

# Docker

<https://app.pluralsight.com/paths/skill/docker-fundamentals-for-developers>

Docker --version

Docker image build

Docker image push

Docker container ls

Comando para crear una imagen en nuestra maquina es:

```
docker image build -t freivincampbell/gsd:first-ctr .
```

Para publicar nuestra image publicamente

```
docker image push freivincampbell/gsd:first-ctr
```

Running a containerized app

```
docker container run -d --name web -p 8000:8080
```

```
docker container run -d --name web -p 8000:8080 \
```

Enter

```
freivincampbell/gsd:first-ctr
```

Para eliminar imágenes locales

```
Docker image rm freivincampbell/gsd:first-ctr
```

Docker container stop "name"

Para eliminar el contenedor

```
Docker container rm "name"
```

```
Docker container run -d
```

```
Docker container run -it
```

Para listar los contenedores

```
Docker container ls
```

Multi container  
Docker-compose up

## Building and Running Your First Docker App

Para listar contenedores aunque no estén corriendo  
Docker ps -a

Docker file is a text document where you can set all commands to build images/  
dockers

Docker pull <image name>

Docker run -p <externalport>:<internalport> para correr el contenedor en el  
puerto que desde local host pueda escuchar

Docker network create --driver bridge <name>  
Docker network ls  
Docker network rm

Docker run -d --net=<name> --name=mongodb mongo

Docker exec -it <name> sh

Docker-compose => define services using a yaml configuration file

Docker-compose build  
Docker-compose up  
Docker-compose down

## Using Docker compose commands

Docker-compose build

## Building and Orchestrating Containers with Docker Compose

- YAML fundamental
- Create a docker compose file

The Role of docker compose

Docker images - define the contents that are need to run a container

Docker container - runs your application

Start, stop and rebuild services

View the status of running services

Stream the log output of running services

Run a one-off command on a service

## Docker compose workflow

-> build services => start up services => tear down services

### Key docker commands

Docker-compose build

Docker-compose up

Docker-compose down

## create a docker-compose.yml

Important key values

Build - environment - image - networks - port - volumes

We can use

Docker compose build - docker compose up - docker compose down without using '-'


Context = where to look with the YML file

Dockerfile = we can name it as we can

Args = ENV variables

Image = image that we are gonna use to build

### Key Docker Compose Commands



```
docker-compose --help
docker-compose build
docker-compose up
docker-compose up -d
docker-compose up -d --no-deps [service]
docker-compose down
docker-compose ps
docker-compose stop [service]
docker-compose start [service]
```

To see logs you can run `docker compose logs`

To run the sell

`Docker exec -it CONTAINERID sh`

Scale containers

`Docker compose up -d --scale api=4` with that you can not set the ports / also we need to configure the reply and replicas segment

Also we need to change the name to avoid conflicts

## Developing Docker Apps: Core Principles

## Container Image

### Dockerfile

```
FROM node:14

# Create app directory
WORKDIR /app

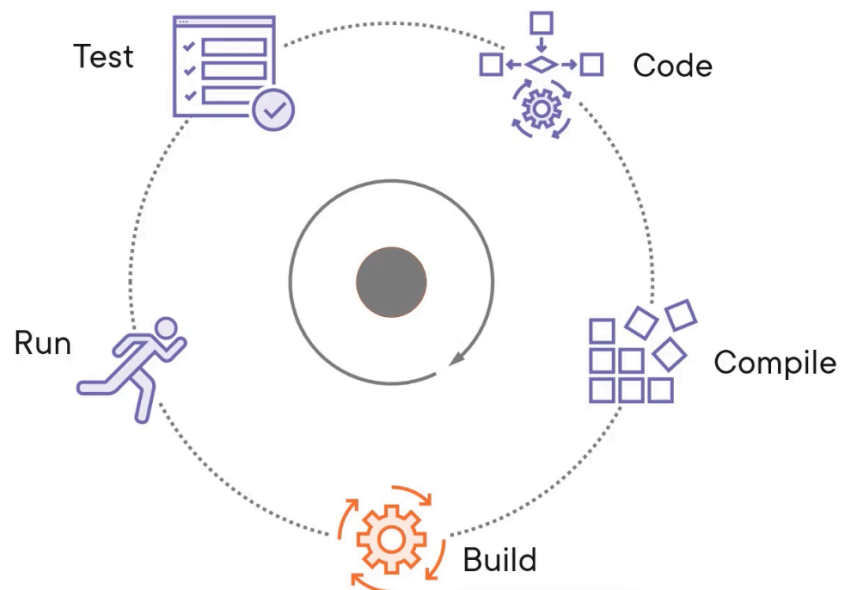
# Copy app source from build context
COPY . .

# Install app dependencies
RUN npm install

# Port app listens on
EXPOSE 3000

# Specify container's default command
CMD [ "node", "src/index.js" ]
```

## The Inner Loop with Containers



# Volume Types



**Tmpfs mount; used to store sensitive data.**

## Temporary storage

- Data stored in memory



**Named or anonymous volume; managed by Docker.**

## Volumes managed by Docker

- Managed using Docker CLI



**Bind mount; arbitrary directory mounted from host.**

## Mounts a specific directory

- Changes reflected on host

Docker volume create volume

\$ docker run --volume code-volume:/app-code

\$ docker volume ls

# Building an Image for an Application

## Build context

```
.
├── Dockerfile
├── package.json
├── spec [...]
├── src
│   ├── index.js
│   ├── persistence [...]
│   ├── routes [...]
│   ├── static
│   │   ├── css [...]
│   │   ├── index.html
│   │   └── js [...]
└──
```

## Dockerfile

```
FROM node:14

# Create app directory
WORKDIR /app

# Copy app source from build context
COPY . .

# Install app dependencies
RUN npm install && npm install -g nodemon

# Port app listens on
EXPOSE 3000

# Specify container's default command
CMD [ "node", "src/index.js" ]
```

## Multi stage builds

### Copying Artifacts Between Stages

#### Dockerfile

```
FROM node:14 AS builder
WORKDIR /deps
COPY . .
RUN npm install

FROM gcr.io/distroless/nodejs
COPY --from=builder /deps /app
WORKDIR /app
CMD ["server.js"]
```

### Common Base

#### Dockerfile

```
FROM golang:1.16 AS base
FROM base AS lint
<snip>
FROM base AS build
<snip>
FROM alpine:3
<snip>
ENTRYPOINT ["/mini"]
```

In this case we don't need to update the version twice

```
# Base stage
FROM golang:1.16 AS base

# Lint stage
FROM base AS lint
COPY golangci-lint /go/bin/
WORKDIR /app
CMD ["golangci-lint", "run"]

# Build stage
FROM base AS build
WORKDIR /app
COPY go.??? ./
RUN go mod download
COPY *.go ./
RUN go build -o mini .

# Execution stage
FROM alpine:3
COPY --from=build /app/mini /
ENTRYPOINT ["/mini"]
```

◀ Base stage for sharing common base image

◀ Lint stage for running linter against source code bind mounted into a derived container

◀ Build stage for fetching the dependencies and compiling the app's binary

◀ Execution stage for copying binary from build stage and executing app with a minimal image

```
$ docker build -t mini-lint:1.0 --target lint .
```

Building an Image for the Lint Stage

An image used only for linting can be built using the '--target' option

## Deploying Containerized Applications



# Platzi DOCKER

Problemas al construir:

Dependencias de desarrollo  
Versiones de entornos de ejecución  
Equivalencia de entornos de desarrollo  
Equivalencia de entornos de producción  
Versions / compatibilidad

Problemas al Distribuir:

Generaciones del build diferentes  
Acceso a servidores de producción  
Ejecución nativa vs la distribuida  
Serverless

Problemas al Ejecutar

Dependencias de aplicación  
Compatibilidad de sistema operativo  
Disponibilidad de servicios externos  
Recursos de hardware

Docker permite:

Construir, distribuir y ejecutar tu código en cualquier lado

Contenedores

- Flexibles
- livianos
- Portables
- Bajo acoplamiento
- Escalables
- Seguros

Componentes DENTRO del círculo de Docker:

- Docker daemon: Es el centro de docker, el corazón que gracias a él, podemos comunicarnos con los servicios de docker.
- REST API: Como cualquier otra API, es la que nos permite visualizar

docker de forma "gráfica".

- Cliente de docker: Gracias a este componente, podemos comunicarnos con el corazón de docker (Docker Daemon) que por defecto es la línea de comandos.

Dentro de la arquitectura de Docker encontramos:

1. Contenedores: Es la razón de ser de Docker, es donde podemos encapsular nuestras imágenes para llevarlas a otra computadora, o servidor, etc.
2. Imágenes: Son las encapsulaciones de x contenedor. Podemos correr nuestra aplicación en Java por medio de una imagen, podemos utilizar Ubuntu para correr nuestro proyecto, etc.
3. Volúmenes de datos: Podemos acceder con seguridad al sistema de archivos de nuestra máquina.
4. Redes: Son las que permiten la comunicación entre contenedores.

---

## Comandos

```
$ docker run hello-world (corro el contenedor hello-world)
```

```
$ docker ps (muestra los contenedores activos)
```

```
$ docker ps -a (muestra todos los contenedores)
```

```
$ docker inspect <containe ID> (muestra el detalle completo de un contenedor)
```

```
$ docker inspect <name> (igual que el anterior pero invocado con el nombre)
```

```
$ docker run --name hello-platzi hello-world (le asigno un nombre custom "hello-platzi")
```

```
$ docker rename hello-platzi hola-platzy (cambio el nombre de hello-platzi a hola-platzi)
```

```
$ docker rm <ID o nombre> (borro un contenedor)
```

```
$ docker container prune (borro todos los contenedores que estén parados)
```

```
$ docker ps -a (veo todos los contenedores)
```

```
$ docker --name <nombre> -d ubuntu -f <comando>
```

```
$ docker --name alwaysup -d ubuntu tail -f /dev/null (mantiene el contenedor activo)
```

```
$ docker exec -it alwaysup bash (entro al contenedor)
```

```
$ docker inspect --format '{{.State.Pid}}' alwaysup (veo el main process del ubuntu)
```

desde Linux si ejecuto `kill -9 <PID>` mata el proceso dentro del contenedor de ubuntu pero desde MAC no funciona

```
$ docker run -d --name proxy nginx (corro un nginx)
```

```
$ docker stop proxy (apaga el contenedor)
```

```
$ docker rm proxy (borro el contenedor)
```

```
$ docker rm -f <contenedor> (lo para y lo borra)
```

```
$ docker run -d --name proxy -p 8080:80 nginx (corro un nginx y expongo el
puerto 80 del contenedor en el puerto 8080 de mi máquina)
localhost:8080 (desde mi navegador compruebo que funcione)
$ docker logs proxy (veo los logs)
$ docker logs -f proxy (hago un follow del log)
$ docker logs --tail 10 -f proxy (veo y sigo solo las 10 últimas entradas del log)
```

```
$ mkdir dockerdata (creo un directorio en mi máquina)
$ docker run -d --name db mongo
$ docker ps (veo los contenedores activos)
$ docker exec -it db bash (entro al bash del contenedor)
$ mongo (me conecto a la BBDD)
$ docker run -d --name db --v mongodata:/data/db mongo
```

```
## The best way to store data alive
$ docker volume ls (listo los volumes)
$ docker volume create dbdata (creo un volume)
$ docker run -d --name db --mount src=dbdata,dst=/data/db mongo (corro la
BBDD y monto el volume)
$ docker inspect db (veo la información detallada del contenedor)
$ mongo (me conecto a la BBDD)
```

```
$ touch prueba.txt (creo un archivo en mi máquina)
$ docker run -d --name copytest ubuntu tail -f /dev/null (corro un ubuntu y le
agrego el tail para que quede activo)
$ docker exec -it copytest bash (entro al contenedor)
$ mkdir testing (creo un directorio en el contenedor)
$ docker cp prueba.txt copytest:/testing/test.txt (copio el archivo dentro del
contenedor)
$ docker cp copytest:/testing localtesting (copio el directorio de un contenedor a
mi máquina)
con "docker cp" no hace falta que el contenedor esté corriendo
```

```
$ docker image ls (veo las imágenes que tengo localmente)
$ docker pull ubuntu:20.04 (bajo la imagen de ubuntu con una versión específica)
$ docker image rm -f {id_image}
```

Comandos: Imágenes

```
$ mkdir imagenes (creo un directorio en mi máquina)
$ cd imagenes (entro al directorio)
```

```
$ touch Dockerfile (creo un Dockerfile)
$ code . (abro code en el directorio en el que estoy)
##Contenido del Dockerfile##
FROM ubuntu:latest
RUN touch /usr/src/hola-platzi.txt (comando a ejecutar en tiempo de build)
##fin##
$ docker build -t ubuntu:platzi . (creo una imagen con el contexto de build
<directorio>)
$ docker run -it ubuntu:platzi (corro el contenedor con la nueva imagen)
$ docker login (me logueo en docker hub)
$ docker tag ubuntu:platzi miusuario/ubuntu:platzy (cambio el tag para poder
subirla a mi docker hub)
$ docker push miusuario/ubuntu:platzi (publico la imagen a mi docker hub)
```

```
$ git clone https://github.com/platzi/docker
$ docker build -t platziapp . (creo la imagen local)
$ docker image ls (listo las imagenes locales)
$ docker run --rm -p 3000:3000 platziapp (creo el contenedor y cuando se
detenga se borra, lo publica el puerto 3000)
$ docker ps (veo los contenedores activos)
```

```
$ docker build -t platziapp . (creo la imagen local)
$ docker run --rm -p 3000:3000 -v pathlocal/index.js:pathcontenedor/index.js
platziapp (corro un contenedor y monto el archivo index.js para que se actualice
dinámicamente con nodemon que está declarado en mi Dockerfile)
```

```
$ docker network ls (listo las redes)
$ docker network create --attachable plazinet (creo la red)
$ docker inspect plazinet (veo toda la definición de la red creada)
$ docker run -d --name db mongo (creo el contenedor de la BBDD)
$ docker network connect plazinet db (conecto el contenedor "db" a la red
"plazinet")
$ docker run -d --name app -p 3000:3000 --env MONGO_URL=mondodb://
db:27017/test platzi (corro el contenedor "app" y le paso una variable)
$ docker network create --attachable plazinet
$ docker network connect plazinet app (conecto el contenedor "app" a la red
"plazinet")
```

DockerNetwork

Comandos:

- Listar las redes

`$docker network ls`

- Crear una red

`$docker network create --attachable <network_name>`

`$docker network create --attachable plazinet`

- Inspeccionar una red

`$docker network inspect <network_name>`

- Conectar un contenedor a la red

`$docker network connect <network_name> <container_name>`

`$docker network connect plazinet db`

- Correr un contenedor usando variables de entorno

`$docker run --env MY_VAR="" <image_name>`

`$docker run -d -name app -p 3000:3000 --env MONGO_URL=mondodb://db:27017/test_platzi`

## imagenes

### **Profundizando en el concepto de imagen**

Es bueno profundizar un poco más en el concepto de una imagen en Docker para entender su función, para posteriormente poder realizar una por nuestra cuenta desde 0, cuando no haya una imagen que cumpla con nuestras necesidades.

.

#### **Imagen**

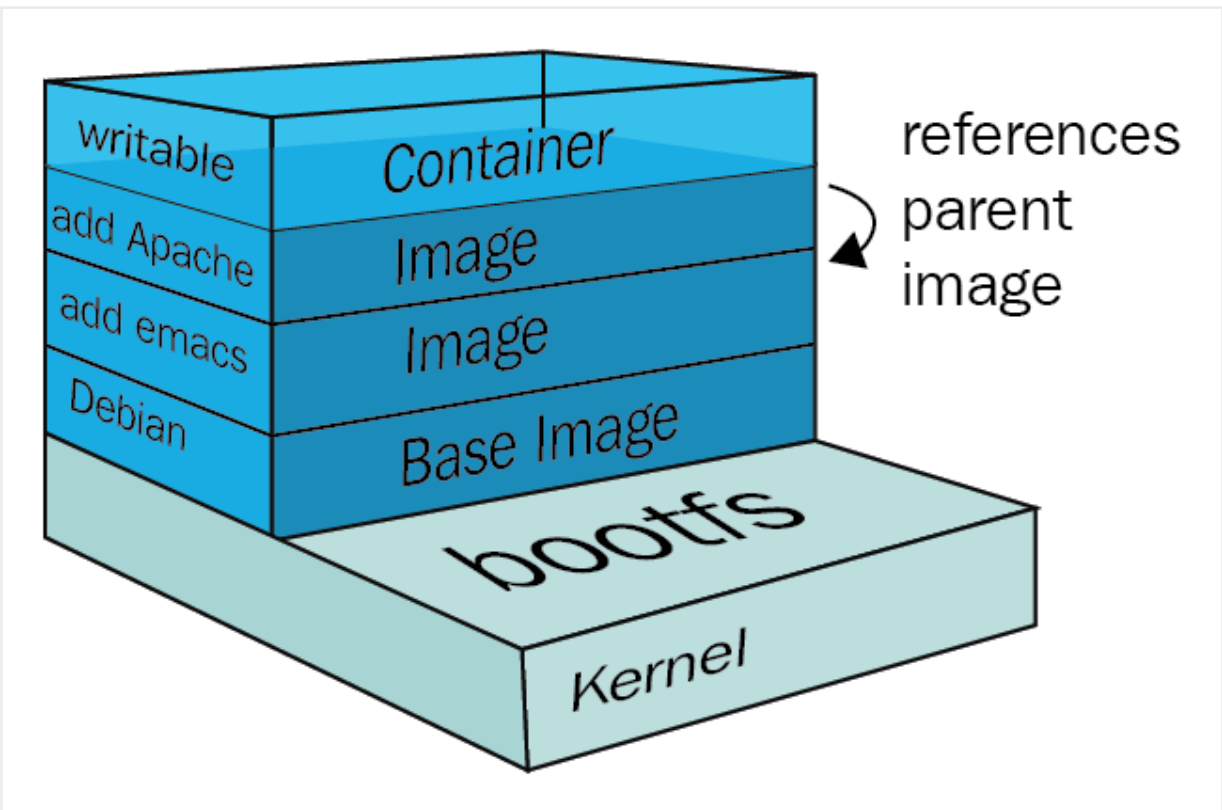
Una imagen contiene distintas capas de datos (distribución, diferente software, librerías y personalización).

.

Podemos llegar a la conclusión, que una imagen se conforma de distintas capas de personalización, en base a una capa inicial (base image), la dicha capa, es el más puro estado del SO.

.

La siguiente ilustración nos mostraría la representación gráfica, del concepto de una imagen en Docker.



Si observamos, partimos desde la base del SO, y vamos agregando capas de personalización hasta obtener la imagen que necesitamos:

1. distribución debian
2. se agrega el editor emacs
3. se agrega el servidor Apache
4. se agregan los permisos de escritura para la carpeta /var/www de Apache

*Hay que tener en cuenta, que todo parte del Kernel de Linux, en caso de utilizar alguna distribución de Linux*

.

### **Historico de una imagen**

Podemos observar la historia de nuestra imagen, con el siguiente comando  
\$ docker history [imagen]

De esta manera podemos ver las capas de personalización que fueron agregadas, para la construcción de la imagen que conocemos.

Docker compose nos ayuda utilizar docker con una estructura declarativa y facilita la gestión. Toda la configuración la podemos encontrar en el fichero .YAML.

```
# Versión del compose file
version: "3.8"
```

```
# Servicios que componen nuestra aplicación.
```

```
## Un servicio puede estar compuesto por uno o más contenedores.
```

```
services:
```

```
# nombre del servicio.
```

```
  app:
```

```
    # Imagen a utilizar.
```

```
    image: platziapp
```

```
      # Declaración de variables de entorno.
```

```
  environment:
```

```
    MONGO_URL: "mongodb://db:27017/test"
```

```
      # Indica que este servicio depende de otro, en este caso DB.
```

```
      # El servicio app solo iniciara si el servicio debe inicia correctamente.
```

```
  depends_on:
```

```
    - db
```

```
      # Puerto del contenedor expuesto.
```

```
  ports:
```

```
    - "3000:3000"
```

```
  db:
```

```
    image: mongo
```

Comandos:

```
$ docker network ls (listo las redes)
```

```
$ docker network inspect docker_default (veo la definición de la red)
```

```
$ docker-compose logs (veo todos los logs)
```

```
$ docker-compose logs app (solo veo el log de "app")
```

```
$ docker-compose logs -f app (hago un follow del log de app)
```

```
$ docker-compose exec app bash (entro al shell del contenedor app)
```

```
$ docker-compose ps (veo los contenedores generados por docker compose)
```

```
$ docker-compose down (borro todo lo generado por docker compose)
```

**Para ejecutar comandos de docker compose desde afuera es necesario pasarle la ruta en del archivo con la opción -f**

```
docker-compose -f path/to/docker-compose.yml up -d
```

**Todo lo demás se puede hacer igual**

```
docker-compose -f path/to/docker-compose.yml logs app
```

**Esta opción brinda versatilidad, pues también se puede utilizar para**

## **especificar un archivo de compose diferente, por ejemplo:**

```
docker-compose -f docker-compose.production.yml up -d
```

Comandos:

```
$ docker-compose build (crea las imágenes)
$ docker-compose up -d (crea los servicios/contenedores)
$ docker-compose logs app (veo los logs de "app")
$ docker-compose logs -f app (hago un follow de los logs de "app")
$ docker-compose down
```

Comandos:

```
$ touch docker-compose.override.yml (creo el archivo override)
$ docker-compose up -d (crea los servicios/contenedores)
$ docker-compose exec app bash (entro al bash del contenedor app)
$ docker-compose ps (veo los contenedores del compose)
$ docker-compose up -d --scale app=2 (escalo dos instancias de app,
previamente tengo que definir un rango de puertos en el archivo compose)
$ docker-compose down (borro todo lo creado con compose)
```

Comandos:

```
$ docker ps -a (veo todos los contenedores de mi máquina)
$ docker container prune (borra todos los contenedores inactivos)
$ docker rm -f $(docker ps -aq) (borra todos los contenedores que estén
corriendo o apagados)
$ docker network ls (lista todas las redes)
$ docker volume ls (lista todos los volumes)
$ docker image ls (lista todas las imágenes)
$ docker system prune (borra todo lo que no se esté usando)
$ docker run -d --name app --memory 1g platziapp (limito el uso de memoria)
$ docker stats (veo cuantos recursos consume docker en mi sistema)
$ docker inspect app (puedo ver si el proceso muere por falta de recursos)
```

Comandos:

```
$ docker build -t loop . (construyo la imagen)
$ docker run -d --name looper loop (corro el contenedor)
$ docker stop looper (le envía la señal SIGTERM al contenedor)
$ docker ps -l (muestra el ps del último proceso)
$ docker kill looper (le envía la señal SIGKILL al contenedor)
$ docker exec looper ps -ef (veo los procesos del contenedor)
```

Comandos:

```
$ docker build -t ping . (construyo la imagen)
```



\$ docker run --name pinger ping <hostname> (ahora le puedo pasar un parámetro, previamente tengo que agregar el ENTRYPOINT en el Dockerfile)

Comandos:

\$ docker build -t prueba .(creo la imagen)

\$ docker run -d --rm --name app prueba (corro el contenedor)

en el archivo .dockerignore puedo poner todo lo que no quiero que copie del contexto de build

\$ docker exec -it app bash (entro al contenedor y verifico que no se haya copiado lo que está en el .dockerignore)

Comandos:

\$ docker build -t prodapp -f Dockerfile . (ahora le especifico el Dockerfile)

\$ docker run -d --name prod prodapp

## testing al hacer build

<https://platzi.com/clases/2066-docker/32870-multi-stage-build/?time=686>

Comandos:


\$ docker run -it --rm -v /var/run/docker.sock:/var/run/docker.sock docker:19.03.12

\$ docker run --rm -it -v /var/run/docker.sock:/var/run/docker.sock -v \$(which docker):/bin/docker wagooodman/dive:latest prodapp

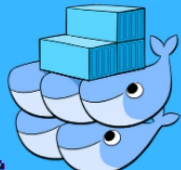
## DOCKER SWARM

- Ejecutar aplicaciones productivas: la aplicación debe estar lista para servir a las usuarios a pesar de situaciones catastróficas, o de alta demanda (carga).
- Escalabilidad: Poder aumentar la potencia de cómputo para poder servir a más usuarios, o a peticiones pesadas.
- Escalabilidad vertical: Más hardware, hay límite físico.

- Escalabilidad horizontal: Distribuir carga entre muchas computadoras. es el más usado.
- Disponibilidad: Es la capacidad de una aplicación o servicio de estar siempre disponible para los usuarios. prevé problemas con servidores, etc.
- La escalabilidad horizontal y la disponibilidad van de la mano.





DOCKER SWARM

## PROBLEMA DE ESCALA



- Escalabilidad: poder aumentar potencia de computo.
  - Escalar vertical: Mejorar el hardware, suele ser más costoso, en caso de falla tenemos 0% disponibilidad.
  - Escalar Horizontal: Equipos paralelos, disponibilidad es garantizada.

Docker Swarm permite escalabilidad horizontal y disponibilidad

 @DiegoDevelops
  @conedie
  @DiegoDevelops

- La arquitectura de docker swarm tiene el esquema de servidores manager y workers.
- Los manager son los servidores que administran la comunicación y recursos entre los contenedores.
- Los workers son los nodos donde se ejecutan los contenedores.

- Todos estos nodos deben tener instalado docker daemon, de ser posible la misma versión y visibles entre sí.

SWARM orquesta el flujo entre todos los workers

Doce factores de la aplicación.

- Codebase, tu código debe estar en un repositorio y este debería estar en relación de 1 a 1 entre código y repositorio.
- Dependencias, deberían venir empaquetadas con la aplicación.
- La configuración, debe ser parte de tu aplicación.
- Backing Service, como bases de datos, deben ser tratados como servicios externos a la aplicación.
- Build, Release, Run. estas tres fases deben estar separadas en tu aplicación.
- Process, la ejecución de tu aplicación no puede depender de que exista cierto estado, todo proceso lo debe realizar de forma atómica, stayless.
- Port binding, la aplicación debe poder exponerse a si misma, sin intermediarios.
- Concurrencia, que la aplicación pueda correr con múltiples instancia en paralelo.
- Disposability, la aplicación debe estar diseñada para ser fácilmente destruible e iniciar rápidamente.
- Dev/prod parity, lograr que entorno de desarrollo, sea lo más parecido a producción.
- Logs, Todos los logs de la aplicación deben tratarse como un flujo de device.
- Admin Process, la aplicación debe poder ejecutar como procesos independientes.

## **The Twelve-Factor App**

<https://12factor.net/>

```
// Iniciar docker swarm
```

```
docker swarm init
```

```
// Obtener token para unir manager
```

```
docker swarm join-token manager
```

```
// Ver los nodos que tenemos
```

docker **node ls**

// Ver **información** del nodo

docker **node inspect** self

// Salir del modo swarm

//salir del modo swarm

docker swarm leave

//si un worker **node se** va, dará error, podemos forzarlo.

docker swarm leave --force

// ver estado de docker Swarm

docker info | **grep** Swarm

// Forzar salida de modo swarm abreviado

docker swarm leave -f

// Imprimir en forma legible la información del nodo

docker node **inspect** --pretty self

- Swarm no es un paquete aparte de docker, swarm es nativo.
- Docker swarm init : inicia un nodo manager.
- Debe existir como mínimo un manager para que exista swarm.
- docker swarm join-token manager
- docker node ls : muestra los nodos disponibles
- Toda la comunicación entre nodos está encriptada usando certificados TLS.
- docker node inspect --pretty self: inspecciona el nodo
- docker swarm leave --force: cierra el modo swarm, regresa a docker normal.

**En swarm siempre tienes que tener algún mecanismo de mantenimiento y limpieza. Yo uso [meltwater/docker-cleanup](#) corriendo como un servicio global de Swarm, lo que me garantiza que corre en todos los nodos, y delego en él la tarea de limpiar todo. Lo hago así:**

```
docker service create \  
  --detach \  
  -e CLEAN_PERIOD=900 \  
  -e DELAY_TIME=600 \  
  --log-driver json-file \  
  --log-opt max-size=1m \  

```

```
--log-opt max-file=2 \  
--name cleanup \  
--mode global \  
--mount type=bind,source=/var/run/docker.sock,target=/var/run/docker.sock \  
meltwater/docker-cleanup
```

```
docker service create --name pinger alpine ping www.google.com  
docker service ls
```

```
/ Ver estado de un servicio  
docker service ps nombreServicio
```

```
// Ver información de un servicio  
docker service inspect nombreServicio
```

```
// Ver logs  
docker service logs -f nombreServicio
```

```
// Eliminar servicio  
docker service rm nombreServicio
```

Un servicio es como una descripción de algo que se tiene que llevar a cabo, ahora esta descripción nos dice qué se quiere hacer, pero recordemos que se puede escalar los servicios, entonces podemos llevar el servicio en marcha según la escalabilidad que le demos, entonces le estaríamos diciendo al nodo manager “che quiero 5 nodos haciendo esto” entonces el nodo manager realiza su planificación, y en su planificación están 5 tareas de determinado servicio, las cuales asigna a los nodos workers.

Entonces tenes identificadores (IDs) para:

- Identificar servicios, es decir la descripción de lo que se tiene que llevar a cabo.
- Identificar los servicios en ejecución (tareas).
- Identificar los contenedores en los cuales las tareas se están llevando a cabo.

Según entiendo sobre el tema los IDs en Swarm.

[How services work | Docker Documentation](https://docs.docker.com/engine/swarm/how-swarm-mode-works/services/)

<https://docs.docker.com/engine/swarm/how-swarm-mode-works/services/>

– Creamos el nodo manager

```
docker swarm init --advertise-addr <MANAGER-IP>
docker swarm init --advertise-addr 192.168.0.18
– Creamos una nueva instancia en play-with-docker (+ ADD NEW INSTANCE)
docker swarm join --token <TOKEN> <MANAGER-IP>:<PORT>
docker swarm join --token
SWMTKN-1-32cege8duoof9cr405bi1fsmcga831l6fcec mzn p5cxcfdc3vg-
ci6f98tjfy9fzhr2swmmo3ter 192.168.0.18:2377
– Creamos otra nueva instancia en play-with-docker (+ ADD NEW INSTANCE)
docker swarm join --token <TOKEN> <MANAGER-IP>:<PORT>
docker swarm join --token
SWMTKN-1-32cege8duoof9cr405bi1fsmcga831l6fcec mzn p5cxcfdc3vg-
ci6f98tjfy9fzhr2swmmo3ter 192.168.0.18:2377
– Nos dirigimos a la terminal del nodo MANAGER, observamos los 3 nodos
docker node ls
– Crear un servicio en este caso multinodo
docker service create --name pinger alpine ping www.google.com
– Ver listado de servicios
docker service ls
– Donde estan asignado las tareas de este servicio, nos indica que esta en el nodo
1
docker service ps pinger
– Podemos ver el container
docker ps
```

Comandos de la clase, con el ejemplo del servicio pinger

```
# Cambia el numero de tareas
docker service scale pinger=5
```

```
# Ver las tareas del servicio
docker service ps pinger
```

```
# Ver los logs del servicio
docker service logs -f pinger
```

```
# Ver la configuración del servicio
docker service inspect pinger
```

```
# actualizar alguna configuración del servicio
docker service update --args "ping www.amazon.com" pinger
```

```
# realiza rollback o cambia al spec anterior
docker service rollback pinger
```

# Actualizar las replicas **de** un servicio  
docker service **update** --replicas=20 pinger

# Actualizar paralelismo y orden **de la** configuración **de update en** el servicio  
pinger  
docker service **update** --**update**-parallelism 4 --**update-order** start-first pinger

# Actualizar accion **en** fallo y radio maximo **de** falla **de la** configuración **de update en** el servicio pinger  
docker service **update** --**update**-failure-action rollback --**update**-max-failure-**ratio** 0.5 pinger

# Actualizar paralelismo **de la** configuracion **de** rollback **en** el servicio **de** pinger  
docker service **update** --rollback-parallelism 0 pinger

docker service scale pinger=10  
docker service **update** --replicas=20 pinger  
docker service **update** -d --replicas=20 pinger  
docker service **inspect** pinger  
docker service **update** --**update**-parallelism 4 --**update-order** start-first pinger  
docker service **inspect** pinger  
docker service **update** --**args** "ping www.facebook.com" pinger  
docker service ps pinger  
docker service **update** --**update**-failure-action rollback --**update**-max-failure-**ratio** 0.5 pinger  
docker service **update** --rollback-parallelism 0 pinger  
docker service **update** --**args** "ping www." pinger

Lanzar mundo exterior

Entendido. Yo cree mi propio entrono en máquinas virtuales con vmware y Ubuntu server 18.04 y todo funciona correctamente.

docker build -t baezdavidsan/swarm-hostname  
docker login  
docker push baezdavidsan/swarm-hostname:latest  
docker node ls  
docker service create --name app --publish 3000:3000 --replicas=3  
baezdavidsan/swarm-hostname:latest  
docker service create --name app -d --publish 3000:3000 --replicas=3

```
baezdavidsan/swarm-hostname:latest
curl http://192.168.5.153:3000/
```

Les dejo el comando para lanzar la aplicación

```
docker service create -d --name app --publish 3000:3000 --replicas=3 gvilarino/
swarm-hostname
```

-Routing mesh habilita cada nodo de swarm para aceptar conexiones en puertos publicados para cualquier servicio corriendo en swarm, incluso si no hay tareas corriendo en el nodo.

- Routing mesh rutea todos los requests en lo puertos publicados en los nodos disponibles en un contenedor activo.

```
docker service scale app=6
docker service ps app
docker ps
docker network ls
docker service create --name app --publish 3000:3000 --replicas=3
baezdavidsan/swarm-hostname:latest
```

Cuando se tenía un contenedor el en puerto 3000 y se volvía a crear otro contenedor con el mismo puerto 3000, **nos mostraba un error**, ya que **el puerto ya estaba ocupado**.

Que pasa cuando creamos o escalamos los servicios con el mismo puerto: **Swarm crea una Red llamada "ingress"** que implementa el **Routing Mesh** para **el uso de un puerto** como **balanceador de carga**.

## ## restricciones de despliegue

```
docker service create -d --name viz -p 8080:8080 --
constraint=node.role==manager --mount=type=bind,src=/var/run/
docker.sock,dst=/var/run/docker.sock dockersamples/visualizer
```

Comando para reubicar los servicios para que sean ejecutados solo en los workers

```
docker service update --constraint-add node.role==worker --update-
parallelism=0 app
```

donde:

-constraint-add node.role => indica el tipo de nodo donde quiero reubicar

-update-parallelism=0 => indica que se harán todos en simultaneo



## Disponibilidad de nodos

En el visualizer podemos ver que todas las tareas siguen corriendo en el "worker 1", esto sucede porque el planificador de **Docker Swarm** no va a replanificar o redistribuir la carga de un servicio o de contenedores a menos que tenga que hacerlo; para solucionar esto, debemos forzar un redeployment o una actualización que se logra cambiando el valor de una variable que no sirva para nada.

```
docker service update -d --env-add UNA_VARIABLE=de-entorno --update-parallelism=0 app
```

```
docker service ps app
```

```
docker node ls
```

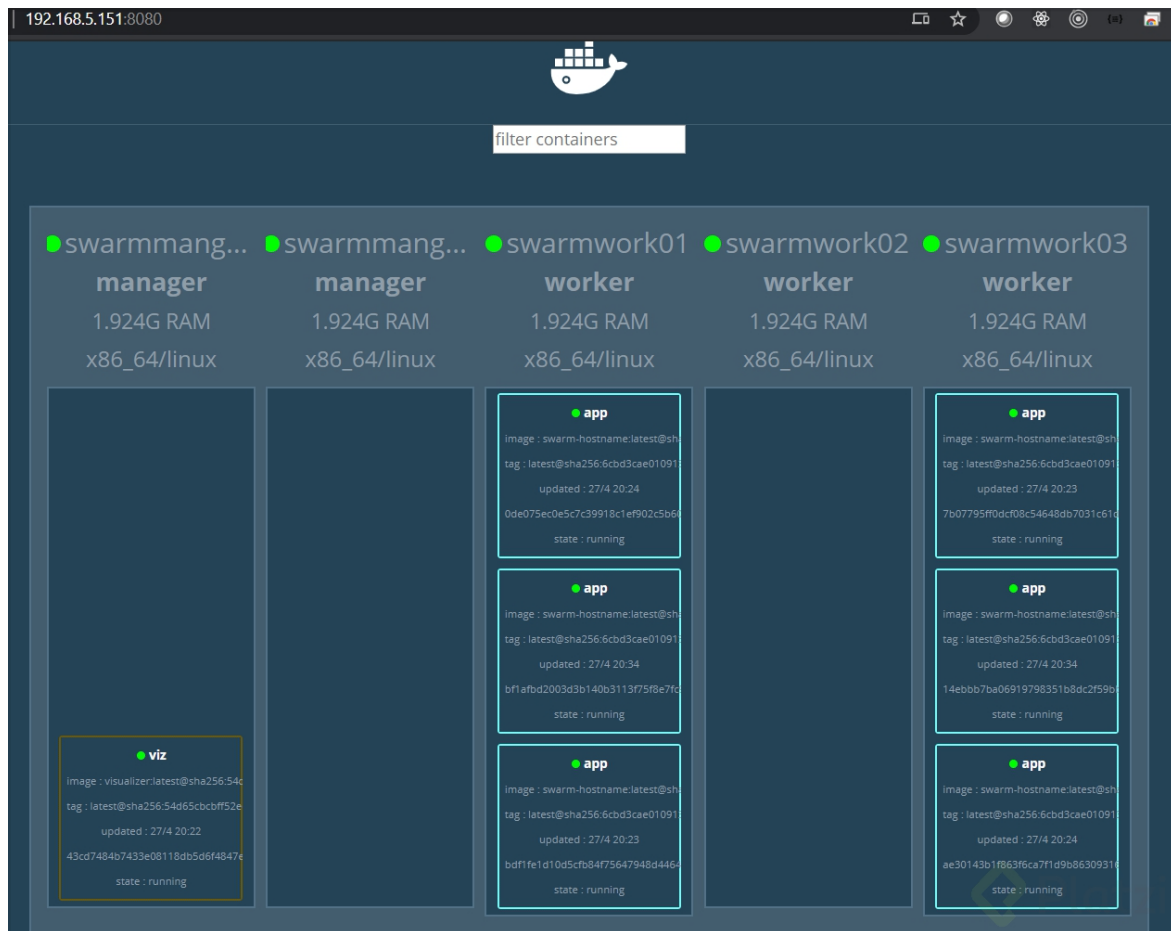
```
docker node inspect --pretty swarmwork02
```

```
docker node update --availability drain swarmwork02
```

```
docker node update --availability active swarmwork02
```

```
docker service update -d --env-add UNA_VARIABLE=de-entorno --update-parallelism=0 app
```

```
docker service rm app
```



– Nos dirigimos a nuestro repositorio <https://github.com/platzi/swarm>, a la carpeta stack.

**cd** swarm/stacks

**cat** stackfile.yml

– Nos dirigimos a nuestro manager1

**docker service rm app db**

**docker network rm app-net**

**vim** stackfile.yml

#####stackfile.yml#####

version: "3"

```
services:
  app:
    image: borisvargas/swarm-networking
    environment:
      MONGO_URL: "mongodb://db:27017/test"
    depends_on:
      - db
    ports:
      - "3000:3000"
```

```
db:
  image: mongo
#####
docker stack deploy --compose-file stackfile.yml app
```

docker **stack ls**

docker **stack ps app**

```
docker stack services app
– Quiero que los servicios esten en los workers`
vim stackfile.yml
#####stackfile.yml#####
version: "3"
```

```
services:
  app:
    image: borisvargas/swarm-networking
    environment:
      MONGO_URL: "mongodb://db:27017/test"
    depends_on:
      - db
    ports:
      - "3000:3000"
    deploy:
      placement:
        constraints: [node.role==worker]
```

```
db:
  image: mongo
#####
docker stack deploy --compose-file stackfile.yml app
```

```
docker stack rm app
```

Nos ayuda a simplificar la administración.

```
docker service rm app
```

```
docker service rm db
```

```
docker network rm app-net
```

```
docker stack deploy --compose-file stackfile.yml app
```

```
docker service ls
```

```
docker stack ls
```

```
docker stack ps app
```

```
docker stack services app
```

```
docker service scale app_app=3
```

```
docker stack rm app
```

### ## Reverse proxy: muchas aplicaciones, un sólo dominio

Me funciona en mi entorno de pruebas

```
docker network create --driver overlay proxy-net
```

```
docker service create --name proxy --constraint=node.role==manager -p 80:80
```

```
-p 9090:8080 --mount type=bind,src=/var/run/docker.sock,dst=/var/run/
```

```
docker.sock --network proxy-net traefik:1.7 --docker --docker.swarmmode --
```

```
docker.domain=dbz.com --docker.watch --api
```

```
docker service create --name app1 --network proxy-net --label traefik.port=3000
```

```
baezdavidsan/swarm-hostname
```

```
curl -H "Host: app1.dbz.com" http://localhost
```

```
docker service create --name app2 --network proxy-net --label
```

```
traefik.port=3000 baezdavidsan/swarm-hostname
```

```
curl -H "Host: app2.dbz.com" http://localhost
```

```
docker service update --image baezdavidsan/networking app2
```

```
curl -H "Host: app1.dbz.com" http://localhost
```

```
curl -H "Host: app2.dbz.com" http://localhost
```

```
http://192.168.5.152:9090/dashboard/
```

- Como mínimo necesitamos tres manager.
- Podemos configurar grupos de workers según la necesidad de cómputo.
- El número de manager debe ser impar. Hay un único líder, y se rotan el liderazgo en un intervalo de tiempo

## Lecturas recomendadas



**GitHub - meltwater/docker-cleanup: Automatic Docker image, container and volume cleanup**

<https://github.com/meltwater/docker-cleanup>



**Configure logging drivers | Docker Documentation**

<https://docs.docker.com/config/containers/logging/configure/>



**GitHub - stefanprodan/swarmprom: Docker Swarm instrumentation with Prometheus, Grafana, cAdvisor, Node Exporter and Alert Manager**

<https://github.com/stefanprodan/swarmprom>