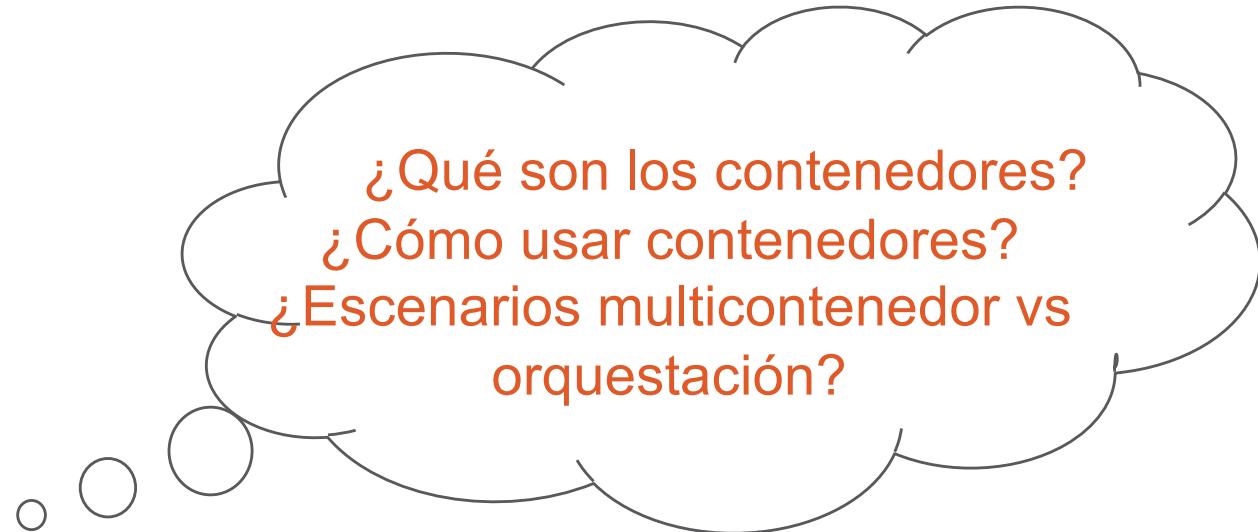


# Servidores Web de Altas Prestaciones



¿Qué son los contenedores?  
¿Cómo usar contenedores?  
¿Escenarios multicontenedor vs  
orquestación?

## Práctica 0: Contenerización



# Índice

01

## Introducción

- Desde inicio hasta actualidad
- Qué es Docker
- Conceptos básicos
- Instalación

02

## Cliente en Docker

- Hola mundo
- Operaciones
- Comandos básicos

03

## Datos en Docker

- Docker volume
- Docker bind mount

## Redes en Docker

- Tipos de redes
- Gestionando redes
- Uso de redes definidas por usuario

04

## Imágenes en Docker

- Gestionando imágenes
- Creación de imágenes propias. Dockerfile
- Distribución de imágenes

05

## Escenarios multi-contenedor

- Escenarios vs orquestación
- Docker compose. Comandos
- Fichero docker-compose.yml
- Almacenamiento en Docker compose
- Redes en Docker compose

06



# Introducción

Debes conocer el pasado para comprender el presente.

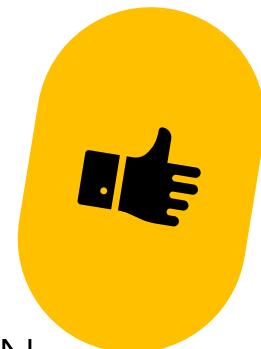
Carl Sagan (1934 - 1996)

Era de escritorio		Era de Máquinas Virtuales													Era de Contenedores						
2003	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013	2014	2015	2016	2017	2018	2019	2020	2021	2022	2023	

¿Que necesitamos generalmente para desarrollar una aplicación?  
¿Qué queremos?

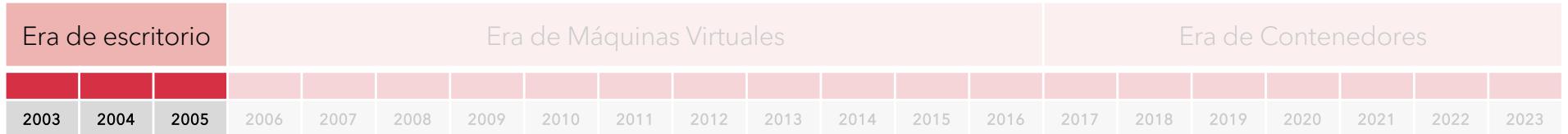


AUTOMATIZACIÓN





# Introducción



Desarrollos  
manuales



- Compilaciones manuales
- Primeros manejadores de paquetes
  - rpm, apt-get
- Primeras aplicaciones web
  - cgi-bin, servlets
- Dependencias mínimas
  - JRE + Tomcat, php
- Desarrollos manuales
  - scp o FTP
- Grandes diferencias entre servidores
- Etc.

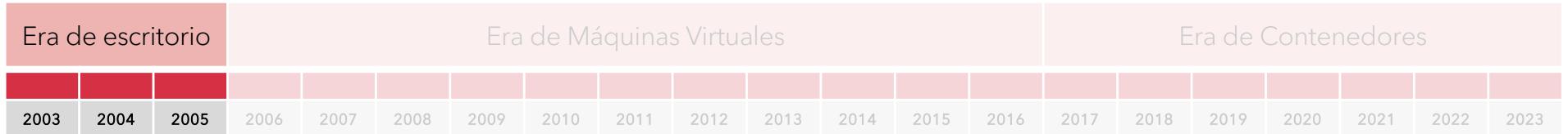


¿Qué podemos  
automatizar?





# Introducción



Desarrollos  
manuales



Código de Aplicación

Base de  
Datos

Runtime

Servidor  
web

Sistema Operativo

Hardware

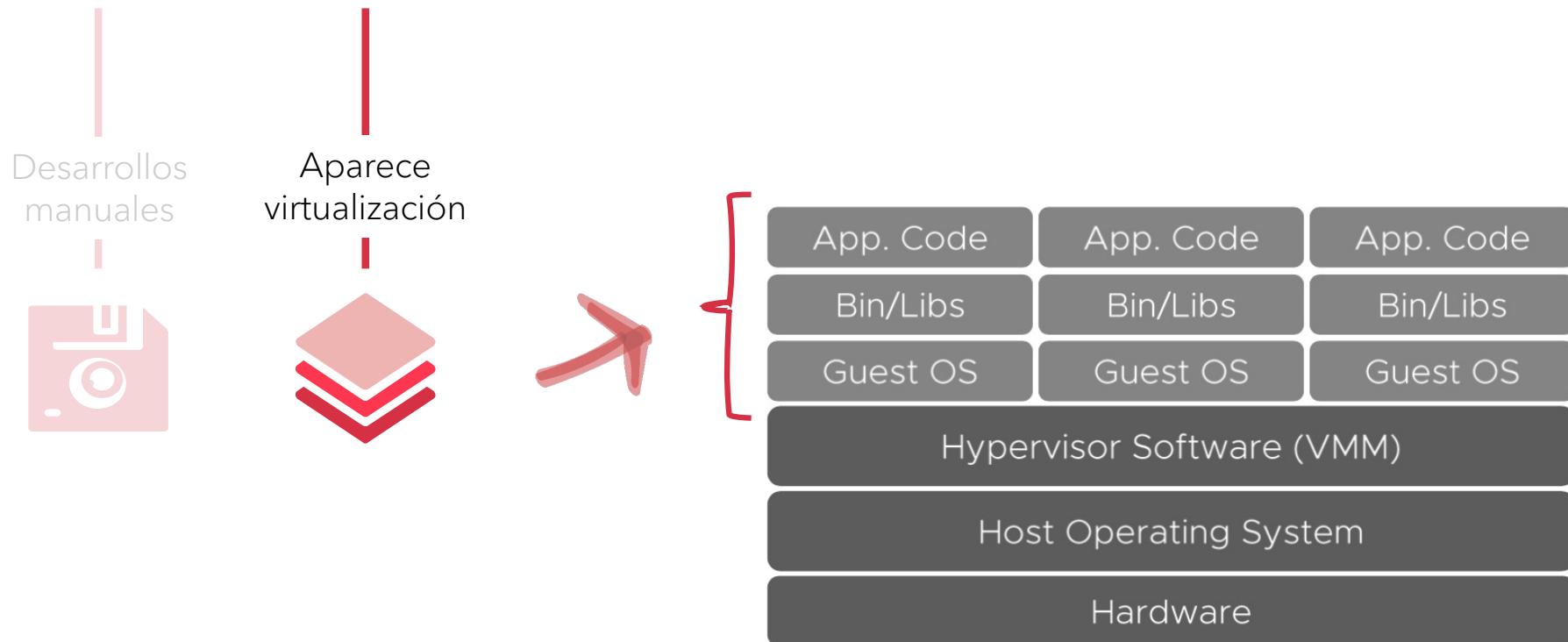


Automatización  
mínima ..





# Introducción



¿Qué podemos automatizar?





# Introducción



Desarrollos manuales



Aparece virtualización



Código de Aplicación

Base de Datos

Runtime

Servidor web

Sistema Operativo

Hardware



Se gana en reproducibilidad, pero existen muchas diferencias entre software de virtualización



Queda aún bastante para virtualización





# Introducción



Desarrollos  
manuales



Aparece  
virtualización



Gestión de  
configuración

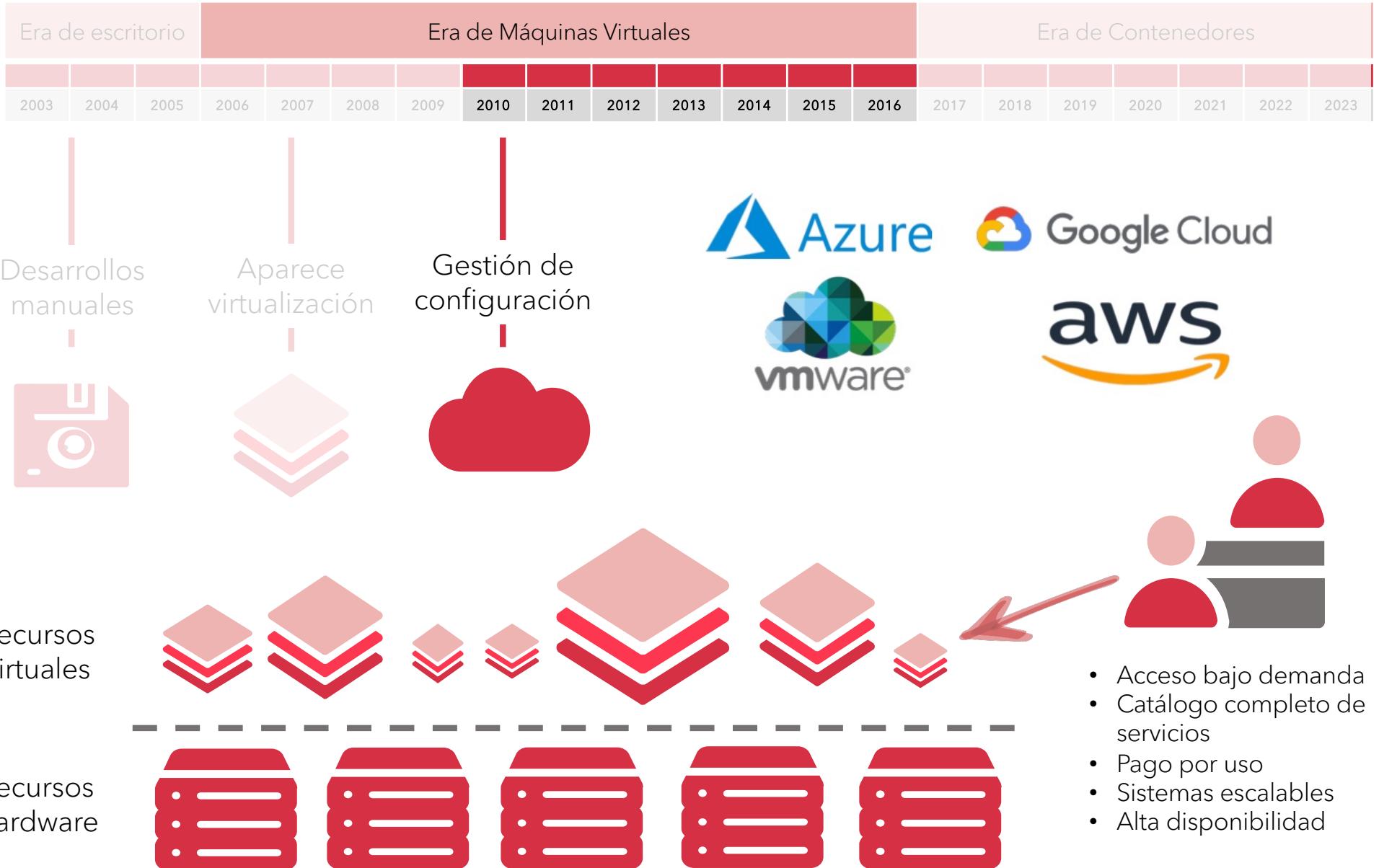


- 
- Web 2.0
  - Virtualización empieza a expandirse
  - Llega el paradigma Cloud Computing!!





# Introducción





# Introducción



Usuario administrador



Proveedor de servicios Cloud



On premises  
(servidores propios)



Aplicaciones
Datos
Rutinas
Middleware
Sistema Operativo
Virtualización
Servidores
Almacenamiento
Redes

IaaS  
(máquinas virtuales)



Aplicaciones
Datos
Rutinas
Middleware
Sistema Operativo
Virtualización
Servidores
Almacenamiento
Redes

PaaS  
(servicio aplicaciones)



Aplicaciones
Datos
Rutinas
Middleware
Sistema Operativo
Virtualización
Servidores
Almacenamiento
Redes

SaaS  
(Aplicaciones)



Aplicaciones
Datos
Rutinas
Middleware
Sistema Operativo
Virtualización
Servidores
Almacenamiento
Redes

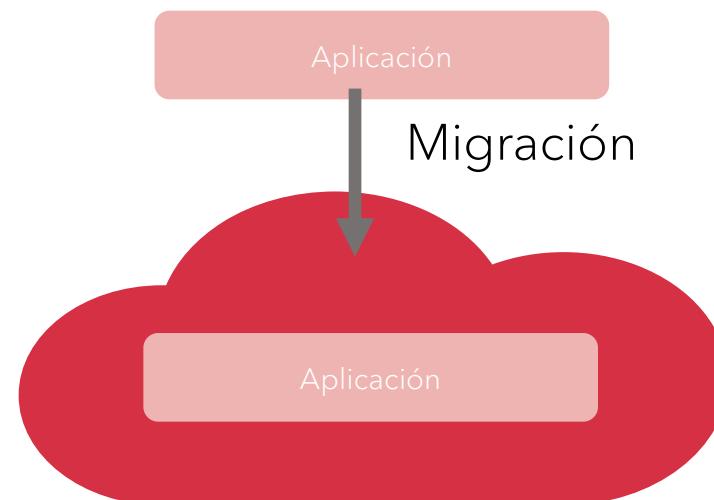




# Introducción



Era de la computación  
en la nube

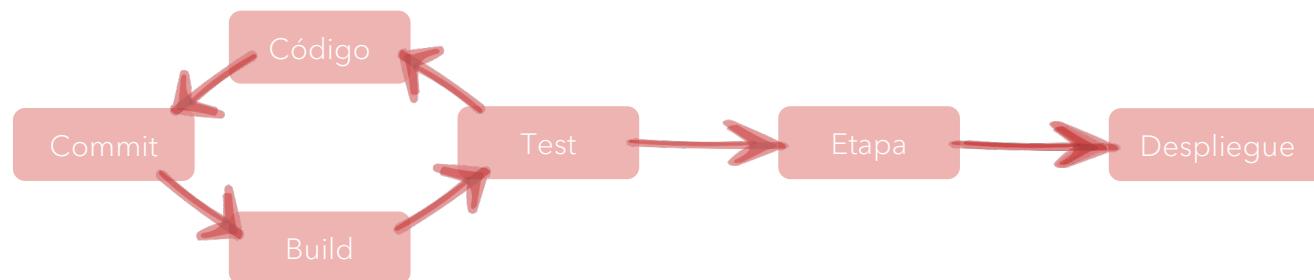
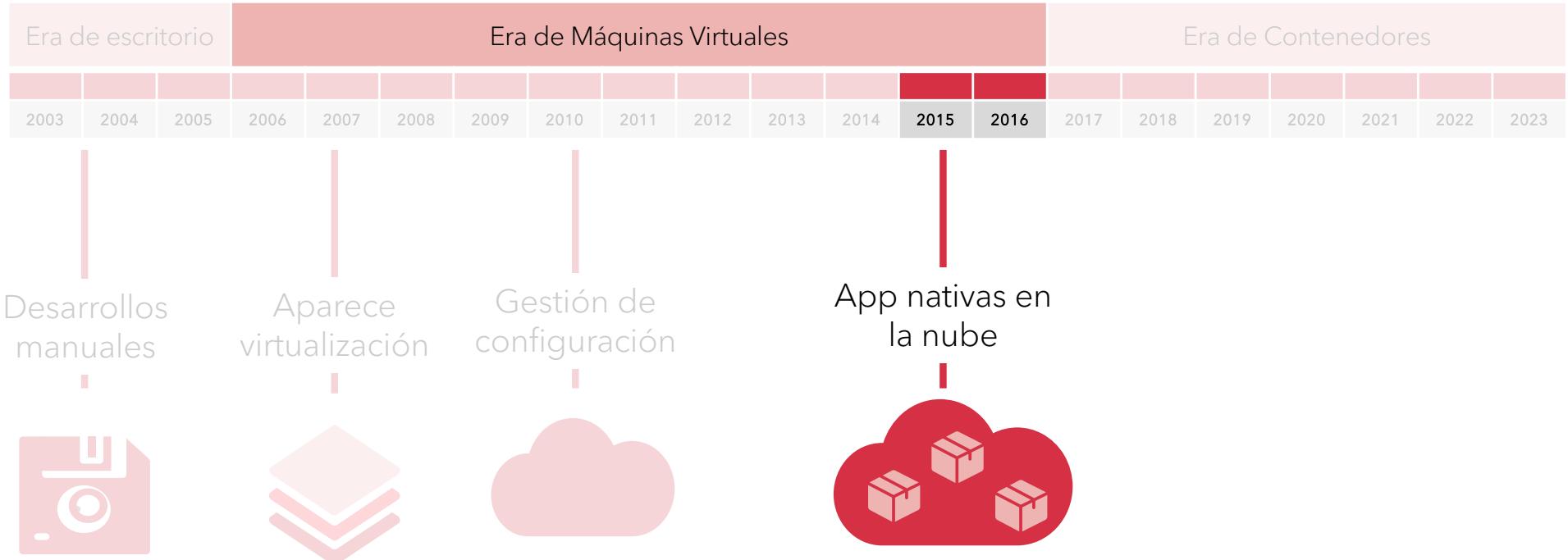


NO aplicaciones nativas en la nube



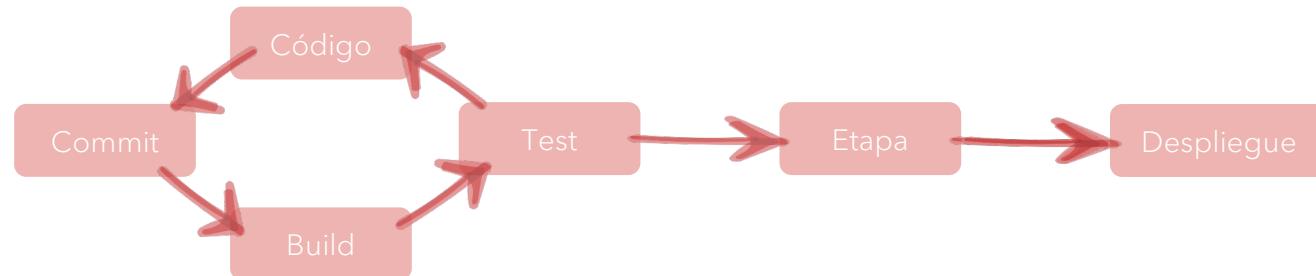
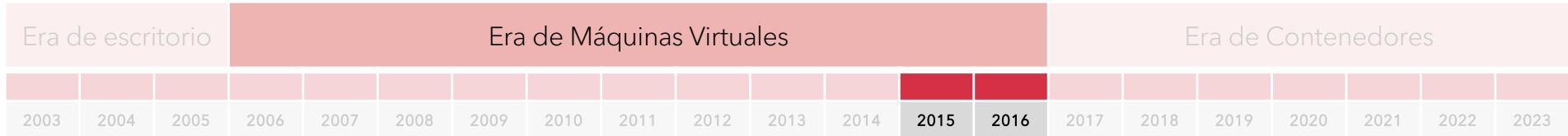


# Introducción

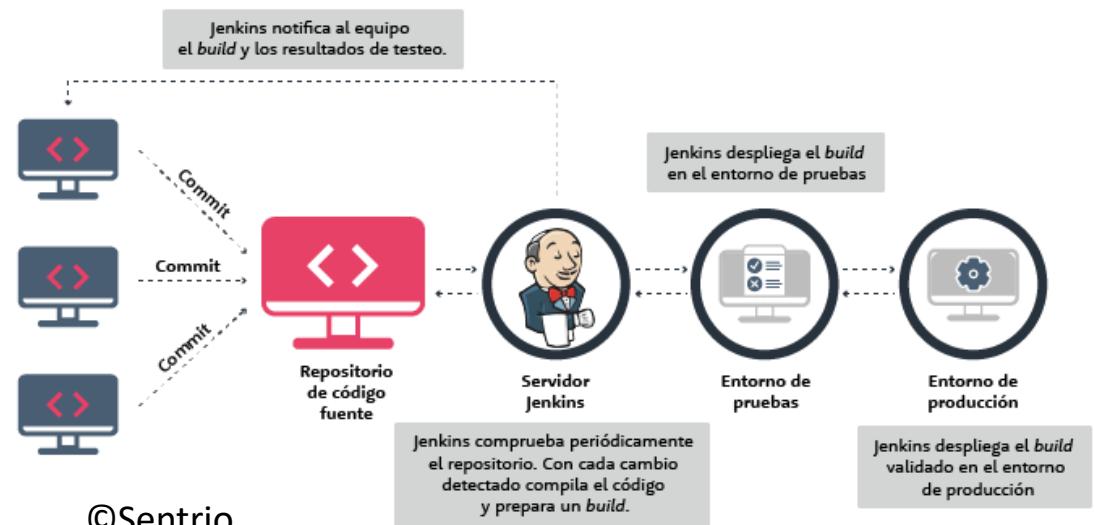




# Introducción



Nuevas herramientas de automatización de desarrollo de aplicaciones

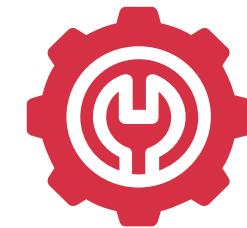
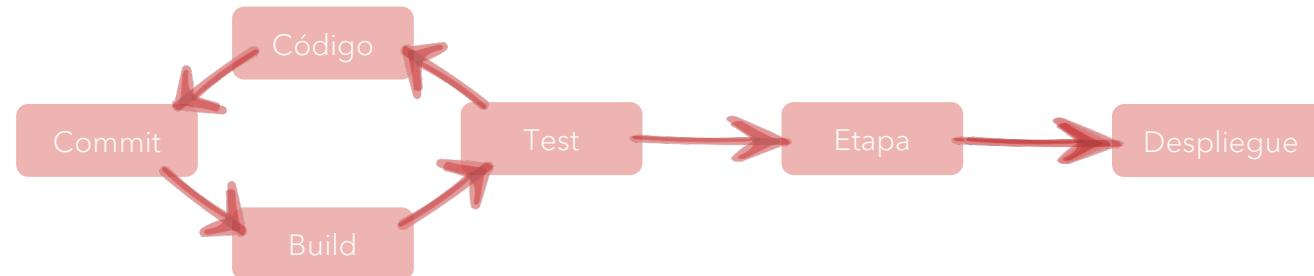
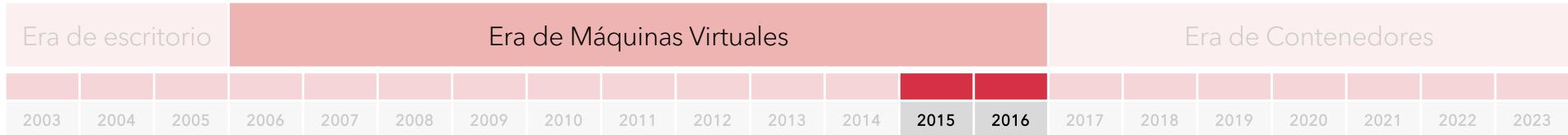


©Sentrio





# Introducción



## Nuevos paradigmas para gestionar recursos

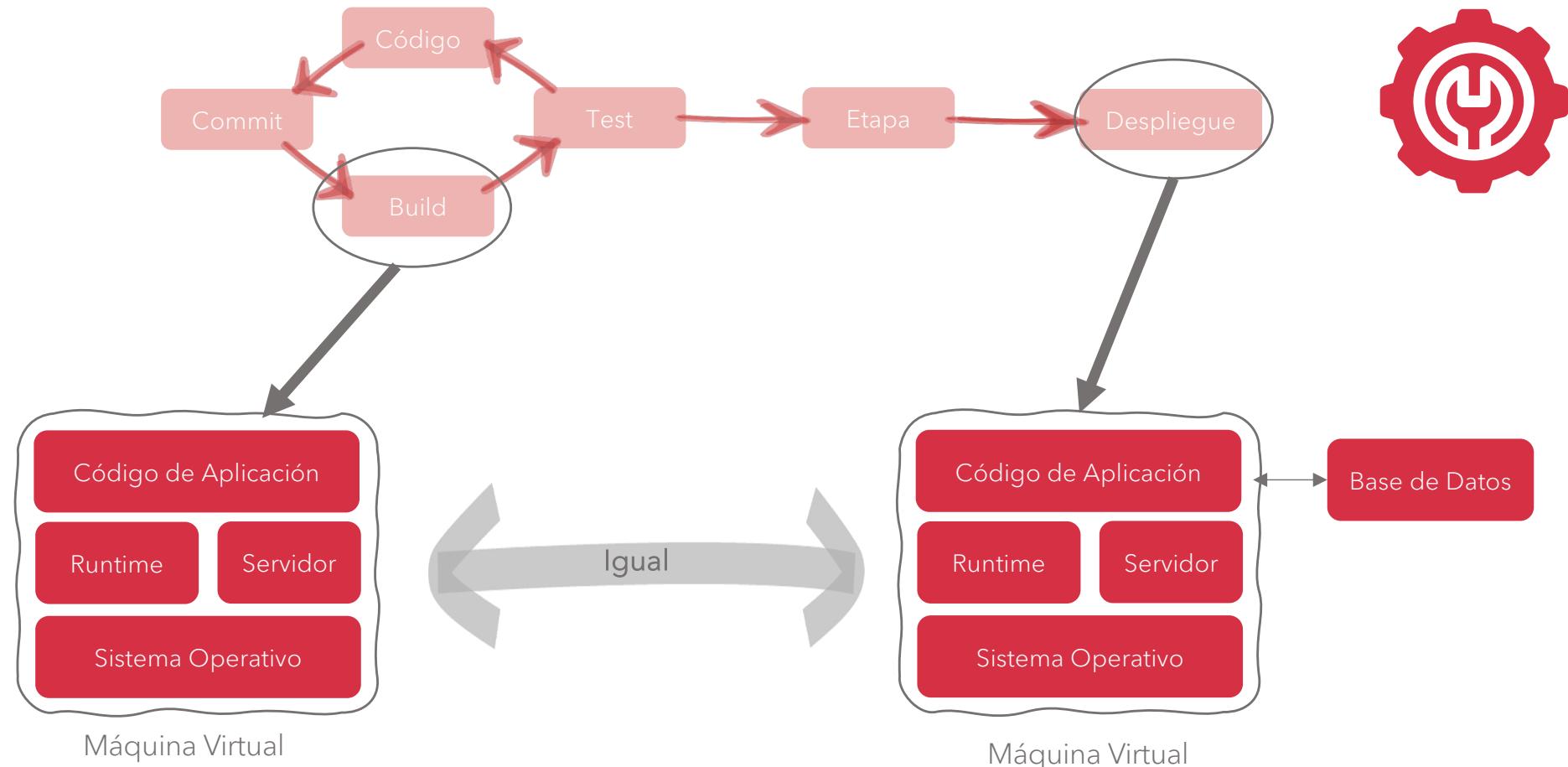
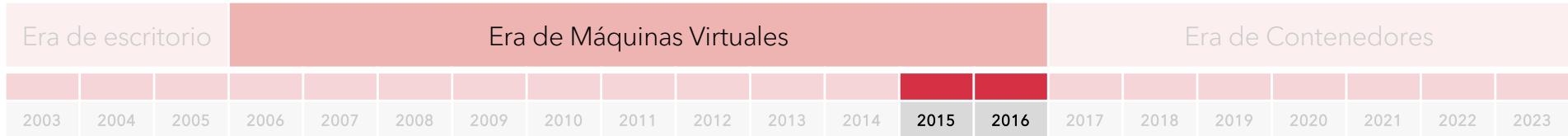
### Mascotas vs Ganado pets cattle

- Cada servidor es único y se le dedica una atención especial. Si un servidor falla, se hacen esfuerzos para restaurarlo rápidamente, a menudo tratando de solucionar el problema específico en ese servidor.
- Suelen tener nombres individuales y configuraciones personalizadas.
- Son costosos de mantener y pueden ser propensos a problemas de escalabilidad y disponibilidad,
- Los servidores son considerados recursos desechables y altamente replicables. Si un servidor falla, se reemplaza rápidamente por otro idéntico.
- Se gestionan mediante **automatización** y orquestación, y no se les da nombres individuales ni configuraciones personalizadas.
- Son fácilmente reemplazables y escalables



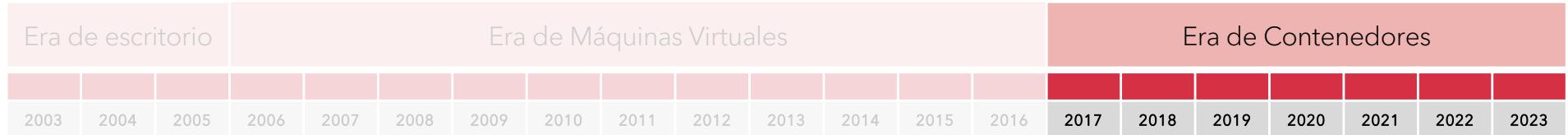


# Introducción





# Introducción



¿Realmente se necesita un Sistema Operativo virtual y su hardware virtual?



Máquina Virtual

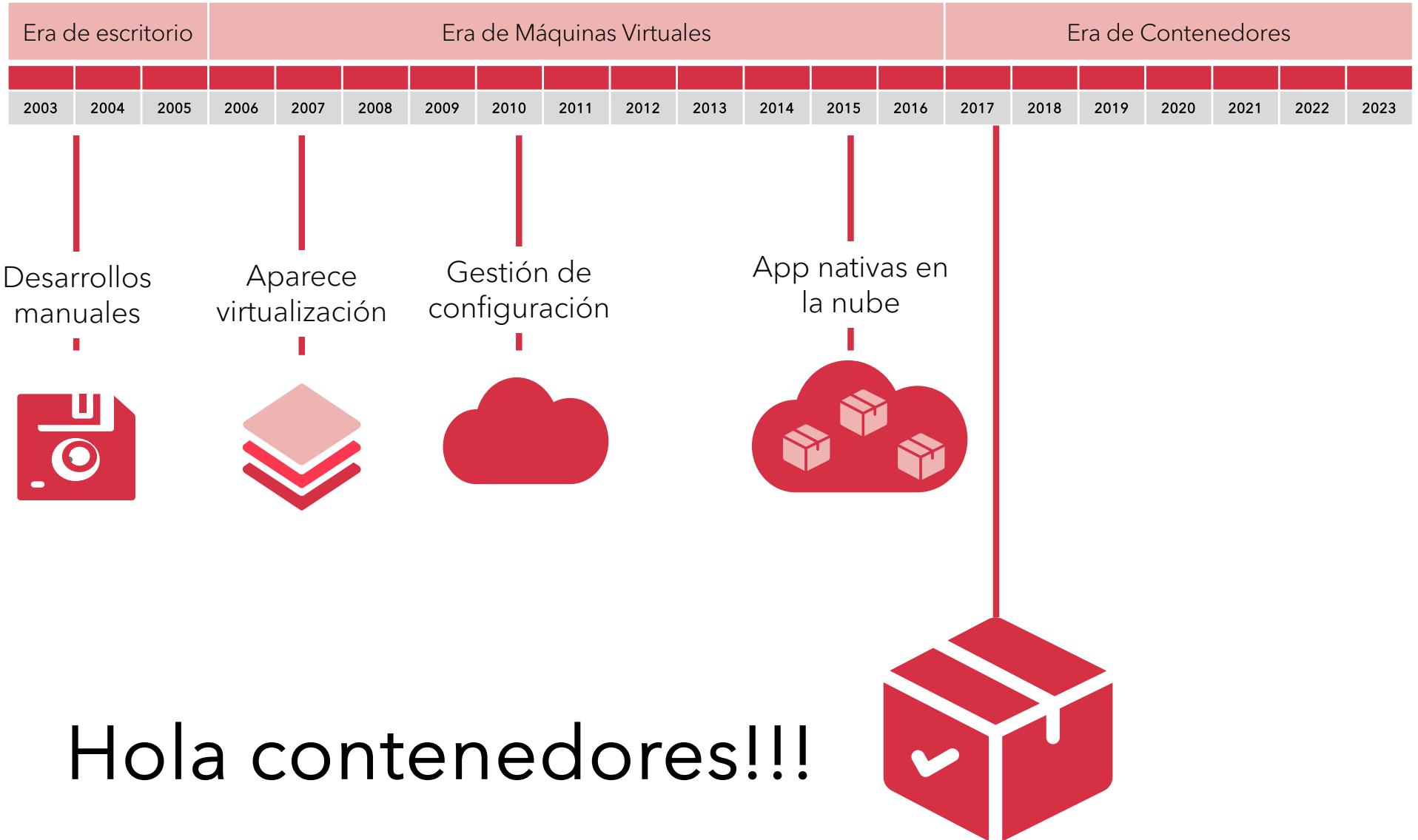


Contenedores



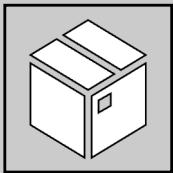


# Introducción





## ¿Qué son los contenedores?



Los **contenedores** son entornos aislados y ligeros que encapsulan aplicaciones y sus dependencias, lo que facilita la portabilidad y la ejecución consistente en diferentes entornos, desde el desarrollo local hasta la nube.



**Docker** se destaca como una de las plataformas más influyentes y populares para la gestión de contenedores

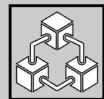




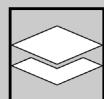
## ¿Qué es Docker?



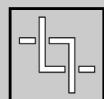
Docker es una plataforma para que desarrolladores y administradores puedan desarrollar, desplegar y ejecutar aplicaciones en un entorno aislado denominado contenedor.



Proyecto Open Source creado en 2013 que hace uso de LXC (Linux Containers). LXC es un método de virtualización a nivel de S.O.



Docker permite separar las aplicaciones de la infraestructura acelerando el proceso de entrega de software a producción.

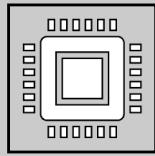
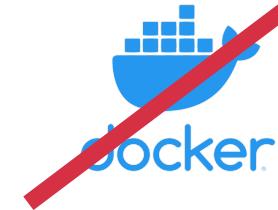


Docker permite empaquetar una aplicación con todas sus dependencias para que pueda ser ejecutada en plataformas diferentes.

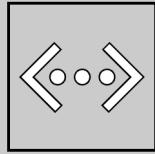




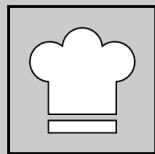
## ¿Qué no es Docker?



Docker no es virtualizado, no hay un hipervisor.



Los procesos que corren dentro de un contenedor de Docker se ejecutan con el mismo kernel que la máquina anfitrión.



Cuando ejecutamos un contenedor, estamos ejecutando un servicio dentro de una distribución construida a partir de una "receta".





## Docker vs Virtualización



**Aislamiento ligero:** Los contenedores Docker comparten el mismo núcleo del sistema operativo, lo que los hace más ligeros y rápidos.

**Recursos compartidos:** Los contenedores comparten recursos con el sistema anfitrión y entre sí, lo que maximiza la eficiencia en el uso de recursos.

**Tamaño pequeño:** Los contenedores son pequeños y solo contienen las dependencias necesarias, lo que reduce significativamente el tamaño y la sobrecarga.

**Arranque rápido:** Los contenedores Docker se inician rápidamente debido a la ausencia de un sistema operativo completo.

**Aislamiento completo:** Las máquinas virtuales (VMs) ofrecen un aislamiento completo, con sistemas operativos separados para cada VM.

**Recursos dedicados:** Cada VM tiene su propia asignación de recursos, como memoria y CPU, lo que puede resultar en un uso menos eficiente de los recursos físicos.

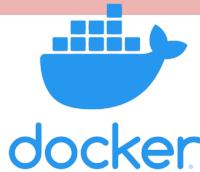
**Tamaño y sobrecarga:** Las VMs son más grandes en tamaño y requieren un sistema operativo completo, lo que conlleva una sobrecarga significativa.

**Arranque lento:** Las VMs pueden tardar más tiempo en arrancar debido a la necesidad de cargar un sistema operativo completo.





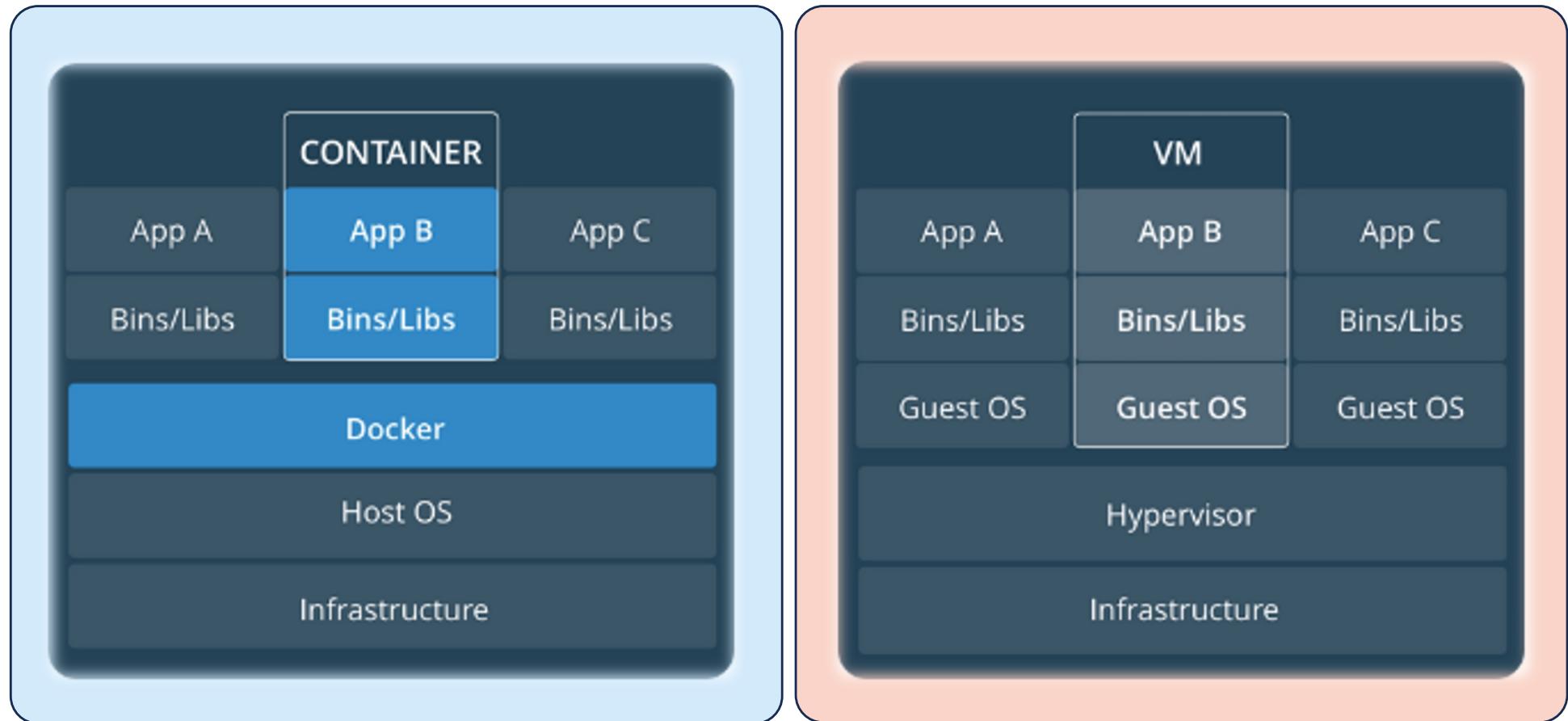
## Docker vs Virtualización

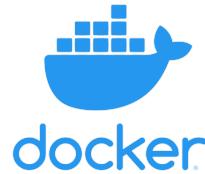


“Si ejecutamos contenedores GNU/Linux dentro de sistemas privativos, sí habrá virtualización”



Virtualización

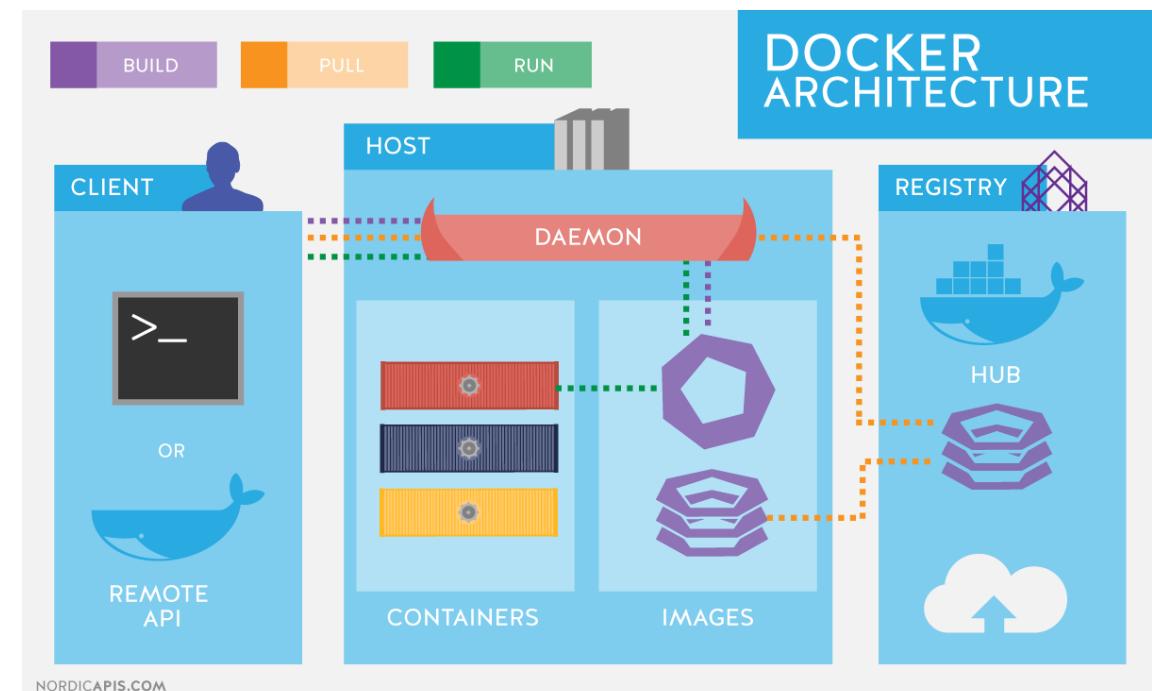


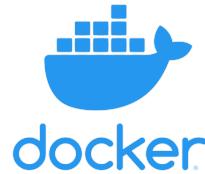


## Conceptos básicos

- ▶ **Imagen (image):** Plantilla de solo lectura que contiene las instrucciones para crear un contenedor Docker. Pueden estar basadas en otras imágenes.
- ▶ **Contenedor (container):** Es una instancia ejecutable de una imagen. Esta instancia puede ser creada, iniciada, detenida, movida o eliminada a través del cliente de Docker o de la API. Las instancias se pueden conectar a una o más redes, sistemas de almacenamiento.
- ▶ **Servicios (services):** Los servicios permiten escalar un contenedor a través de múltiples demonios de Docker, los cuales trabajarán conjuntamente como un enjambre (swarm)

### Objetos

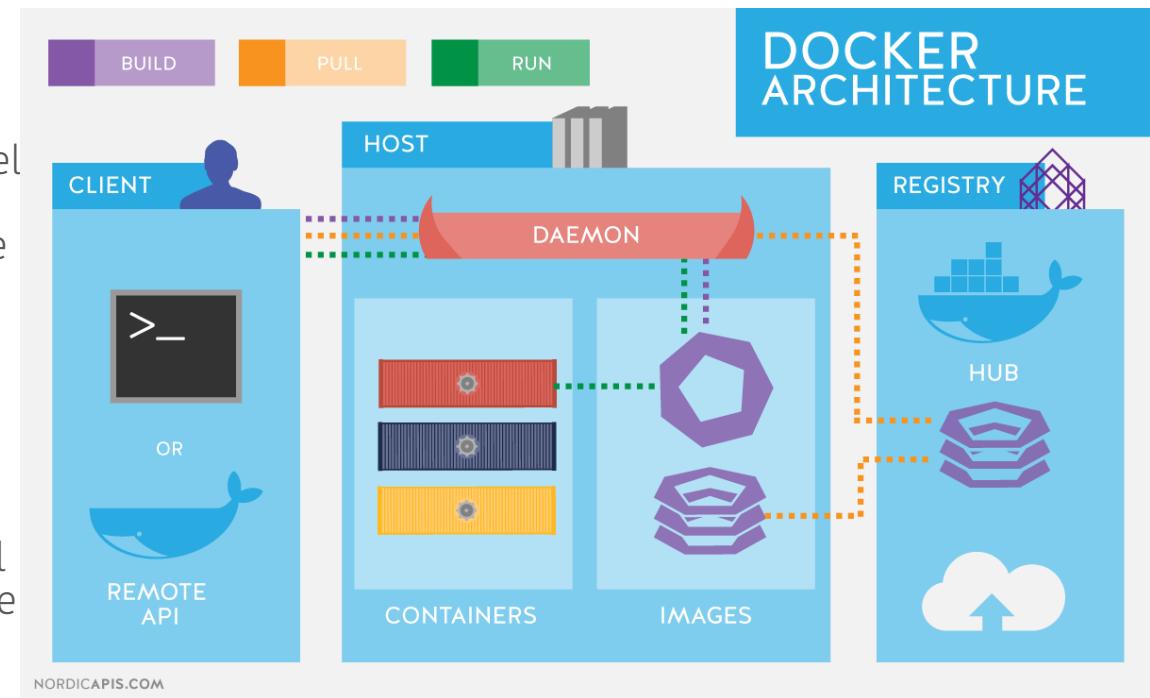




## Conceptos básicos

- ▶ **Cliente de docker (docker client):** Es la principal herramienta que usan los administradores de sistema para interaccionar con el sistema Docker.
- ▶ **Demonio de docker (docker daemon):** Es el proceso principal de docker. Escucha peticiones a la API y maneja los objetos de docker: imágenes, contenedores, redes, volúmenes. También es capaz de comunicarse con otros demonios para controlar servicios Docker.
- ▶ **Registro de docker (docker registry):** Es el lugar donde se almacenan las imágenes de Docker y poder descargarlas para reutilizarlas.

### Elementos

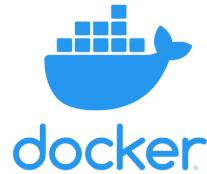


[Docker Hub](#) es el principal registro público de Docker y contiene gran diversidad de imágenes listas para ser usadas de multitud de servicios (mysql, wordpress, etc)..





# Introducción



## Conceptos básicos

### Docker Hub

<https://hub.docker.com/>

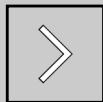
The screenshot shows the Docker Hub homepage with a blue header bar containing the Docker Hub logo, a search bar, and navigation links for Explore, Repositories, Organizations, and Help. On the right, there's an Upgrade button, a user profile icon for 'jmsoto', and a dropdown menu. Below the header, there are five large cards for popular official repositories: NGINX, mongoDB, alpine, node, and redis. The main content area displays a grid of smaller repository cards:

busybox Official ↓ 1B+	ubuntu Official ↓ 1B+	python Official ↓ 1B+	postgres Official ↓ 1B+	httpd Official ↓ 1B+
memcached Official ↓ 1B+	mysql Official ↓ 1B+	traefik Official ↓ 1B+	mariadb Official ↓ 1B+	docker Official ↓ 1B+

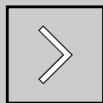




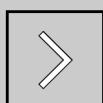
## ¿Qué ventajas aporta Docker?



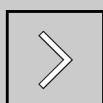
**Portabilidad:** Las aplicaciones y sus dependencias se empaquetan en contenedores, lo que garantiza la misma ejecución en cualquier entorno.



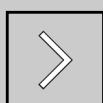
**Eficiencia:** Los contenedores comparten recursos, lo que los hace eficientes en el uso de memoria y CPU.



**Isolación:** Ofrecen alto nivel de aislamiento entre aplicaciones, evitando conflictos y problemas de dependencias.



**Despliegue rápido:** Facilita implementaciones rápidas, acelerando el desarrollo.

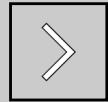


**Escalabilidad:** Permite escalar aplicaciones horizontalmente de manera sencilla con múltiples instancias de contenedores.

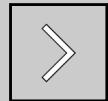




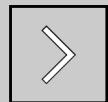
## ¿Qué ventajas aporta Docker?



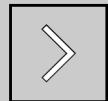
**Versionamiento y control de cambios:** Facilita el seguimiento de cambios y la reversión.



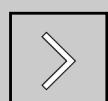
**Colaboración:** Mejora la colaboración entre equipos de desarrollo y operaciones.



**Integración:** Se integra con herramientas de orquestación como Kubernetes permitiendo orquestar contenedores en clústeres



**Ecosistema robusto:** Ofrece imágenes preconstruidas en Docker Hub.



**Seguridad:** Proporciona características de seguridad como aislamiento y gestión de identidad protegiendo aplicaciones y datos.





# Introducción. Instalación



Docker CE

Existen dos versiones de Docker, una libre y otra que no lo es. Nos ocuparemos exclusivamente de la primera: Docker CE (Community Edition).

<https://docs.docker.com/get-docker/>

Docker CE está disponible para los siguientes sistemas GNU/Linux: CentOS, Debian, Fedora , Raspbian y Ubuntu. Y esta soportado por arquitecturas como x86\_64 / amd64 , ARM y ARM64 / AARCH64.

Debido a que, dependiendo de la distribución, la forma de instalarlo difiere, es mejor consultar la documentación oficial para saber cómo instalar Docker en tu máquina.



Docker Desktop for Mac



Docker Desktop for Windows



Docker Desktop for Linux





# Índice

**01**

## Introducción

- Desde inicio hasta actualidad
- Qué es Docker
- Conceptos básicos
- Instalación

**02**

## Cliente en Docker

- Hola mundo
- Operaciones
- Comandos básicos

**03**

## Datos en Docker

- Docker volume
- Docker bind mount

## Redes en Docker

- Tipos de redes
- Gestionando redes
- Uso de redes definidas por usuario

**04**

## Imágenes en Docker

- Gestionando imágenes
- Creación de imágenes propias. Dockerfile
- Distribución de imágenes

**05**

## Escenarios multi-contenedor

- Escenarios vs orquestación
- Docker compose. Comandos
- Fichero docker-compose.yml
- Almacenamiento en Docker compose
- Redes en Docker compose

**06**

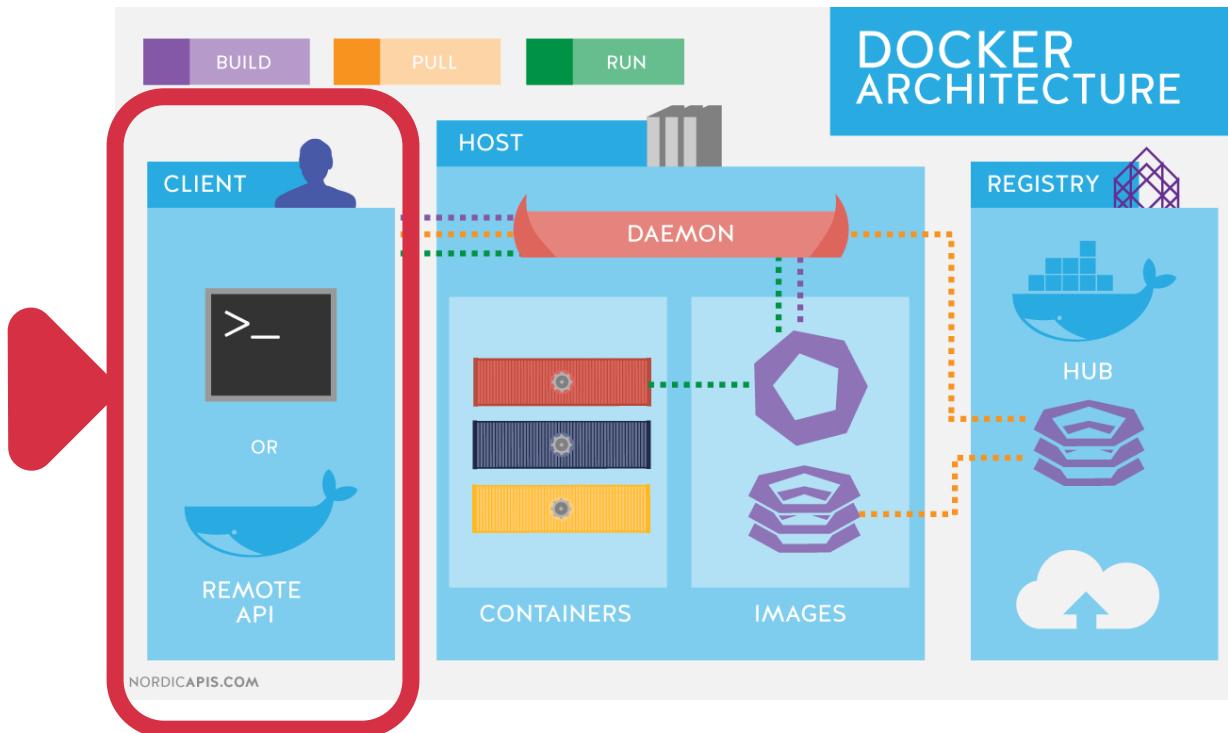




# Cliente. Funcionamiento básico



Docker CE





# Cliente. Funcionamiento básico



Docker CE

Hola mundo!

```
$ sudo docker container run hello-world
```

```
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
70f5ac315c5a: Pull complete
Digest: sha256:4f53e2564790c8e7856ec08e384732aa38dc43c52f02952483e3f003afb23db
Status: Downloaded newer image for hello-world:latest
```

Hello from Docker!

This message shows that your installation appears to be working correctly.

	Name	Image	Status	CPU (%)	Port(s)	Last started	Actions
<input type="checkbox"/>	<a href="#">determined</a>	<a href="#">hello-world</a>	Exited	N/A		19 minutes ago	





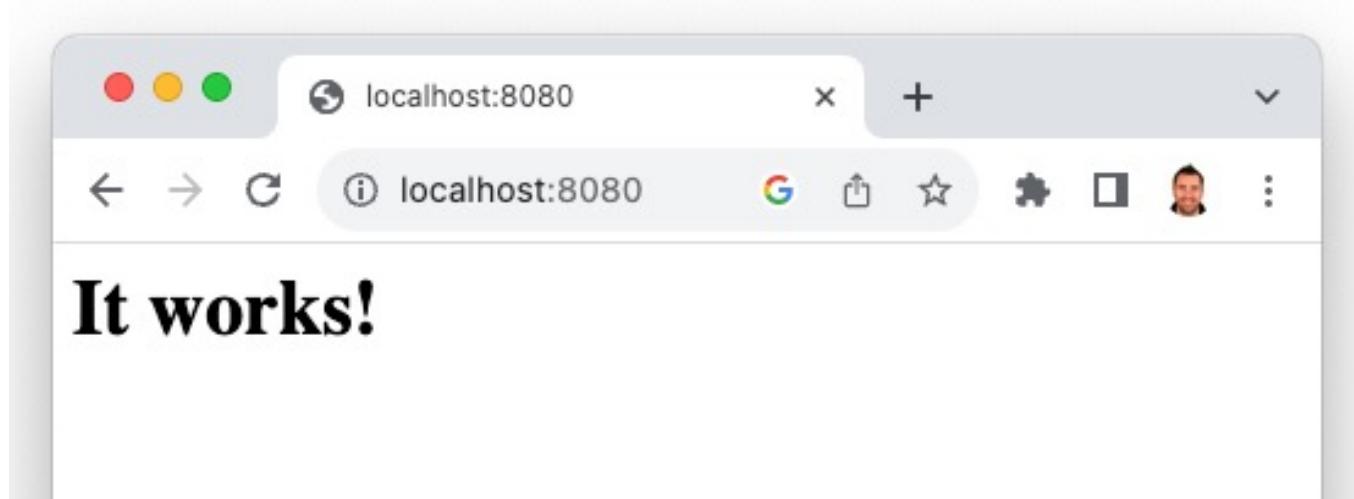
# Cliente. Funcionamiento básico



## Crear un contenedor Apache

```
$ docker container run -d -p 8080:80 httpd
```

Descarga una imagen si no existe localmente, lanza un contenedor y asocia el puerto 8080 del host al puerto 80 del contenedor





# Cliente. Funcionamiento básico



## Operaciones sobre contenedores

- Mostrar contenedores. Sin la opción **-a** solo se muestran contenedores que no estén detenidos

```
$ docker container ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
d29e2ba8af1d	httpd	"httpd-foreground"	6 minutes ago	Up 6 minutes	0.0.0.0:8080->80/tcp	keen_noyle

- Detener y reanudar contenedores. Necesario el CONTAINER ID

```
$ docker container stop d29e2ba8af1d
```

```
$ docker container start d29e2ba8af1d
```

- Detener todos los contenedores en ejecución

```
$ docker container stop `docker ps -q`
```





# Cliente. Funcionamiento básico



## Operaciones sobre contenedores

- Abrir un terminal en un contenedor

```
$ docker container exec -it d29e2ba8af1d /bin/bash  
root@d29e2ba8af1d:/usr/local/apache2# ls  
bin build cgi-bin conf error htdocs icons include logs modules
```

- Copiar datos. NOTA: El almacenamiento en un contenedor no es persistente. Se eliminan los datos escritos en él tras su eliminación.

- Del contendor a local

```
$ docker container cp NOMBRE_CONTENEDOR:RUTA_DEL_CONTENEDOR RUTA_LOCAL
```

- De local al contendor

```
$ docker container cp RUTA_LOCAL NOMBRE_CONTENEDOR:RUTA_DEL_CONTENEDOR
```





# Cliente. Funcionamiento básico

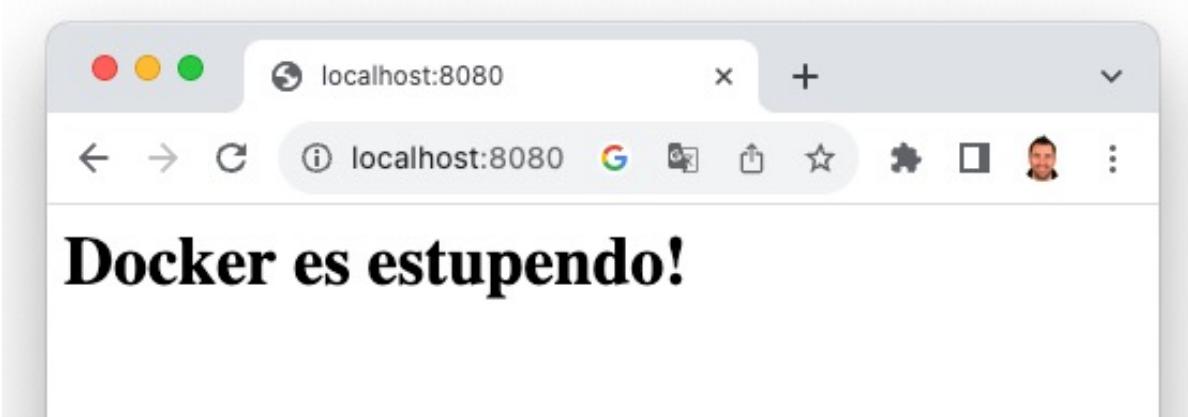


## Operaciones sobre contendores

Crear un archivo `index.html` en local y vamos a copiarlo al contenedor

```
$ nano index.html
<!-- Ejemplo de archivo index.html -->
<html>
  <body>
    <h1>Docker es estupendo!</h1>
  </body>
</html>
```

```
$ docker container cp index.html d29e2ba8af1d:/usr/local/apache2/htdocs/
Successfully copied 2.05kB to d29e2ba8af1d:/usr/local/apache2/htdocs/
```





# Cliente. Funcionamiento básico



## Operaciones sobre contenedores

▶ Eliminar un contenedor. Primero se detiene y luego se elimina.

```
$ docker container stop d29e2ba8af1d
```

```
$ docker container rm d29e2ba8af1d
```

▶ Eliminar contenedor en ejecución forzando su eliminación

```
$ docker container rm -f d29e2ba8af1d
```

NOTA: Los datos almacenados en el contenedor desaparecen

El index.html que hemos creado y copiado en el contenedor con Apache ha desaparecido





# Cliente. Funcionamiento básico



## Operaciones sobre contendores. Resumen de comandos básicos

```
$ docker container info
```

```
Commands:
  attach      Attach local standard input, output, and error streams to a running container
  commit      Create a new image from a container's changes
  cp          Copy files/folders between a container and the local filesystem
  create      Create a new container
  diff        Inspect changes to files or directories on a container's filesystem
  exec        Execute a command in a running container
  export      Export a container's filesystem as a tar archive
  inspect    Display detailed information on one or more containers
  kill        Kill one or more running containers
  logs        Fetch the logs of a container
  ls          List containers
  pause       Pause all processes within one or more containers
  port        List port mappings or a specific mapping for the container
  prune      Remove all stopped containers
  rename     Rename a container
  restart    Restart one or more containers
  rm         Remove one or more containers
  run         Create and run a new container from an image
  start      Start one or more stopped containers
  stats      Display a live stream of container(s) resource usage statistics
  stop       Stop one or more running containers
  top        Display the running processes of a container
  unpause   Unpause all processes within one or more containers
  update     Update configuration of one or more containers
  wait       Block until one or more containers stop, then print their exit codes
```





# Cliente. Funcionamiento básico



## Operaciones sobre contenedores. Resumen de comandos básicos

```
$ docker container run <image>: Crea un contenedor a partir de una imagen. Si no tenemos la imagen en local, la descarga
```

```
$ docker container run -d -p 8080:80 apache: Crea un contenedor en modo detached (independiente, en segundo plano) accesible desde el puerto 8080
```

```
$ docker container stop|start <id>: Detiene|Arranca un contenedor
```

```
$ docker container ps -a: Listado de contenedores (-a muestra también los parados)
```

```
$ docker container ps -q: Listado de los ids de los contenedores
```

```
$ docker container stop `docker ps -q`: Para todos los contenedores que devuelve el subcomando `docker ps -q`
```

```
$ docker container rm <id>: Borra un contenedor si está parado
```

```
$ docker container rm -f <id>: Fuerza el borrado de un contenedor aunque esté parado
```

```
$ docker container exec -it <id> sh: Abre una terminal en el contenedor
```

```
$ docker container exec <id> ls: Ejecuta el comando ls en el contenedor para mostrar sus archivos
```

```
$ docker container cp <id>:/file ..::/file: Copia el fichero file del contenedor en nuestro sistema de archivos local
```





# Cliente. Funcionamiento básico



## Operaciones sobre contendores

### Operaciones con Docker Desktop. Interfaz gráfica

Name	Image	Status	CPU (%)	Port(s)	Last started	Actions
<a href="#">keen_noxy</a> d29e2ba8af1d	<a href="#">httpd</a>	Running	0.01%	<a href="#">8080:80</a>	6 minutes ago	

- View details
- View image packages and CVEs
- Copy docker run
- Open in terminal
- View files
- Pause
- Restart
- Open with browser





# Cliente. Funcionamiento básico



## Operaciones sobre contendores

### Visualizar Logs

The screenshot shows the Docker interface for a container named 'keen\_nooyce'. The container is running an Apache web server ('httpd') on port 8080. The logs tab is selected, displaying the following log entries:

```
AH00558: httpd: Could not reliably determine the server's fully qualified domain name, using 172.17.0.2. Set the 'ServerName' directive globally to suppress this message
AH00558: httpd: Could not reliably determine the server's fully qualified domain name, using 172.17.0.2. Set the 'ServerName' directive globally to suppress this message
[Mon Oct 09 09:50:23.433583 2023] [mpm_event:notice] [pid 1:tid 281473517891616] AH00489: Apache/2.4.57 (Unix) configured -- resuming normal operations
[Mon Oct 09 09:50:23.433644 2023] [core:notice] [pid 1:tid 281473517891616] AH00094: Command line: 'httpd -D FOREGROUND'
[Mon Oct 09 10:02:44.367604 2023] [mpm_event:notice] [pid 1:tid 281473517891616] AH00492: caught SIGWINCH, shutting down gracefully
AH00558: httpd: Could not reliably determine the server's fully qualified domain name, using 172.17.0.2. Set the 'ServerName' directive globally to suppress this message
AH00558: httpd: Could not reliably determine the server's fully qualified domain name, using 172.17.0.2. Set the 'ServerName' directive globally to suppress this message
[Mon Oct 09 10:19:36.006961 2023] [mpm_event:notice] [pid 1:tid 281472870977568] AH00489: Apache/2.4.57 (Unix) configured -- resuming normal operations
[Mon Oct 09 10:19:36.009545 2023] [core:notice] [pid 1:tid 281472870977568] AH00094: Command line: 'httpd -D FOREGROUND'
192.168.65.1 - - [09/Oct/2023:09:50:38 +0000] "GET / HTTP/1.1" 200 45
192.168.65.1 - - [09/Oct/2023:09:50:38 +0000] "GET /favicon.ico HTTP/1.1" 404 196
192.168.65.1 - - [09/Oct/2023:09:51:29 +0000] "-" 408 -
192.168.65.1 - - [09/Oct/2023:10:19:44 +0000] "GET / HTTP/1.1" 304 -
192.168.65.1 - - [09/Oct/2023:10:20:36 +0000] "-" 408 -
```





# Cliente. Funcionamiento básico



## Operaciones sobre contendores

### Inspeccionar elementos

The screenshot shows the Docker interface with the container `keen_noyce` selected. The `Logs` tab is active, but the `Inspect` tab is currently selected. The `Raw JSON` button is checked. The JSON output is as follows:

```
16      "StartedAt": "2023-10-09T10:19:35.975065172Z",
17      "FinishedAt": "2023-10-09T10:02:45.392724377Z"
18    },
19    "Image": "sha256:ce6083df2933edb2dal320f39fc84e39f430115f7b5e6e52b83a0924d711a58",
20    "ResolvConfPath": "/var/lib/docker/containers/d29e2ba8af1de0f3cc5041fea91f9ff5e5ac04dc9eb82f3b21f76169780c2937/resolv.conf",
21    "HostnamePath": "/var/lib/docker/containers/d29e2ba8af1de0f3cc5041fea91f9ff5e5ac04dc9eb82f3b21f76169780c2937/hostname",
22    "HostsPath": "/var/lib/docker/containers/d29e2ba8af1de0f3cc5041fea91f9ff5e5ac04dc9eb82f3b21f76169780c2937/hosts",
23    "LogPath": "/var/lib/docker/containers/d29e2ba8af1de0f3cc5041fea91f9ff5e5ac04dc9eb82f3b21f76169780c2937/d29e2ba8af1de0f3cc5041fea91f9f
24    "Name": "/keen_noyce",
25    "RestartCount": 0,
26    "Driver": "overlay2",
27    "Platform": "linux",
28    "MountLabel": "",
29    "ProcessLabel": "",
30    "AppArmorProfile": "",
31    "ExecIDs": [
32      "7aa557868ae5a42c593878cd3cb4903ff071b9725637bf7112d93380dd657791"
33    ],
34    "HostConfig": {
35      "Binds": null,
36      "ContainerIDFile": ""
    }
```





# Cliente. Funcionamiento básico



## Operaciones sobre contendores

### Ejecutar comandos consola

The screenshot shows the Docker interface for a container named 'keen\_nooyce'. The container is running an 'httpd' service, as indicated by the icon and port mapping '8080:80'. The status is 'Running (15 minutes ago)'. The 'Exec' tab is selected, showing a terminal session with the command '# ls -la' executed. The output of the command is displayed below:

```
# ls -la
total 3492
drwxr-xr-x 2 root      root        4096 Sep 20 09:46 .
drwxr-xr-x 1 www-data  www-data    4096 Sep 20 09:46 ..
-rw xr-xr-x 1 root      root     172288 Sep 20 09:46 ab
-rw xr-xr-x 1 root      root      3437 Sep 20 09:45 apachectl
-rw xr-xr-x 1 root      root     23879 Sep 20 09:45 apxs
-rw xr-xr-x 1 root      root    76384 Sep 20 09:46 checkgid
-rw xr-xr-x 1 root      root     8925 Sep 20 09:45 dbmmanage
-rw r--r-- 1 root      root     1073 Sep 20 09:45 envvars
-rw r--r-- 1 root      root     1073 Sep 20 09:45 envvars-std
-rw xr-xr-x 1 root      root    83360 Sep 20 09:46 fcgistarter
-rw xr-xr-x 1 root      root   114144 Sep 20 09:46 htcachecl
-rw xr-xr-x 1 root      root   101688 Sep 20 09:46 htdbm
-rw xr-xr-x 1 root      root    83040 Sep 20 09:46 htdigest
-rw xr-xr-x 1 root      root    97680 Sep 20 09:46 htpasswd
-rw xr-xr-x 1 root      root  2407192 Sep 20 09:46 httpd
-rw xr-xr-x 1 root      root    80448 Sep 20 09:46 httx2dbm
-rw xr-xr-x 1 root      root    83552 Sep 20 09:46 logresolve
-rw xr-xr-x 1 root      root  103600 Sep 20 09:46 rotatelogs
-rwsr-xr-x 1 root      root    89784 Sep 20 09:46 suexec
#
```





# Cliente. Funcionamiento básico



## Operaciones sobre contendores

Navegar por archivos

The screenshot shows the Docker Desktop interface with a container named "keen\_nooyce" running. The "Files" tab is selected in the navigation bar. On the left, a tree view shows the directory structure of the container's file system, including .dockerenv, bin, boot, dev, etc, home, lib, media, mnt, opt, and proc. On the right, a table lists the files and directories with their details:

Name	Note	Size	Last modified	Mode
.dockerenv		0 Bytes	48 minutes ago	-rwxr-xr-x
bin -> usr/bin		7 Bytes	20 days ago	Lrwxrwxrwx
boot			3 months ago	drwxr-xr-x
dev			19 minutes ago	drwxr-xr-x
etc			48 minutes ago	drwxr-xr-x
home			3 months ago	drwxr-xr-x
lib -> usr/lib		7 Bytes	20 days ago	Lrwxrwxrwx
media			20 days ago	drwxr-xr-x
mnt			20 days ago	drwxr-xr-x
opt			20 days ago	drwxr-xr-x
proc			19 minutes ago	dr-xr-xr-x



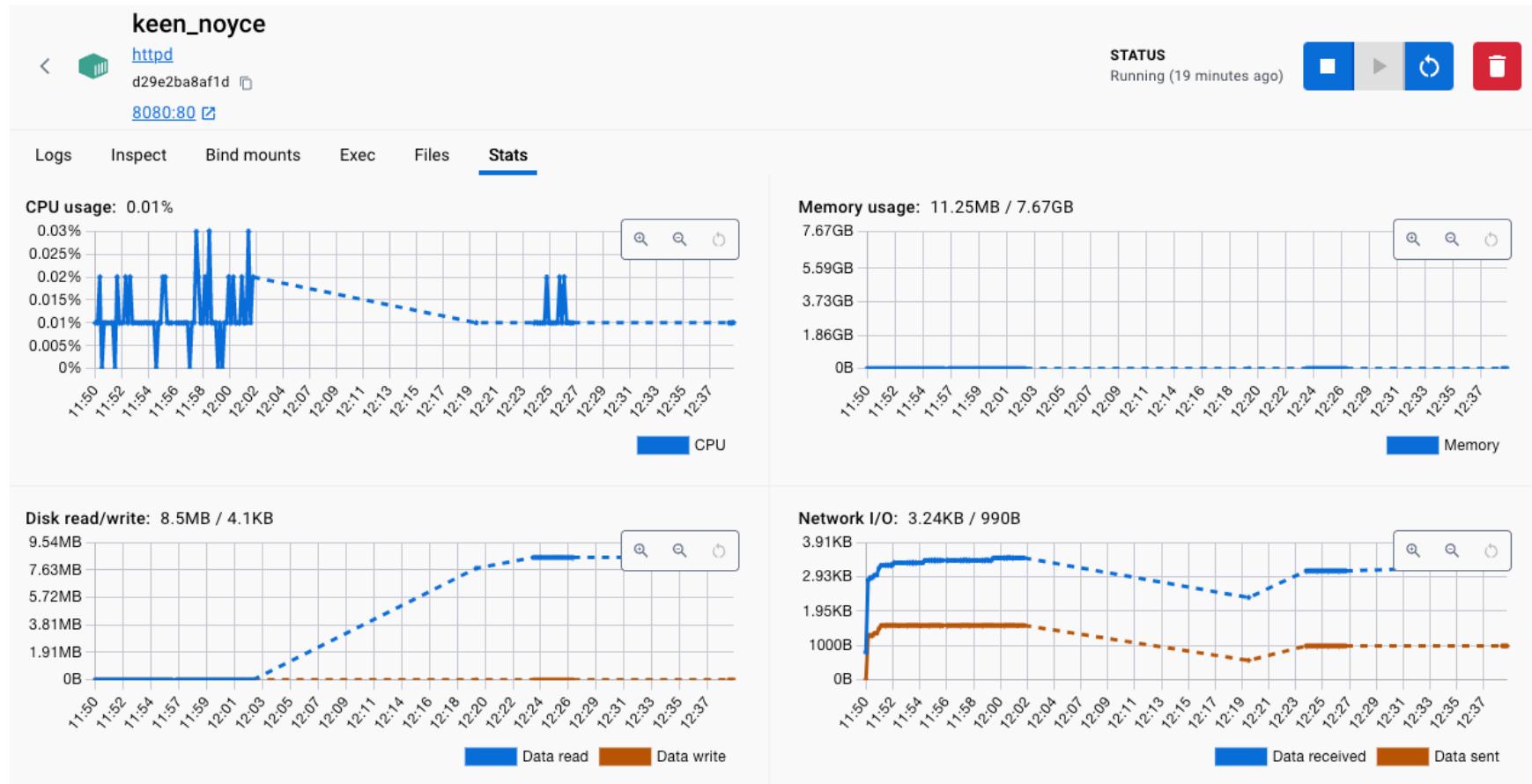


# Cliente. Funcionamiento básico



## Operaciones sobre contendores

### Estadísticas del contenedor





# Índice

**01**

## Introducción

- Desde inicio hasta actualidad
- Qué es Docker
- Conceptos básicos
- Instalación

**02**

## Cliente en Docker

- Hola mundo
- Operaciones
- Comandos básicos

**03**

## Datos en Docker

- Docker volume
- Docker bind mount

## Redes en Docker

- Tipos de redes
- Gestionando redes
- Uso de redes definidas por usuario

**04**

## Imágenes en Docker

- Gestionando imágenes
- Creación de imágenes propias. Dockerfile
- Distribución de imágenes

**05**

## Escenarios multi-contenedor

- Escenarios vs orquestación
- Docker compose. Comandos
- Fichero docker-compose.yml
- Almacenamiento en Docker compose
- Redes en Docker compose

**06**

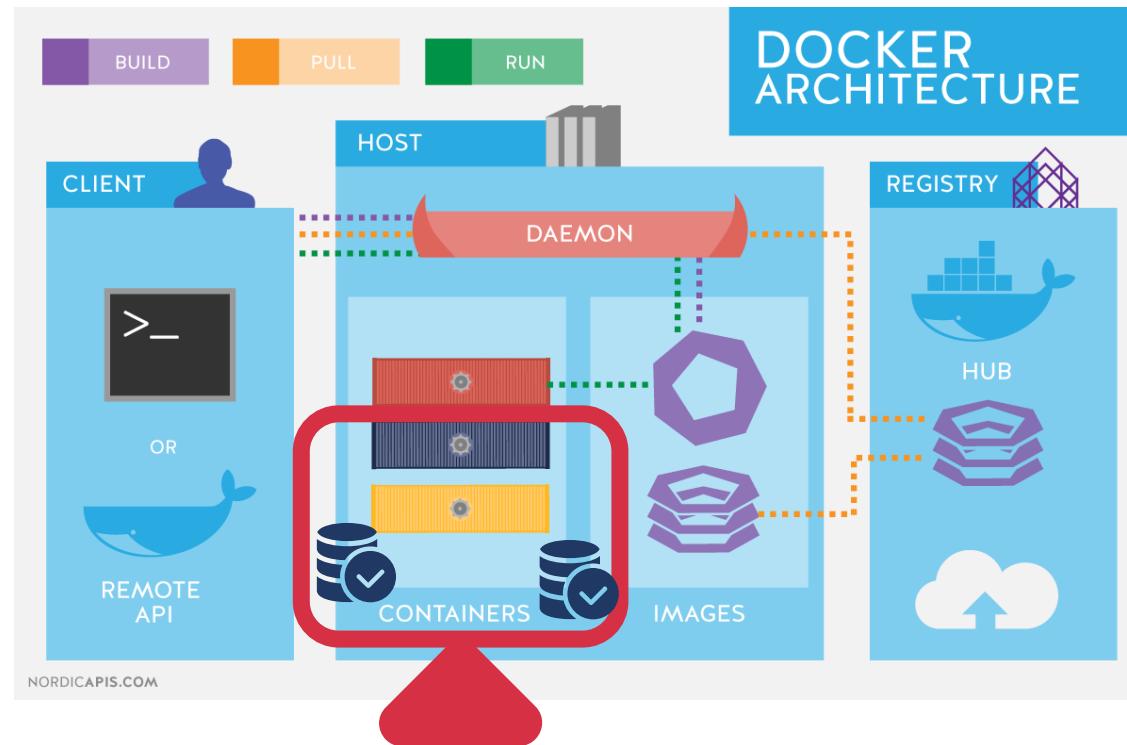




# Almacenamiento



## Docker CE



Los contenedores son efímeros, es decir, los datos, configuraciones, archivos, etc. que se crean en los contenedores persisten hasta que el contenedor se elimina. Persisten a arranques y paradas pero no a eliminaciones.





# Almacenamiento



## Persistiendo datos

Docker ofrece tres opciones de almacenar datos:

- Volúmenes: **docker volume** (objetos Docker)
- Montando un directorio: **docker bind mount** (un directorio de la máquina anfitrión dentro del contenedor)
- Almacenándolo en memoria: **docker tmpfs mounts** (se perderían al reiniciar el anfitrión)

Recomendaciones: Por ejemplo, para guardar los datos de una base de datos podemos usar volúmenes, pero para guardar el código de una aplicación o de una página web montaremos el directorio.

La razón es que tanto nuestro entorno de desarrollo como el contenedor tengan acceso a los archivos del código fuente. Los volúmenes, al contrario que los directorios montados, no deben accederse desde la máquina anfitrión.





# Almacenamiento



## Persistiendo datos

Docker ofrece tres opciones de almacenar datos:

- Volúmenes: **docker volume** (objetos Docker)
- Montando un directorio: **docker bind mount** (un directorio de la máquina anfitrión dentro del contenedor)
- Almacenándolo en memoria: **docker tmpfs mounts** (se perderían al reiniciar el anfitrión)

### Volume



- Para compartir datos entre contenedores. Simplemente tendrán que usar el mismo volumen.
- Para copias de seguridad ya sea para que sean usadas posteriormente por otros contenedores o para mover esos volúmenes a otros hosts.
- Para almacenar los datos de mi contenedor no localmente si no en un proveedor cloud.





# Almacenamiento



## Persistiendo datos

Docker ofrece tres opciones de almacenar datos:

- Volúmenes: **docker volume** (objetos Docker)
- Montando un directorio: **docker bind mount** (un directorio de la máquina anfitrión dentro del contenedor)
- Almacenándolo en memoria: **docker tmpfs mounts** (se perderían al reiniciar el anfitrión)

### bind mount



- Para montar tanto directorios como ficheros.
- Compartir ficheros entre el host y los contenedores.
- Para que otras aplicaciones que no sean Docker tengan acceso a esos ficheros, ya sean código, ficheros etc.





# Almacenamiento



## Persistiendo datos

Docker ofrece tres opciones de almacenar datos:

- Volúmenes: **docker volume** (objetos Docker)
- Montando un directorio: **docker bind mount** (un directorio de la máquina anfitrión dentro del contenedor)
- Almacenándolo en memoria: **docker tmpfs mounts** (se perderían al reiniciar el anfitrión)

### tmpfs mount

- Para evitar la escritura de los datos e información dentro de la capa de almacenamiento del contenedor.
- Solo puede cumplir sus funciones si el usuario ejecuta la plataforma de Docker en el entorno de Linux.
- No tiene la opción de compartirse entre más de un contenedor.





# Almacenamiento



## Docker volume. Comandos básicos

- ▶ Crear un volumen con el nombre indicado: **docker volume create <nombre>**
- ▶ Eliminar un volumen con el nombre indicado: **docker volume rm <nombre>**
- ▶ Eliminar volúmenes que no están siendo usados por ningún contenedor:  
**docker volume prune**
- ▶ Listar los volúmenes creados: **docker volume ls**
- ▶ Información detallada del volumen con el nombre indicado:  
**docker volume inspect <nombre>**



Si se usan imágenes de DockerHub es importante leer la información relativa a esa imagen en cuanto a la persistencia de datos. Se suele indicar cómo persistir los datos de esa imagen y cuáles son las carpetas importantes en caso de ser imágenes que contengan ciertos servicios (web, base de datos etc...)





# Almacenamiento



## Docker volume

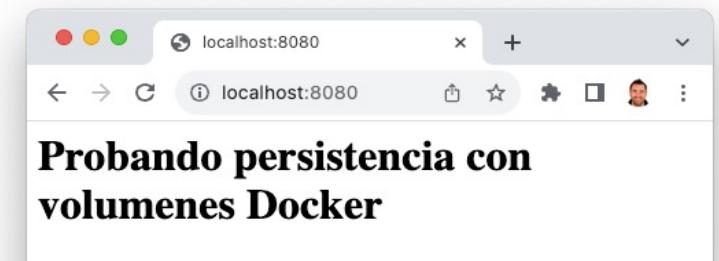
Ejemplo: Persistencia de datos web creando un volumen y usando flag `--v` y `--mount`

Creamos un volumen Docker llamado `mivolumenweb`

```
$ docker volume create mivolumenweb  
mivolumenweb
```

Creamos un contenedor `miapacheapp` con el volumen `mivolumenweb`, usando `--mount`, y creamos un fichero `index.html`:

```
$ docker run -d --name miapacheapp --mount  
type=volume,src=mivolumenweb,dst=/usr/local/apache2/htdocs -p 8080:80 httpd:2.4  
907178a0c8fb332588d1b0e4e186632a6b19870bc88a2b8b5328c0ae637eaf24  
  
$ docker exec miapacheapp bash -c 'echo "<h1>Probando persistencia con  
volúmenes Docker</h1>" > /usr/local/apache2/htdocs/index.html'
```





# Almacenamiento

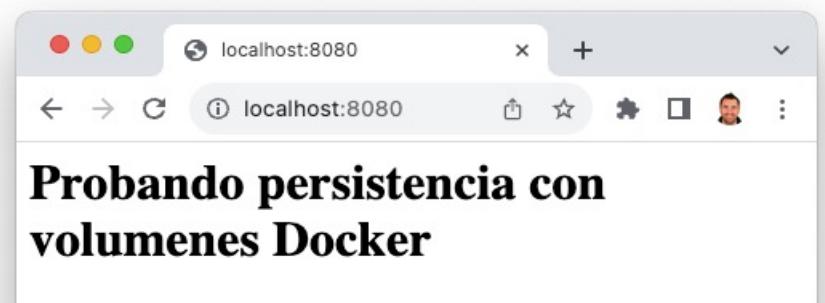


## Docker volume

Ejemplo: Persistencia de datos web creando un volumen y usando flag `--v` y `--mount`

Eliminamos el contenedor `miapacheapp` y creamos un nuevo contenedor con el volumen `mivolumenweb` asociado pero ahora con el parámetro `-v`

```
$ docker rm -f miapacheapp  
miapacheapp  
  
$ docker run -d --name miapacheapp2 -v  
mivolumenweb:/usr/local/apache2/htdocs -p 8080:80 httpd:2.4  
1d21c2e6c413ba5dd968a1a76bcea656bd972e0051bb51c40eaf6187af6907dd
```





# Almacenamiento



## Docker volume

Ejemplo: Persistencia de datos web creando un volumen y usando flag `--v` y `--mount`

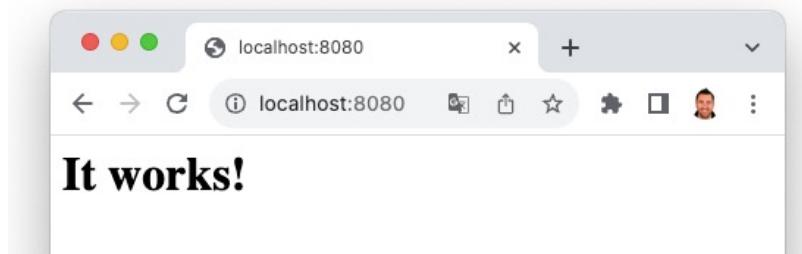


### Aclaraciones

Si no se indica `src=mivolumenweb`

Si no se indica el volumen, se creará un nuevo volumen y no persistirán los datos.

```
$ docker run -d --name miapacheapp3 --mount  
type=volume,dst=/usr/local/apache2/htdocs -p 8080:80 httpd:2.4  
2f1cbf7e42a1ecbb01b863745c2cb985772ae33e762a7bd0c5d15868ff899648  
  
$ docker volume ls  
DRIVER      VOLUME NAME  
local      07760f41eed095079d371067ebd66c156c56af027697c4b8fad49ac7798019ba  
local      mivolumenweb
```





# Almacenamiento



## Docker volume

Ejemplo: Persistencia de datos web creando un volumen y usando flag `--v` y `--mount`



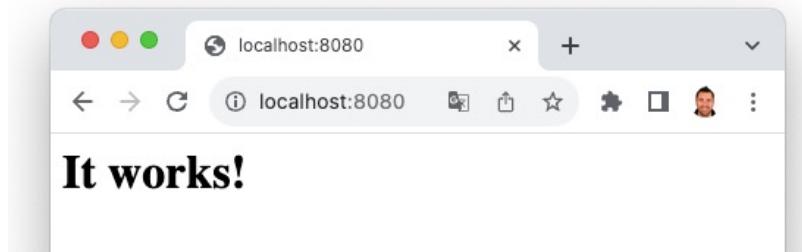
### Aclaraciones

Si se indica otro nombre con `-v`

Si usamos el flag `-v` e indicamos un nombre, se creará un volumen docker nuevo.

```
$ docker run -d --name miapacheapp3 -v  
otrovolumenweb:/usr/local/apache2/htdocs -p 8080:80 httpd:2.4  
40944374e00c14f494939d9c2ea7e571ec73b5f4f8e71de6b3dd1f36cb8ef15f
```

```
$ docker volume ls  
DRIVER      VOLUME NAME  
local       mivolumenweb  
local       otrovolumenweb
```





# Almacenamiento



## Docker bind mount

Ejemplo: Persistencia de datos web montando directorios con `bind mount`

Creamos un directorio en el host y un fichero index.html

```
$ mkdir ejemplo_bindmount  
$ cd ejemplo_bindmount  
$ echo "<h1>Probando persistencia de datos con bind mount en Docker</h1>" > index.html
```

Montamos el directorio `ejemplo_bindmount` en el contenedor `miapacheapp_mount` con la opción `-v`

```
$ docker run -d --name miapacheapp_mount -v  
/home/jmsoto/ejemplo_bindmount:/usr/local/apache2/htdocs -p 8080:80 httpd:2.4  
a7d0c9e97a3b8d343a5ae920b900847d780ef9bbbfdaff2dcda81b6b9efc142
```





# Almacenamiento



## Docker bind mount

Ejemplo: Persistencia de datos web montando directorios con `bind mount`

Eliminamos el contenedor `miapacheapp_mount` y creamos otro contenedor usando la opción `--mount`

```
$ docker container rm -f miapacheapp_mount  
miapacheapp_mount  
  
$ docker run -d --name miapacheapp_mount --mount  
type=bind,src=/home/jmsoto/ejemplo_bindmount,dst=/usr/local/apache2/htdocs  
-p 8080:80 httpd:2.4
```



Cualquier modificación en el directorio `ejemplo_bindmount` quedará reflejado en el contenedor





# Índice

01

## Introducción

- Desde inicio hasta actualidad
- Qué es Docker
- Conceptos básicos
- Instalación

02

## Cliente en Docker

- Hola mundo
- Operaciones
- Comandos básicos

03

## Datos en Docker

- Docker volume
- Docker bind mount

## Redes en Docker

- Tipos de redes
- Gestionando redes
- Uso de redes definidas por usuario

04

## Imágenes en Docker

- Gestionando imágenes
- Creación de imágenes propias. Dockerfile
- Distribución de imágenes

05

## Escenarios multi-contenedor

- Escenarios vs orquestación
- Docker compose. Comandos
- Fichero docker-compose.yml
- Almacenamiento en Docker compose
- Redes en Docker compose

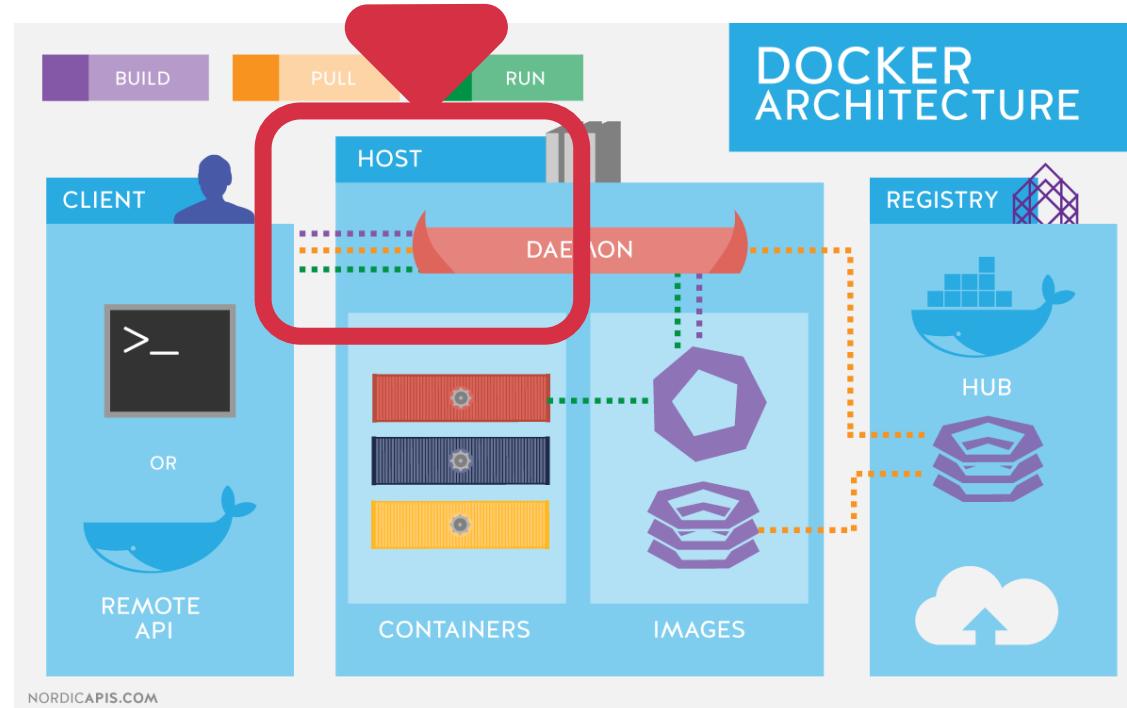
06



# Redes. Funcionamiento básico



## Docker CE



Cuando se crea un contenedor, éste se conecta a una red virtual y Docker hace una configuración del sistema (usando interfaces puente e iptables)





## Tipos de redes en Docker

### Bridge network (red puente)

Red predeterminada. Los contenedores conectados a esta red que quieren exponer algún puerto al exterior tienen que usar la opción `-p` para mapear puertos. Los contenedores pueden comunicarse con otros contenedores en la misma red mediante el nombre del contenedor. Por defecto el direccionamiento es 172.17.0.0/16

### Host network (red de anfitrión)

El contenedor comparte la red del host. Es decir, el contenedor utiliza la misma interfaz de red y la misma dirección IP que el host. Puede ser útil para aplicaciones que necesitan una comunicación de alto rendimiento entre el contenedor y el host, pero también puede presentar desafíos de aislamiento y seguridad.

### None network (red de anfitrión)

El contenedor no está conectado a ninguna red de Docker. El contenedor solo puede comunicarse con el host o con redes externas a través de enrutamiento NAT.

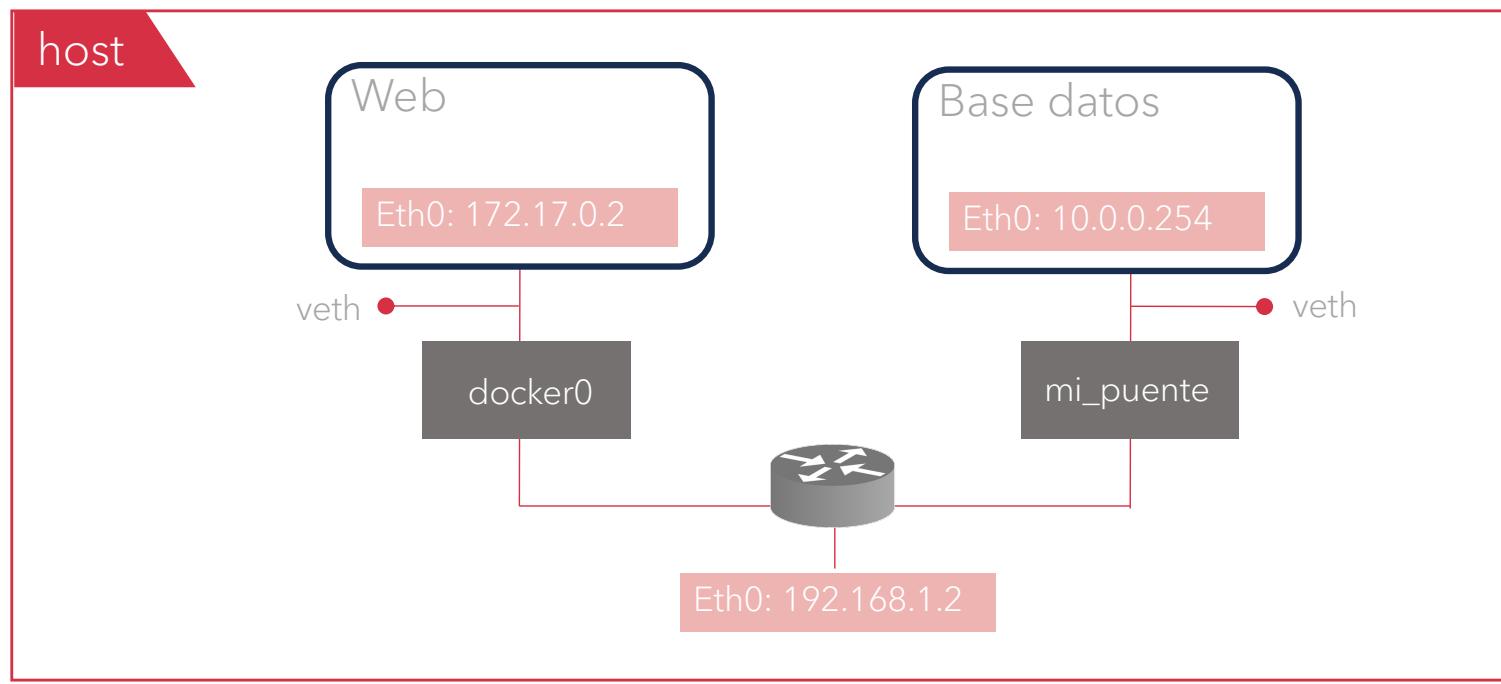




## Tipos de redes en Docker

### Custom Bridge Network (red puente personalizada)

Se pueden crear redes personalizadas para aislar grupos de contenedores y gestionar su comunicación de manera independiente. Esto es útil para controlar más precisamente la conectividad y la seguridad entre grupos de contenedores





# Redes. Funcionamiento básico



## Gestionando las redes en Docker

Listado de redes: `docker network ls`

```
$ docker network ls
NETWORK ID      NAME      DRIVER      SCOPE
c5f96d646d54    bridge    bridge      local
6bf6995ffba9    host      host       local
f4942c88036d    none     null       local
056d55045d57    red1     bridge      local
```

Borrado de redes: `docker network rm`

```
$ docker network rm red1
red1
```

*NOTA: No se puede borrar una red que haya contendores que la están usando.  
Primero hay que borrar el contenedor o desconectar la red.*





# Redes. Funcionamiento básico



## Gestionando las redes en Docker

Información de redes: `docker network inspect <nombre_red>`

```
$ docker network inspect bridge
[
    {
        "Name": "bridge",
        "Id": "c5f96d646d543b712a96473e2511bbfe4b79b9774604a793229e88e6af031f12",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": null,
            "Config": [
                {
                    "Subnet": "172.17.0.0/16",
                    "Gateway": "172.17.0.1"
                }
            ]
        },
        "Internal": false,
        ...
    }
]
```





## Gestionando las redes en Docker

Creación de redes: `docker network create OPCIIONES <nombre_red>`

### Options:

--attachable	Enable manual container attachment
--aux-address map	Auxiliary IPv4 or IPv6 addresses used by Network driver...
--config-from string	The network from which to copy the configuration
--config-only	Create a configuration only network
-d, --driver string	Driver to manage the Network (default "bridge")
--gateway strings	IPv4 or IPv6 Gateway for the master subnet
--ingress	Create swarm routing-mesh network
--internal	Restrict external access to the network
--ip-range strings	Allocate container ip from a sub-range
--ipam-driver string	IP Address Management Driver (default "default")
--ipam-opt map	Set IPAM driver specific options (default map{})
--ipv6	Enable IPv6 networking
--label list	Set metadata on a network
-o, --opt map	Set driver specific options (default map{})
--scope string	Control the network's scope
--subnet strings	Subnet in CIDR format that represents a network segment





## Gestionando las redes en Docker

Ejemplo. Crear una red por defecto (red puente)

```
$ docker network create red1  
2d5ae7940725985944ec84ce37752a0381d6c34e68fb6e5d4e89557144225175
```

```
$ docker network inspect red1  
[  
  {  
    "Name": "red1",  
    "Id": "2d5ae7940725985944ec84ce37752a0381d6c34e68fb6e5d4e89557144225175",  
    "Scope": "local",  
    "Driver": "bridge",  
    "EnableIPv6": false,  
    "IPAM": {  
      "Driver": "default",  
      "Options": {},  
      "Config": [  
        {  
          "Subnet": "172.19.0.0/16",  
          "Gateway": "172.19.0.1"  
        }  
      ]  
    },  
    "Internal": false,  
    ...  
  }]
```





# Redes. Funcionamiento básico



## Gestionando las redes en Docker

Ejemplo. Crear una red puente indicando dirección y pasarela

```
$ docker network create -d bridge --subnet 172.24.0.0/16 --gateway 172.24.0.1 red2  
89ecf967bf987ec2c6b1e976f36bf0ff38e260792a329b9b7b0665b7ad4ff27f
```

```
$ docker network inspect red2  
[  
  {  
    "Name": "red2",  
    "Id": "89ecf967bf987ec2c6b1e976f36bf0ff38e260792a329b9b7b0665b7ad4ff27f",  
    "Scope": "local",  
    "Driver": "bridge",  
    "EnableIPv6": false,  
    "IPAM": {  
      "Driver": "default",  
      "Options": {},  
      "Config": [  
        {  
          "Subnet": "172.24.0.0/16",  
          "Gateway": "172.24.0.1"  
        }  
      ]  
    },  
    "Internal": false,  
    ...  
  }]
```





# Redes. Funcionamiento básico



## Uso de redes definidas por el usuario

Ejemplo. Conectar dos contenedores a una misma red

```
$ docker network inspect red1
[
    {
        "Name": "red1",
        ...
        "Config": [
            {
                "Subnet": "172.19.0.0/16",
                "Gateway": "172.19.0.1"
            }
        ],
        ...
    },
    ...
]
```

```
$ docker network inspect red2
[
    {
        "Name": "red1",
        ...
        "Config": [
            {
                "Subnet": "172.24.0.0/16",
                "Gateway": "172.24.0.1"
            }
        ],
        ...
    },
    ...
]
```

Creamos un contenedor conectado a la red1 y vemos su IP

```
$ docker run -it --name contenedor1 --network red1 debian bash
root@eedf23ca7d70:/# apt update && apt install iproute2 && apt install iputils-ping
root@eedf23ca7d70:/# ip a
...
    inet 172.19.0.3/16 brd 172.19.255.255 scope global eth0
...
```





# Redes. Funcionamiento básico



## Uso de redes definidas por el usuario

Ejemplo. Conectar dos contenedores a una misma red

Creamos un contenedor conectado a la red2 y vemos su IP

```
$ docker run -it --name contenedor2 --network red2 debian bash  
root@bdc2b191b922:/# apt update && apt install iproute2 && apt install iputils-ping  
root@bdc2b191b922:/# ip a  
...  
    inet 172.24.0.2/16 brd 172.24.255.255 scope global eth0  
    ...
```

Comprobamos si contenedor2 tiene conectividad con contenedor1 y viceversa

```
$ docker run -it --name contenedor2 --network red2 debian bash  
root@bdc2b191b922:/# ping contenedor1  
ping: contenedor1: Name or service not known
```

```
$ docker run -it --name contenedor1 --network red1 debian bash  
root@eedf23ca7d70:/# ping contenedor2  
ping: contenedor2: Name or service not known
```





# Redes. Funcionamiento básico



## Uso de redes definidas por el usuario

### Ejemplo. Conectar dos contenedores a una misma red

Conectaremos un contenedor a una red (contenedor1 conectado a red2)

```
$ docker network connect red2 contenedor1
```

Arrancamos los contenedores: contenedor1 y contenedor2

```
$ docker container start contenedor1
```

```
$ docker container start contenedor2
```

Comprobamos si contenedor2 tiene conectividad con contenedor1

```
$ docker container attach contenedor1
root@eedf23ca7d70:/# ip a
...
28: eth0@if29: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
      inet 172.19.0.3/16 brd 172.19.255.255 scope global eth0
...
30: eth1@if31: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
      inet 172.24.0.3/16 brd 172.24.255.255 scope global eth1
...
root@eedf23ca7d70:/# ping contenedor2
PING contenedor2 (172.24.0.2) 56(84) bytes of data.
64 bytes from contenedor2.red2 (172.24.0.2): icmp_seq=1 ttl=64 time=0.356 ms
...
```





# Índice

**01**

## Introducción

- Desde inicio hasta actualidad
- Qué es Docker
- Conceptos básicos
- Instalación

**02**

## Cliente en Docker

- Hola mundo
- Operaciones
- Comandos básicos

**03**

## Datos en Docker

- Docker volume
- Docker bind mount

## Redes en Docker

- Tipos de redes
- Gestionando redes
- Uso de redes definidas por usuario

**04**

## Imágenes en Docker

- Gestionando imágenes
- Creación de imágenes propias. Dockerfile
- Distribución de imágenes

**05**

## Escenarios multi-contenedor

- Escenarios vs orquestación
- Docker compose. Comandos
- Fichero docker-compose.yml
- Almacenamiento en Docker compose
- Redes en Docker compose

**06**

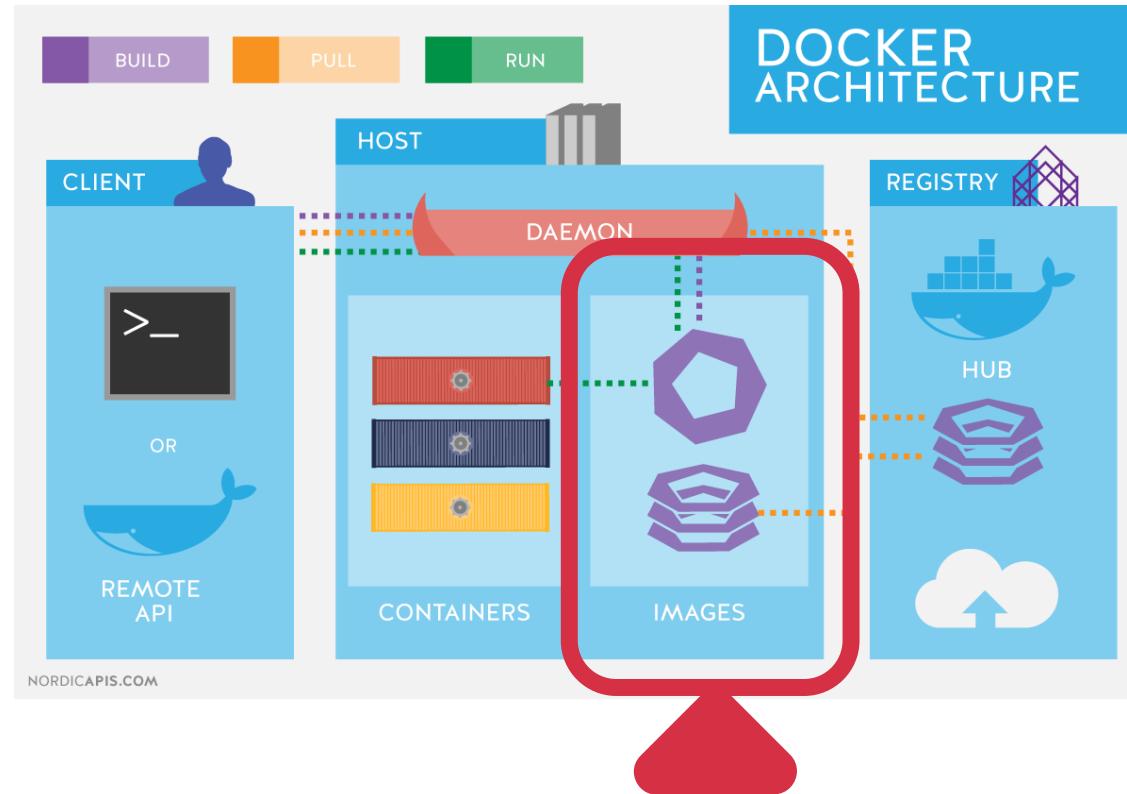




# Imágenes. Funcionamiento básico



Docker CE



Las imágenes se usan para ejecutar contenedores





## Gestión de Imágenes en Docker

Una imagen es una colección de archivos (una plantilla de solo lectura).

Buscar imágenes

- Registros oficiales: <https://hub.docker.com>
- Nuestro propio registro de imágenes

Descargar imágenes

```
$ docker pull [OPTIONS] NAME[:TAG|@DIGEST]
```

- Las imágenes se identifican tanto por nombre como por versión.
- Podemos tener distintas versiones de una misma imagen.

Listado de imágenes

```
$ docker images
```

Borrado de imágenes

```
$ docker rmi [OPTIONS] IMAGE [IMAGE...]
```





# Imágenes en Docker



## Gestión de Imágenes en Docker

Podemos crear una imagen a partir de otra imagen de un contenedor que ya tengamos arrancado y configurado.

```
$ docker commit [OPTIONS] CONTAINER [REPOSITORY[:TAG]]
```

Ejemplo. Crear un contenedor (milinux) con un Debian e instalar apache

```
$ docker container run -p 8080:80 -it --name milinux debian bash
```

```
root@2ad36fc7f311: apt update && apt install apache2 -y
root@2ad36fc7f311: echo "<h1>Creando imagen Docker a partir de un contenedor</h1>" >
/var/www/html/index.html
root@2ad36fc7f311: exit
```

Crear una nueva imagen partiendo de ese contenedor usando docker commit

```
$ docker commit milinux jmsoto/miapache2:v1
```





## Gestión de Imágenes en Docker

Ejemplo. Crear un contenedor (milinux) con un Debian e instalar apache

Ver imagen apache2 creada con docker commit

\$ docker images				
REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
jmsoto/miapache2	v1	c906e7082f38	3 minutes ago	284MB

Crear un nuevo contenedor a partir de esta nueva imagen, pero no podemos configurar el proceso que se va a ejecutar por defecto al crear el contenedor (el proceso por defecto que se ejecuta sería el de la imagen base).

En este caso para ejecutar el servidor web apache2 tendremos que ejecutar el comando **apache2ctl -D FOREGROUND**:

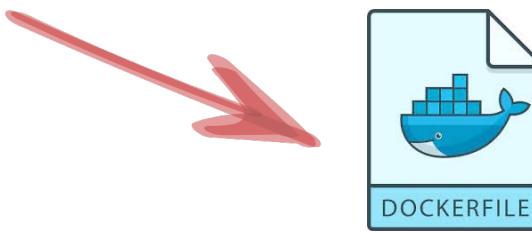
```
$ docker run -d -p 8081:80 --name miservidorweb jmsoto/miapache2:v1 bash  
-c "apache2ctl -D FOREGROUND"
```





### LIMITACIONES

- ▶ No se puede reproducir la imagen. Si la perdemos tenemos que recordar toda la secuencia de órdenes que habíamos ejecutado desde que arrancamos el contenedor hasta que teníamos una versión definitiva e hicimos docker commit.
- ▶ No podemos configurar el proceso que se ejecutará en el contenedor creado desde la imagen. Los contenedores creados a partir de la nueva imagen ejecutarán por defecto el proceso que estaba configurado en la imagen base.
- ▶ No podemos cambiar la imagen de base. Si ha habido alguna actualización, problemas de seguridad, etc. con la imagen de base tenemos que descargar la nueva versión, volver a crear un nuevo contenedor basado en ella y ejecutar de nuevo toda la secuencia de órdenes.





## Creación de imágenes propias: Dockerfile

Dockerfile es un archivo de texto con un conjunto de instrucciones que serán ejecutadas de forma secuencial para construir una nueva imagen Docker. Cada una de estas instrucciones crea una nueva capa en la imagen y especifican lo que va a ir en el entorno, dentro del contenedor (redes, volúmenes, puertos al exterior, archivos, etc).

Se puede reproducir fácilmente la imagen ya que en el fichero Dockerfile se definen todas y cada una de las órdenes necesarias para la construcción de la imagen



el contenedor se comporte de la misma forma en cualquier lugar que se ejecute





## Creación de imágenes propias: Dockerfile

### Estructura fundamental de un Dockerfile



- Imagen base:  
FROM
- Metadatos:  
LABEL
- Instrucciones de construcción:  
RUN, COPY, ADD, WORKDIR
- Configuración (variable de entornos, usuarios, puertos):  
USER, EXPOSE, ENV
- Instrucciones de arranque:  
CMD, ENTRYPOINT





## Creación de imágenes propias: Dockerfile

### Estructura fundamental de un Dockerfile

- **FROM:** Especifica la imagen base.
  - Ejemplo: `FROM php:7.4-apache`
- **LABEL:** Añade metadatos a la imagen mediante clave=valor.
  - Ejemplo: `LABEL company=ugr`
- **COPY:** Copia archivos desde el host a la imagen. Esos ficheros deben estar en el mismo contexto (carpeta o repositorio). Su sintaxis es `COPY [--chown=<usuario>:<grupo>] src dest`.
  - Ejemplo: `COPY --chown=www-data:www-data myapp /var/www/html`
- **ADD:** Similar a COPY pero con funcionalidades adicionales como especificar URLs y tratar archivos comprimidos.
- **RUN:** Ejecuta una orden creando una nueva capa. Su sintaxis es `RUN orden / RUN ["orden","param1","param2"]`.
  - Ejemplo: `RUN apt update && apt install -y git` (En este caso es muy importante que pongamos la opción -y porque en el proceso de construcción no puede haber interacción con el usuario).





## Creación de imágenes propias: Dockerfile

### Estructura fundamental de un Dockerfile

- **WORKDIR**: Establece el directorio de trabajo dentro de la imagen para posteriormente usar las órdenes RUN, COPY, ADD, CMD o ENTRYPOINT.
  - Ejemplo: `WORKDIR /usr/local/apache/htdocs`
- **EXPOSE**: Información acerca de los puertos del contenedor. Es meramente informativo.
  - Ejemplo: `EXPOSE 80`
- **USER**: Para especificar (por nombre o UID/GID) el usuario de trabajo para todas las órdenes RUN,CMD Y ENTRYPOINT posteriores.
  - Ejemplo: `USER jmsoto`
- **ARG**: Definir variables para poder especificar valores cuando se construya el contenedor mediante el flag --build-arg. Su sintaxis es  
`ARG nombre_variable` o `ARG nombre_variable=valor_por_defecto`.
  - Ejemplo: `ARG usuario=www-data`. No se puede usar con ENTRYPOINT Y CMD.





## Creación de imágenes propias: Dockerfile

### Estructura fundamental de un Dockerfile

- **ENV**: Establecer variables de entorno dentro del contenedor. Puede ser usado posteriormente en las órdenes RUN añadiendo \$ delante del nombre de la variable de entorno.

- Ejemplo: `ENV WEB_DOCUMENT_ROOT=/var/www/html`. No se puede usar con ENTRYPOINT Y CMD.

- **ENTRYPOINT**: Establecer el ejecutable que se lanza siempre cuando se crea el contenedor con docker run, salvo que se especifique expresamente algo diferente con el flag --entrypoint. Su sintaxis es:

- `ENTRYPOINT <command> / ENTRYPOINT ["executable", "param1", "param2"] .`

- Ejemplo: `ENTRYPOINT ["/usr/sbin/apache2ctl", "-D", "FOREGROUND"]`.

- **CMD**: Establecer el ejecutable por defecto (salvo que se sobreesciba desde la orden docker run) o para especificar parámetros para un ENTRYPOINT. Su sintaxis: `CMD param1 param2 / CMD ["param1", "param2"] / CMD["command", "param1"] .`

- Ejemplo: `CMD ["-c", "/etc/nginx.conf"] / ENTRYPOINT ["nginx"]`.

Descripción completa sobre el fichero Dockerfile: [documentación oficial](#).





## Creación de imágenes propias: Dockerfile

### Construyendo imágenes con docker build

El comando **docker build** construye la nueva imagen leyendo las instrucciones del fichero Dockerfile y la información de un entorno.

El **docker engine** crea la imagen a partir del Dockerfile y toda la información del entorno. Es recomendable guardar el Dockerfile en un directorio vacío y añadir los ficheros necesarios para la creación de la imagen.

El comando **docker build** ejecuta las instrucciones de un Dockerfile línea por línea y va mostrando los resultados. Cada instrucción ejecutada crea una imagen intermedia, una vez finalizada la construcción de la imagen se devuelve su id. Algunas imágenes intermedias se guardan en caché, otras se borran.

Si en algún momento falla la creación de la imagen, al modificar el Dockerfile y volver a construir la imagen, las instrucciones que no habían dado error anteriormente no se repiten ya que las imágenes intermedias correspondientes están cacheadas.





## Creación de imágenes propias: Dockerfile

Ejemplo. Crear una imagen con docker build y Dockerfile con un Debian e instalar apache

Crear un directorio "miservidorweb" (entorno) y un archivo index.html:

```
$ mkdir miservidorweb  
$ cd miservidorweb; echo "<h1>Creando imagen Docker a partir de un Dockerfile</h1>" > index.html
```

Crear un Dockerfile en el directorio "miservidorweb" con las siguientes instrucciones:

```
FROM debian:buster-slim  
MAINTAINER José Manuel Soto Hidalgo "jmsoto@ugr.es"  
RUN apt-get update && apt-get install -y apache2  
COPY index.html /var/www/html/  
CMD ["/usr/sbin/apache2ctl", "-D", "FOREGROUND"]
```

Crear la imagen indicando el nombre de la nueva imagen (opción -t) y el directorio contexto (en este caso el directorio miservidorweb (".")):

```
$ docker build -t jmsoto/miapache2:v2 .
```





## Creación de imágenes propias: Dockerfile

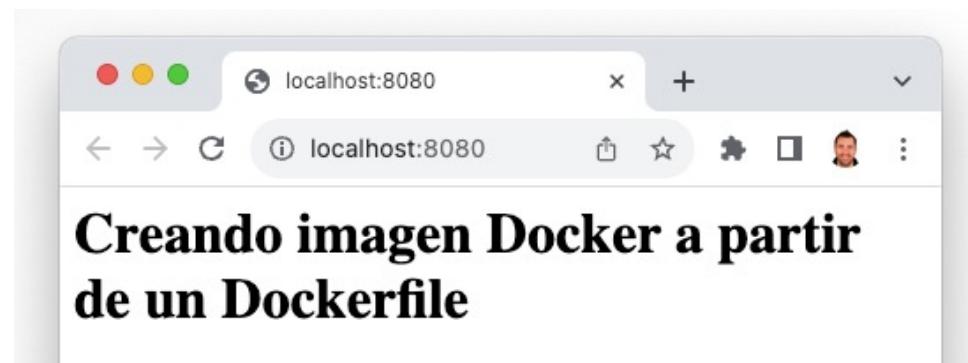
Ejemplo. Crear una imagen con docker build y Dockerfile con un Debian e instalar apache

Ver la imagen creada:

```
$ docker images
REPOSITORY          TAG      IMAGE ID      CREATED        SIZE
jmsoto/miapache2    v2      0cf76ebd1060   52 seconds ago  187MB
jmsoto/miapache2    v1      c906e7082f38   47 hours ago   284MB
```

Crear un contenedor con la imagen miapache:v2

```
$ docker run -d -p 8080:80 --name miservidorweb2 jmsoto/miapache2:v2
```





## Buenas prácticas con Dockerfile

- ▶ **1. Usa imágenes base oficiales:** Comienza siempre desde una imagen base oficial proporcionada por Docker o el proveedor de software. Estas imágenes suelen estar bien mantenidas y actualizadas regularmente.
- ▶ **2. Mantén los Dockerfiles simples:** Evita Dockerfiles complejos y largos. Divide las tareas en pasos claros y utiliza múltiples etapas si es necesario.
- ▶ **3. Minimiza capas:** Cada instrucción en un Dockerfile crea una nueva capa en la imagen. Trata de combinar instrucciones siempre que sea posible para reducir el número de capas, lo que disminuye el tamaño de la imagen.
- ▶ **4. Usa .dockerignore:** Crea un archivo ` `.dockerignore` para excluir archivos y directorios innecesarios del contexto de construcción. Esto acelerará la compilación de la imagen.
- ▶ **5. Especifica una etiqueta:** Utiliza la etiqueta ` `-t` para asignar un nombre y una etiqueta a la imagen que construyes facilitando la identificación de versiones.





## Buenas prácticas con Dockerfile

- ▶ **6. Evita contraseñas y credenciales en el Dockerfile:** No incluyas contraseñas, tokens u otros secretos en el Dockerfile. Utiliza variables de entorno para manejar esta información de manera segura.
- ▶ **7. Ordena las instrucciones lógicamente:** Coloca las instrucciones más estáticas y menos cambiantes al principio del Dockerfile. Esto permite aprovechar las cachés de capas en la construcción.
- ▶ **8. Usa COPY en lugar de ADD:** A menos que necesites URLs o desempaquetado automático, utiliza `COPY` en lugar de `ADD` para mayor claridad y control.
- ▶ **9. Limpieza en el mismo RUN:** Si necesitas instalar paquetes o descargar archivos, hazlo y límpialos en el mismo `RUN` para evitar la acumulación de archivos temporales.





## Buenas prácticas con Dockerfile

- ▶ **10. Minimiza el número de dependencias:** Reduce al mínimo las dependencias y los componentes innecesarios en tu imagen. Cuanto más ligera sea la imagen, mejor.
- ▶ **11. Utiliza capas de forma estratégica:** Agrupa instrucciones que cambiarán juntas y rara vez en la misma capa para reducir el tiempo de construcción.
- ▶ **12. Documenta tu Dockerfile:** Incluye comentarios explicativos en el Dockerfile para que otros puedan entender su propósito y funcionamiento.
- ▶ **13. Pruebas y revisión:** Realiza pruebas exhaustivas de tu Dockerfile para asegurarte de que funcione como se esperaba.
- ▶ **14. Automatiza la construcción y publicación:** Utiliza herramientas de automatización (como CI/CD) para construir y publicar imágenes automáticamente cuando se realicen cambios en el código fuente.





## Distribución de imágenes

- ▶ 1. Utilizar la secuencia de órdenes **docker commit / docker save / docker load**.

En este caso la distribución se producirá a partir de un archivo.

- ▶ 2. Utilizar la pareja de órdenes **docker commit / docker push**.

En este caso la distribución se producirá a través de DockerHub.

- ▶ 3. Utilizar la pareja de órdenes **docker export / docker import**.

En este caso la distribución de producirá a través de un archivo.





## Distribución de imágenes

### Distribución a partir de un archivo

1. Guardar esa imagen en un archivo .tar usando el comando **docker save**:

```
$ docker save jmsoto/miapache2:v2 > miapache2.tar
```

2. Distribuir el archivo .tar con la imagen

3. Cargar la imagen en el repositorio local a partir del archivo .tar

```
$ docker load -i miapache2.tar  
Loaded image: jmsoto/miapache2:v2
```





## Distribución de imágenes

### Distribución a través de Docker Hub

1. Autentificarse en Docker Hub usando el comando `docker login`

```
$ docker login
Log in with your Docker ID or email address to push and pull images from Docker Hub...
Username: jmsoto
Password:
Login Succeeded
```

2. Distribuir la imagen a DockerHub mediante `docker push`.

Nota: El nombre de la imagen tiene que tener como primera parte el nombre del usuario de DockerHub.

```
$ docker push jmsoto/miapache2:v2
The push refers to repository [docker.io/jmsoto/miapache2]
a1b44e501760: Pushed
f449b9954e55: Pushed
51a11a0fabdb: Mounted from library/debian
v2: digest: sha256:bf579f708877a1b90f060d0fc4b33b3b33ba8660ed0773887ee6aa8ab869056
size: 948
```

3. Cualquier usuario de DockerHub puede bajar la imagen mediante `docker pull`.





# Índice

01

## Introducción

- Desde inicio hasta actualidad
- Qué es Docker
- Conceptos básicos
- Instalación

02

## Cliente en Docker

- Hola mundo
- Operaciones
- Comandos básicos

03

## Datos en Docker

- Docker volume
- Docker bind mount

## Redes en Docker

- Tipos de redes
- Gestionando redes
- Uso de redes definidas por usuario

04

## Imágenes en Docker

- Gestionando imágenes
- Creación de imágenes propias. Dockerfile
- Distribución de imágenes

05

## Escenarios multi-contenedor

- Escenarios vs orquestación
- Docker compose. Comandos
- Fichero docker-compose.yml
- Almacenamiento en Docker compose
- Redes en Docker compose

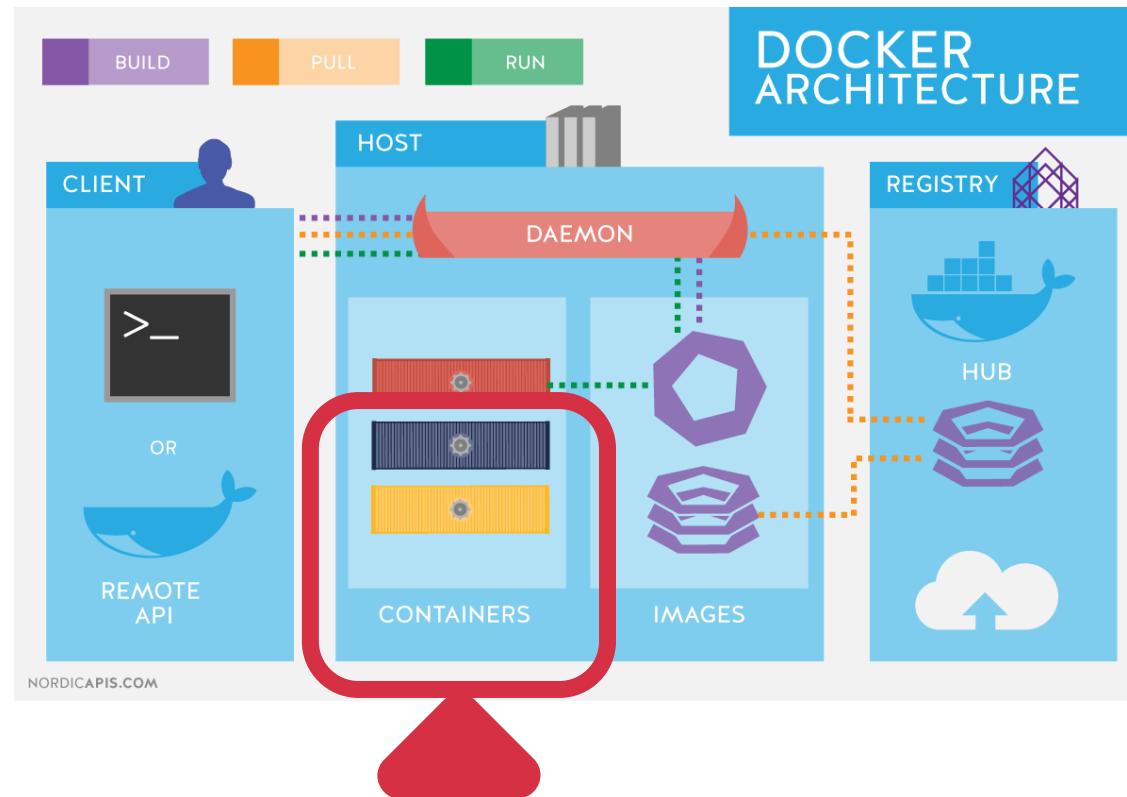
06



# Escenarios multi-contenedor



Docker CE



Generalmente, una aplicación requiere varios servicios → varios contenedores





# Escenarios multi-contenedor



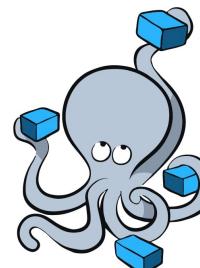
## Gestión de varios contenedores

▶ Generalmente, una aplicación requiere varios servicios → varios contenedores

EJEMPLO: Partiendo del principio de que cada contenedor ejecuta un sólo proceso, si necesitamos que la aplicación use varios servicios (web, base de datos, proxy inverso, ...) cada uno de ellos se implementará en un contenedor.



Definir el escenario en un fichero llamado `docker-compose.yaml` y gestionar el ciclo de vida de la aplicación y de todos los contenedores que necesitamos con la utilidad `docker-compose`.



**docker**  
Compose

¿Orquestación de contenedores?





# Escenarios multi-contenedor



## Escenarios multi-contenedor vs orquestación

La principal diferencia radica en el **propósito** y en el **contexto**

### Escenario multi-contenedor

- Definir y ejecutar aplicaciones multi-contenedor en un solo host.
- Principalmente para entornos de desarrollo y pruebas.
- No proporciona capacidades de escalabilidad y alta disponibilidad en sí mismo, ya que se enfoca en entornos locales o de desarrollo.

### Orquestación

- Administrar un clúster de nodos de Docker y coordinar la ejecución de contenedores en ese clúster.
- Administrar la escalabilidad y la alta disponibilidad de aplicaciones en entornos de producción.
- Gestión de aplicaciones a gran escala en múltiples nodos de Docker.
- Características de escalabilidad automática y alta disponibilidad para aplicaciones en un clúster de Docker.





## Características

- ▶ Hacer todo de manera **declarativa** para que no tenga que repetir todo el proceso cada vez que construyo el escenario.
- ▶ Poner en **funcionamiento todos los contenedores** que necesita mi aplicación de una sola vez y debidamente configurados.
- ▶ Garantizar que los contenedores se **arrancan en el orden adecuado**. Por ejemplo: mi aplicación no podrá funcionar debidamente hasta que no esté el servidor de bases de datos funcionando en marcha.
- ▶ Asegurarnos de que hay **comunicación** entre los contenedores que pertenecen a la aplicación.





## El fichero **docker-compose.yaml**

### Servicios

- Son los que conforman nuestra aplicación y, normalmente, cada uno despliega un contenedor, asociado a una imagen Docker. Dentro de cada servicio podemos definir las características de cada contenedor, como el nombre, la imagen, los puertos que expone, volúmenes y redes a las que se conecta, etc.

### Volúmenes

- El mecanismo que tienen los contenedores para persistir y compartir datos.

### Redes

- Comunicación entre los distintos contenedores. Definir una red en común, siguiendo el modelo Container Network Model.

### Variables entorno

- Definición de variables de entorno para definir valores que están disponibles para los procesos que se ejecutan dentro del contenedor.





## El fichero **docker-compose.yaml**

### Servicios

todas las propiedades son opcionales a excepción de `image`

- **[image]**: Nombre de la imagen Docker usada para desplegar el contenedor.
- **[container\_name]**: Nombre del contenedor a desplegar.
- **[build]**: Ruta al directorio donde se encuentra el Dockerfile para construir la imagen, si no está construida.
- **[command]**: Comandos que ejecutar al iniciar el contenedor.
- **[ports]**: Puertos que exponer al exterior del contenedor. Tienen el formato `puerto_host:puerto_contenedor`, donde, como host, podremos acceder al servicio desde `localhost:puerto_host`.
- **[volumes]**: Volúmenes que montar en el contenedor.
- **[environment]**: Variables de entorno que definir en el contenedor. Es recomendable configurar los contenedores con variables de entorno y no valores fijos, para poder cambiar la configuración de los servicios sin tener que reconstruir las imágenes.
- **[depends\_on]**: Servicios que deben desplegarse antes que este.
- **[networks]**: Redes a las que pertenece el contenedor.
- **[restart]**: Política de reinicio del contenedor. Puede tomar los valores `no`, `always`, `on-failure`, `unless-stopped`, por defecto `no`.





## El fichero **docker-compose.yaml**

### Volúmenes      todas las propiedades son opcionales

- **[driver]**: Driver del volumen. Por defecto, local.
- **[driver\_opts]**: Opciones del driver del volumen.
  - **type**: Tipo de volumen. Puede ser `none`, `bind`, `volume` o `tmpfs`.
  - **device**: Ruta al directorio del host que se va a montar.
  - **o**: Opciones de montaje del volumen. Puede tomar valores `bind`, `private`, `ro`, `rw` y `shared`, entre otros.
- **[external]**: Indica si el volumen es externo o no. Por defecto, false.
- **[labels]**: Etiquetas del volumen.
- **[name]**: Nombre del volumen.
- **[scope]**: Alcance del volumen. Por defecto, local.





## El fichero **docker-compose.yaml**

### Redes

todas las propiedades son obligatorias

- **SandBox:** Aísla el contenedor del resto del sistema, limitando el acceso a esta red al tráfico que llega por los **endpoints**.
- **Endpoints:** Puntos de comunicación entre las redes aisladas de los contenedores y la **network** que los conecta con el resto del sistema.
- **Network:** Red que comunica las **sandbox** de los diferentes contenedores por medio de los **endpoints**.

Nombre	Alcance	Descripción
<b>bridge</b>	Local	Red por defecto de Docker. Crea una red virtual en el host, que conecta los contenedores por medio de un bridge virtual.
<b>host</b>	Local	Deshabilita el aislamiento entre los contenedores y el host, no hace falta exponer puertos
<b>overlay</b>	Global	Permite conectar múltiples demonios Docker entre sí y habilitar la comunicación entre servicios distribuidos.
<b>ipvlan</b>	Global	El usuario obtiene el control total del direccionamiento IPv4 e IPv6 de la red.
<b>macvlan</b>	Global	Permite asignarle una dirección MAC a un contenedor, haciendo que aparezca como un dispositivo físico en la red. El tráfico se dirige por MAC.
<b>none</b>	Local	Deshabilita toda la gestión de red, normalmente usado en conjunto con un driver de red propio.





# Docker compose



## El fichero **docker-compose.yaml**

### Variables entorno

```
version: '3.9'  
services:  
  web:  
    build: .  
    environment:  
      - DEBUG_MODE=true
```

```
version: '3.9'  
services:  
  web:  
    image: "webapp:${TAG}"
```

- Variables en archivos: un [environment file](#) denominado **.env** donde se definen las variables

docker-compose.yml

```
version: '3.9'  
services:  
  web:  
    image: "webapp:${TAG}"
```

.env

```
TAG=v1.5
```



El archivo **.env** se carga por defecto si está en la misma ruta que el **yml**:

```
$ docker-compose up  
  
$ docker-compose config  
version: '3.9'  
services:  
  web:  
    image: "webapp:v1.5"
```





# Docker compose



## El fichero **docker-compose.yaml**

### Variables entorno

```
version: '3.9'  
services:  
  web:  
    build: .  
    environment:  
      - DEBUG_MODE=true
```

```
version: '3.9'  
services:  
  web:  
    image: "webapp:${TAG}"
```

- Variables en archivos: un [environment file](#) denominado **.env** donde se definen las variables

docker-compose.yml

```
version: '3.9'  
services:  
  web:  
    image: "webapp:${TAG}"
```

.env

TAG=v1.5

./config/.env

TAG=v1.6

El argumento **--env-file** anula la ruta de archivo predeterminada

```
$ docker-compose up --env-file ./config/.env  
$ docker-compose config  
version: '3.9'  
services:  
  web:  
    image: "webapp:v1.6"
```





## Comandos docker-compose

- **docker-compose up**: Crear los contenedores (servicios) que están descritos en el docker-compose.yml.
- **docker-compose up -d**: Crear en modo detach los contenedores (servicios) que están descritos en el docker-compose.yml. Eso significa que no muestran mensajes de log en el terminal y que se nos vuelve a mostrar un prompt.
- **docker-compose stop**: Detiene los contenedores que previamente se han lanzado con docker-compose up.
- **docker-compose run**: Inicia los contenedores descritos en el docker-compose.yml que estén parados.
- **docker-compose rm**: Borra los contenedores parados del escenario. Con la opción -f elimina también los contenedores en ejecución.
- **docker-compose pause**: Pausa los contenedores que previamente se han lanzado con docker-compose up.
- **docker-compose unpause**: Reanuda los contenedores que previamente se han pausado.
- **docker-compose restart**: Reinicia los contenedores. Orden ideal para reiniciar servicios con nuevas configuraciones.





## Comandos docker-compose

- **docker-compose down**: Para los contenedores, los borra y también borra las redes que se han creado con docker-compose up (en caso de haberse creado).
- **docker-compose down -v**: Para los contenedores y borra contenedores, redes y volúmenes.
- **docker-compose logs**: Muestra los logs de todos los servicios del escenario. Con el parámetro -f podremos ir viendo los logs en "vivo".
- **docker-compose logs servicio1**: Muestra los logs del servicio llamado servicio1 que estaba descrito en el docker-compose.yml.
- **docker-compose exec servicio1 /bin/bash**: Ejecuta una orden, en este caso /bin/bash en un contenedor llamado servicio1 que estaba descrito en el docker-compose.yml
- **docker-compose build**: Ejecuta, si está indicado, el proceso de construcción de una imagen que va a ser usado en el docker-compose.yml a partir de los ficheros Dockerfile que se indican.
- **docker-compose top**: Muestra los procesos que están ejecutándose en cada uno de los contenedores de los servicios.





## Almacenamiento en docker-compose. Volúmenes

Ejemplo: Definir un contenedor con base datos mariadb y un volumen

1. Definimos el archivo Docker-compose.yml

```
version: '3.1'
services:
  db:
    container_name: contenedor_mariadb
    image: mariadb
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: mipasswd
    volumes:
      - mariadb_data:/var/lib/mysql
volumes:
  mariadb_data:
```



2. Iniciamos el escenario

```
$ docker compose up -d
[+] Building 0.0s (0/0)
[+] Running 3/3
  ✓ Network almacenamiento_dockercompose_default      Created
  ✓ Volume "almacenamiento_dockercompose_mariadb_data" Created
  ✓ Container contenedor_mariadb                      Started
```

docker:desktop-linux  
0.0s  
0.0s  
0.0s





## Almacenamiento en docker-compose. Volúmenes

Ejemplo: Definir un contenedor con base datos mariadb y un volumen

3. Comprobamos que se ha creado el contenedor y el volumen

```
$ docker compose ps
NAME          IMAGE        COMMAND
contenedor_mariadb    mariadb    "docker-entrypoint.sh mariadb"
SERVICE      STATUS      PORTS
db           Up 2 minutes  3306/tcp

$ docker volume ls
DRIVER      VOLUME NAME
local       almacenamiento_dockercompose_mariadb_data

$ docker inspect contenedor_mariadb
...
"Mounts": [
  {
    "Type": "volume",
    "Name": "almacenamiento_dockercompose_mariadb_data",
    "Source": "/var/lib/docker/volumes/almacenamiento_dockercompose_mariadb_data/_data",
    "Destination": "/var/lib/mysql",
    "Driver": "local",
    "Mode": "z",
    "RW": true,
    "Propagation": ""
  }
],
...

```

Para iniciar el escenario desde 0, debes eliminar el volumen:  
**\$ docker compose down -v**





# Docker compose

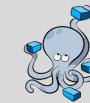


## Almacenamiento en docker-compose. Bind mount

Ejemplo: Definir un contenedor con base datos mariadb y montar un directorio

1. Definimos el archivo Docker-compose.yml

```
version: '3.1'
services:
  db:
    container_name: contenedor_mariadb
    image: mariadb
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: mipasswd
    volumes:
      - ./data:/var/lib/mysql
```



2. Iniciamos el escenario y comprobamos que se ha creado la carpeta **data** con los datos

```
$ docker compose up -d
[+] Building 0.0s (0/0)
[+] Running 1/1
  ✓ Container contenedor_mariadb Started
                                            docker:desktop-linux
                                            0.0s

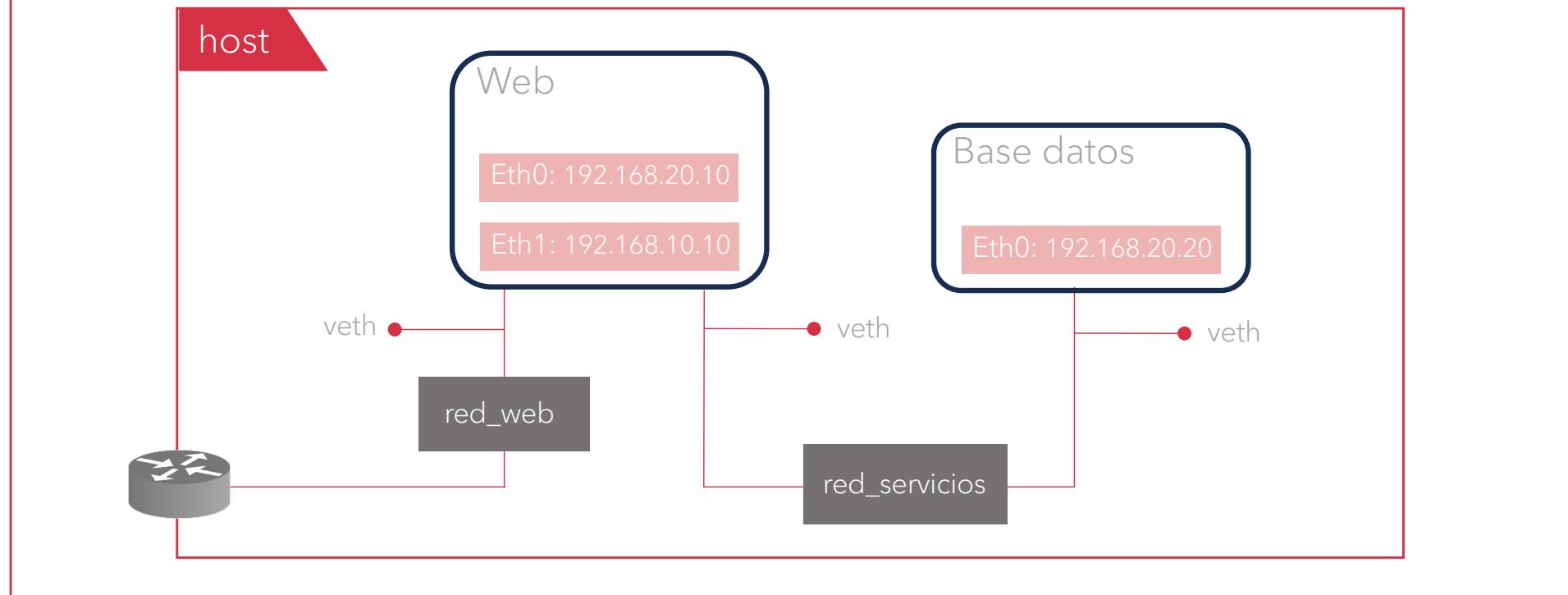
$ ls data
aria_log.00000001 ib_buffer_pool ibtmp1 mysql undo001
aria_log_control ib_logfile0 mariadb_upgrade_info performance_schema undo002
ddl_recovery.log ibdata1 multi-master.info sys undo003
```





## Redes en docker-compose. Networks

Ejemplo: Definir un escenario de aplicación web con dos contenedores, uno con un servidor web y otro con una base de datos con dos redes: una red interna con conexión del servidor web a la base datos y otra red web para salir al exterior.





## Redes en docker-compose. Networks

1. Definimos el archivo Docker-compose.yml

```
version: '3.1'
services:
  app:
    container_name: servidor_web
    image: httpd:2.4
    restart: always
    ports:
      - 8080:80
    networks:
      red_web:
        ipv4_address: 192.168.10.10
      red_servicios:
        ipv4_address: 192.168.20.10
    hostname: servidor_web

  db:
    container_name: servidor_mariadb
    image: mariadb
    environment:
      MYSQL_ROOT_PASSWORD: mipasswd
    restart: always
    networks:
      red_servicios:
        ipv4_address: 192.168.20.20
    hostname: servidor_mariadb
```

```
networks:
  red_web:
    ipam:
      config:
        - subnet: 192.168.10.0/24
  red_servicios:
    ipam:
      config:
        - subnet: 192.168.20.0/24
```





# Docker compose



## Redes en docker-compose. Networks

### 2. Iniciamos el escenario

```
$ docker compose up -d
[+] Building 0.0s (0/0)
[+] Running 4/4
  ✓ Network contenedores_red_web      Created
  ✓ Network contenedores_red_servicios Created
  ✓ Container servidor_mariadb        Started
  ✓ Container servidor_web            Started
                                                     docker:desktop-linux
                                                     0.0s
                                                     0.0s
                                                     0.0s
                                                     0.0s
```

### 3. Comprobamos que los dos contenedores se están ejecutando

```
$ docker compose ps
NAME          IMAGE       COMMAND
servidor_mariadb  mariadb    "docker-entrypoint.sh mariadb"
servidor_web     httpd:2.4   "httpd-foreground"
minutes         0.0.0.0:8080->80/tcp
                                         SERVICE  STATUS      PORTS
                                         db        Up 6 minutes  3306/tcp
                                         app        Up 6
```

### 4. Accedemos al servidor web e instalamos los paquetes necesarios para hacer las comprobaciones de configuración de la red

```
$ docker-compose exec app bash
root@servidor_web:/usr/local/apache2# apt-get update && apt-get install -y inetutils-ping \
  iproute2 \
  dnsutils
```





## Redes en docker-compose. Networks

5. Comprobamos que el contenedor servidor\_web está conectado a las dos redes y tiene las direcciones que hemos indicado:

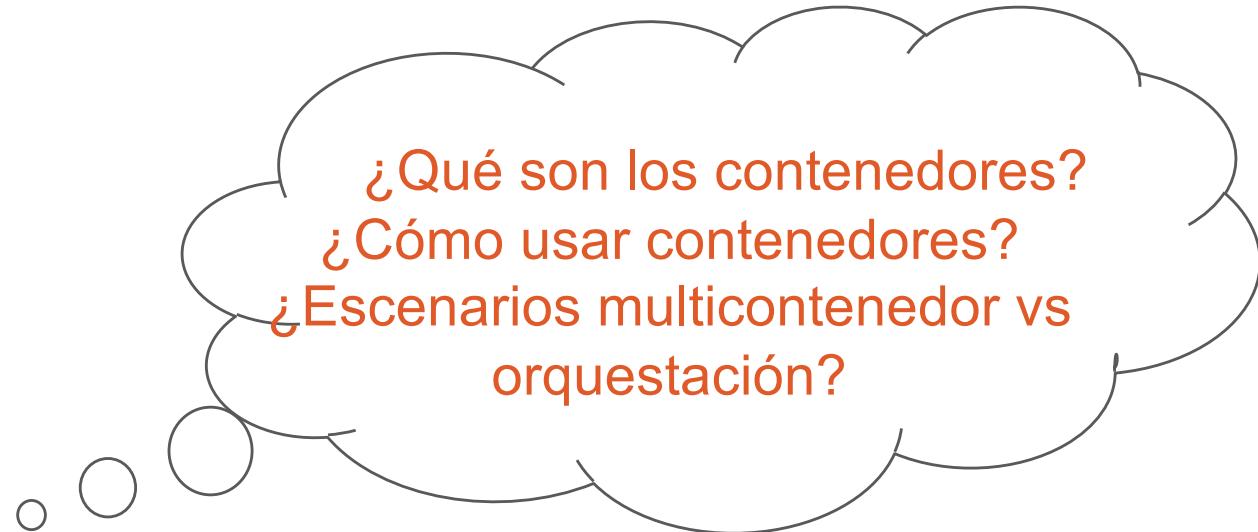
```
root@servidor_web:/usr/local/apache2# ip a
...
9: eth1@if10: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:c0:a8:0a:0a brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 192.168.10.10/24 brd 192.168.10.255 scope global eth1
        valid_lft forever preferred_lft forever
13: eth0@if14: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:c0:a8:14:0a brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 192.168.20.10/24 brd 192.168.20.255 scope global eth0
        valid_lft forever preferred_lft forever
```

6. Comprobamos que el contendor servidor\_web tiene conexión con el servidor de bases de datos

```
root@servidor_web:/usr/local/apache2# ping servidor_mariadb
PING servidor_mariadb (192.168.20.10): 56 data bytes
64 bytes from 192.168.20.10: icmp_seq=0 ttl=64 time=0.168 ms
64 bytes from 192.168.20.10: icmp_seq=1 ttl=64 time=0.331 ms
```



# Servidores Web de Altas Prestaciones



¿Qué son los contenedores?  
¿Cómo usar contenedores?  
¿Escenarios multicontenedor vs orquestación?

## Práctica 0: Contenerización