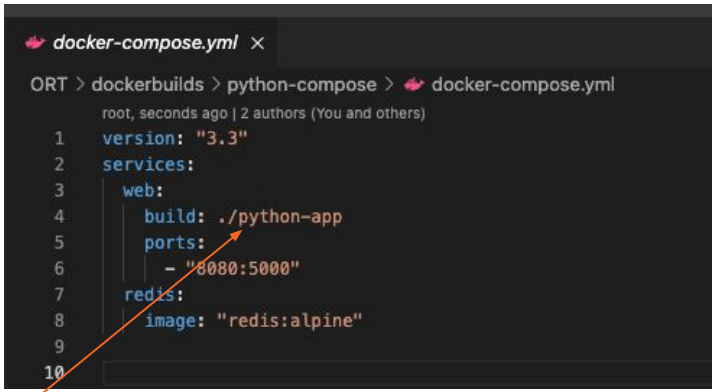
Building Images

Docker stack / Docker Compose

- ➔ Es una forma sencilla de levantar containers que tienen dependencias entre sí
- ➔ También es una forma de levantar varios containers a la vez
- ➔ Ambos soportan un “compose file” (docker-compose.yml)
- ➔ docker stack viene con la instalación de Docker
- ➔ docker-compose se instala cómo adicional (yum install docker-compose)
- ➔ docker stack no soporta buildear imágenes
- ➔ docker stack necesita el “modo” swarm en on

Building Images: docker-compose





Building Images: docker-compose

```
[root@ip-172-31-78-62 python-compose]# tree
.
├── docker-compose.yml
└── python-app
    ├── count-app.py
    ├── Dockerfile
    └── requirements.txt

1 directory, 4 files
[root@ip-172-31-78-62 python-compose]#
```

Ejecutamos: docker-compose up

```
Creating pythoncompose_web_1   ... done
Creating pythoncompose_redis_1 ... done
Attaching to pythoncompose_web_1, pythoncompose_redis_1
redis_1  | 1:C 11 May 2021 15:29:48.287 # oO0OoO0OoO0Oo Redis is starting oO0OoO0OoO0Oo
redis_1  | 1:C 11 May 2021 15:29:48.287 # Redis version=6.2.3, bits=64, commit=00000000, modified=0, pid=1, just started
redis_1  | 1:C 11 May 2021 15:29:48.287 # Warning: no config file specified, using the default config. In order to specify a config file use redis-server /path/to/re
redis_1  | 1:M 11 May 2021 15:29:48.287 * monotonic clock: POSIX clock_gettime
redis_1  | 1:M 11 May 2021 15:29:48.288 * Running mode=standalone, port=6379.
redis_1  | 1:M 11 May 2021 15:29:48.288 # WARNING: The TCP backlog setting of 511 cannot be enforced because /proc/sys/net/core/somaxconn is set to the lower value o
redis_1  | 1:M 11 May 2021 15:29:48.288 # Server initialized
redis_1  | 1:M 11 May 2021 15:29:48.288 # WARNING overcommit_memory is set to 0! Background save may fail under low memory condition. To fix this issue add 'vm.overc
d 'sysctl vm.overcommit_memory=1' for this to take effect.
redis_1  | 1:M 11 May 2021 15:29:48.288 * Ready to accept connections
web_1    |  * Serving Flask app "count-app.py"
web_1    |  * Environment: production
web_1    |    WARNING: This is a development server. Do not use it in a production deployment.
web_1    |    Use a production WSGI server instead.
web_1    |  * Debug mode: off
web_1    |  * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

Building Images: Buildah

De la misma forma que Podman, Buildah adopta lo bueno que tiene Docker en el rubro de construir imágenes y agrega algunas características que extienden las capacidades.

Entre las características más notables, destaca el hecho de que no necesitamos un Daemon para construir la imagen.

Building Images: Buildah

Con buildah podemos crear imágenes realmente muy pequeñas ya que nos permite crear un container vacío e ir agregando los componentes necesarios. Este container “vacío” (from scratch) no se basa en ninguna imagen ya existente y por ese solo hecho ya es más liviano.