

## Introducción a Docker

### 1. ¿Cuál es el problema que podemos tener?

Problems arise when the supporting software environment is not identical, says Docker creator Solomon Hykes. "You're going to test using Python 2.7, and then it's going to run on Python 3 in production and something weird will happen. Or you'll rely on the behavior of a certain version of an SSL library and another one will be installed. You'll run your tests on Debian and production is on Red Hat and all sorts of weird things happen."














### 2. ¿Cuáles son los motivos?

<i>Múltiples stacks</i>	<i>Múltiples targets</i>
Lenguajes	Entornos de Desarrollo Locales
Versiones	Preproducción
Frameworks	Control de Calidad (QA)
Librerías	Puesta en escena (staging)
Bases de Datos	Producción: hierro, nube, híbrido.

### 3. El problema que intentamos resolver:

Idealmente nos gustaría tener un método que replicase un completo y complejo sistema con una variedad de plataformas hardware y entornos con la posibilidad de actualizar componentes en lugar del sistema completo para cada cambio.

**Docker** llama a esto “la matriz del infierno”:

	Static website	?	?	?	?	?	?	?
	Web frontend	?	?	?	?	?	?	?
	Background workers	?	?	?	?	?	?	?
	User DB	?	?	?	?	?	?	?
	Analytics DB	?	?	?	?	?	?	?
	Queue	?	?	?	?	?	?	?
		Development VM	QA Server	Single Prod Server	Onsite Cluster	Public Cloud	Contributor's laptop	Customer Servers
								

Necesitamos una manera de entregar actualizaciones desde los sistemas de desarrollo a cualquier número de entornos de misión crítica.

Una analogía para la entrega o despliegue de software podría ser un proveedor de bienes que envía un producto físico a un cliente. Imaginate que vendes un producto físico y tenés clientes en tu mismo continente así como fuera de él. Los requisitos para la entrega de dicho producto depende del tamaño, embalaje y sistema de transporte. En algunos casos, estos requisitos puede variar de un país a otro. Así que para transportar bienes tenemos una “*matriz del infierno*” parecida a la que tenemos en nuestros centros de datos:

	?	?	?	?	?	?	?
	?	?	?	?	?	?	?
	?	?	?	?	?	?	?
	?	?	?	?	?	?	?
	?	?	?	?	?	?	?
	?	?	?	?	?	?	?
							

Necesitas saber cómo va a ser transportado tu producto en cada etapa, la ruta que va a seguir, asegurando que el embalaje será adecuado para la entrega de tus bienes en el destino final. Antes de los 60, esta era la situación hasta que el contenedor fue inventado.

Cada contenedor es diseñado como una unidad estandarizada que permite que los bienes fuesen cargados en una caja que se sellaba y permanecía sellada hasta su entrega. Mientras tanto, podía ser cargada y descargada, apilada y movida con eficiencia a lo largo de grandes distancias y transferida de un método de transporte a otro.

## 4. ¿Qué es Docker?

**Docker** es un proyecto de código abierto que automatiza el *Deployment* de aplicaciones dentro de **containers** de software, proporcionando una capa adicional de abstracción y automatización de virtualización de aplicaciones en múltiples sistemas operativos.

**Docker** es un “emulador” de entornos aislado para poder ejecutar programas sin que afecte al sistema operativo (SO) y pudiéndose llevar y replicar en otros sistemas operativos o entornos. Parecido a *VirtualBox* o *VMWare*, pero mucho más ligero y a nivel de sistema operativo. Básicamente no vamos a tener más de un sistema

**Docker** consta de imágenes y **containers**:

- Una imagen es la especificación inerte, inmutable, una foto del estado y de unas piezas de software que incluyen desde la aplicación que queremos ejecutar hasta las librerías y todo lo necesario para que corra encima del sistema operativo en el cual se ejecuta.
- Un contenedor es un entorno aislado con la instanciación de una imagen, el cual se puede configurar.

Una analogía sería que la imagen es la clase y el contenedor el objeto de la clase.

Para una detallada explicación de los conceptos relevantes de Docker, recomendamos la lectura de los siguientes links:

1. [A Beginner-Friendly Introduction to Containers, VMs and Docker](#)
2. [What are containers and why do you need them?](#)
3. [Comandos de Docker y Docker Compose](#)
4. [Orchestration - Cluster Management - Kubernetes - Docker Swarm](#)

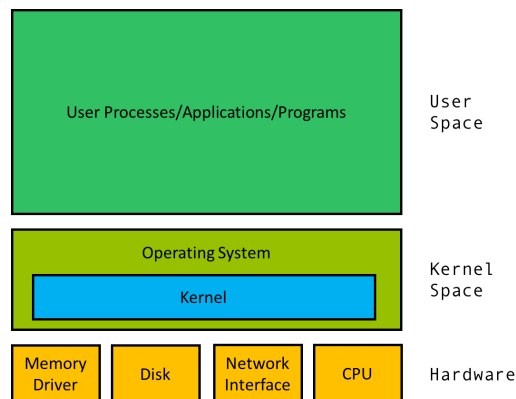
## 5. Resumen

### 5.1. ¿Cómo nos cambia el paradigma?

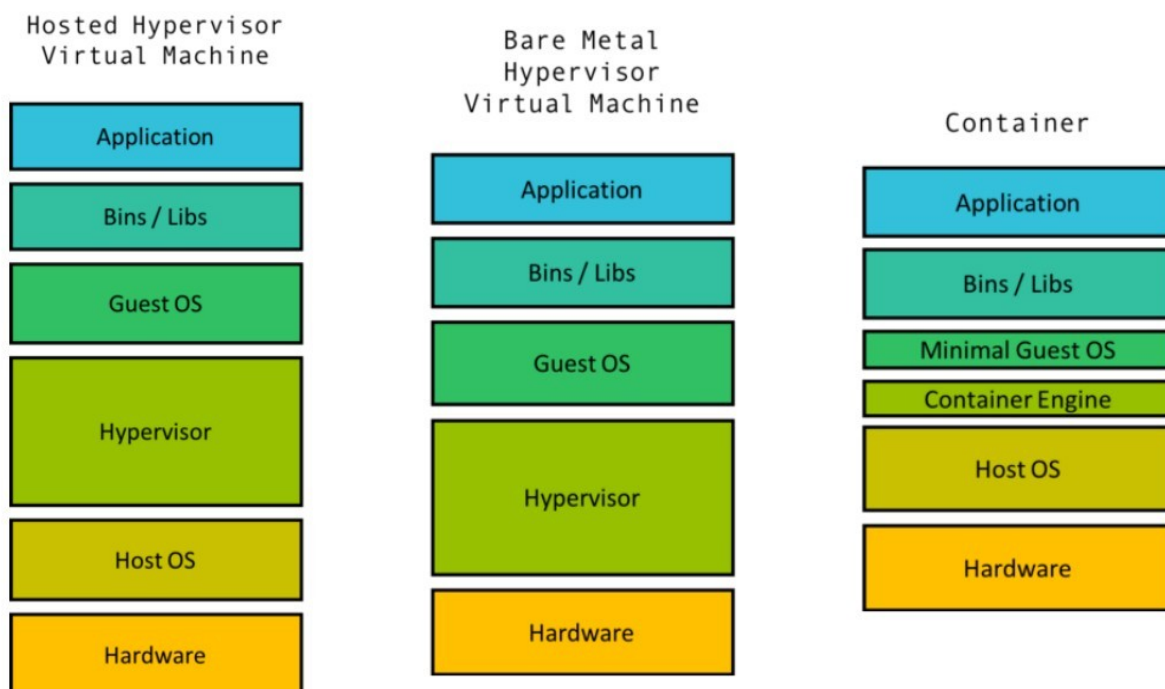
<i>Antes</i>	<i>Ahora</i>
Aplicaciones monolíticas	Servicios desacoplados
Largos ciclos de desarrollo	Mejoras rápidas e iterativas
Escalado planificado	Escalado dinámico
Único entorno/servidor	Múltiples entornos/servidores

### 5.2. Virtualización

¿Cómo se ven gráficamente el Kernel, el Sistema Operativo y el Userspace?



¿Cómo comparamos las VMs y los Containers?



**Virtual Machines:**

- Se basa en procesos que emulan un sistema operativo completo (generalmente).
- Es útil para correr casi todos los sistemas operativos *guest*.
- Tiene penalidad de alta performance.

**Bare Metal Hypervisor:**

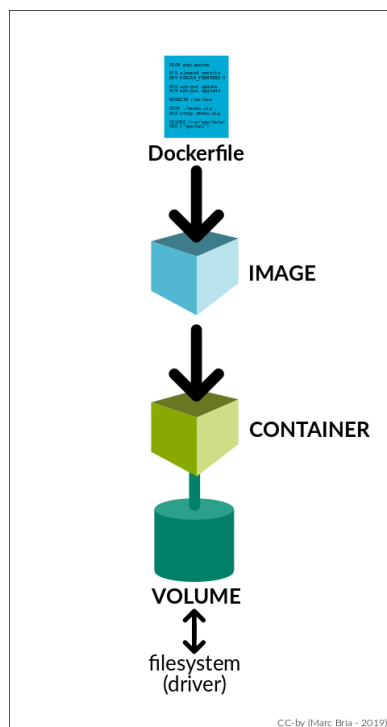
- El software corre sin un sistema operativo de base.
- Penalidad de más baja performance que la virtualización completa del Sistema Operativo.

**Containers**

- Se basa en Sistemas Operativos para el manejo de memoria, el *scheduling* de procesos y otros servicios.
- El *process isolation* se hace via las funcionalidades de Sistema Operativos *host*.
- Los procesos que corren dentro de un container no tienen casi penalidad de performance.
- El software corre sin un Sistema Operativo de base.
- Puede correr múltiples Sistemas Operativos *guest* con una penalidad de performance más baja que las *virtual machines* estándares.

## 5.3. Docker

### 5.3.1. ¿Cuáles son los conceptos básicos?

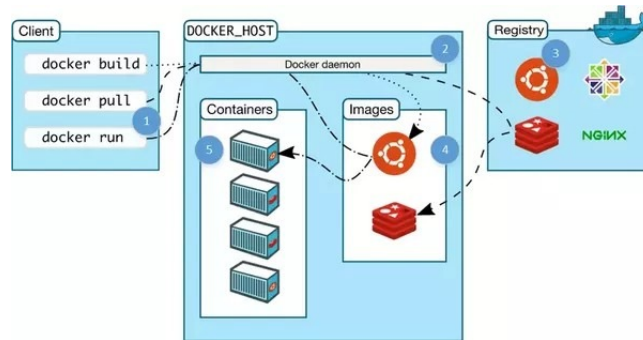


- **Dockerfile**: Las instrucciones para construir la imagen. ["La receta"]
- **Imagen**: El modelo abstracto del container. ["El molde"]
- **Container**: La concreción de una imagen abstracta. ["La galleta"]
- **Volumen**: Los datos, que pueden ser persistentes o volátiles. ["¿El plato?"]

### 5.3.2. Los Containers de Docker

- El *container* sirve como una unidad auto-aislada que pueden correr en cualquier lugar que lo soporta.
- El *container* también actúa como una unidad estandarizada de trabajo o cómputo.

- Hay herramientas de *orchestration* (lo veremos más adelante) que manejan el *deployment* y la ejecución del container en máquinas múltiples.



### 5.3.3. ¿Qué puedo hacer con Docker

- Simplifica la configuración, el *deployment* y el testeo.
- Hace más fácil reproducir el ambiente de producción en todos lados.
- Provee *application isolation*.
- Trata los ambientes como si fueran código regular (reusable, versionable, etc).

### 5.3.4. Elementos de un Dockerfile

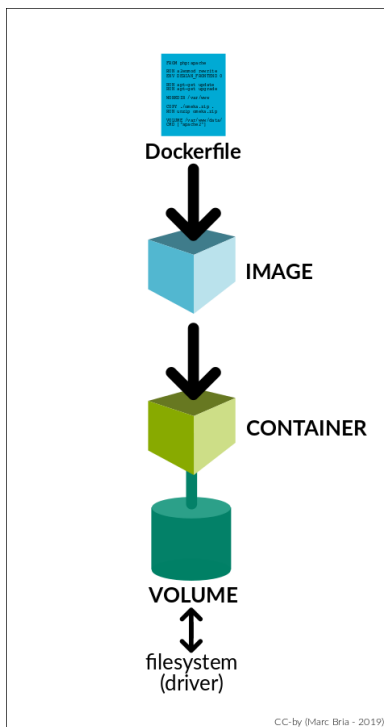
Para la definición, usaremos:

- **FROM**: define la IMAGEN BASE (nuestro punto de partida).
- **ENV**: define variables de entorno.
- **EXPOSE**: abre puertos del container.
- **WORKDIR**: directorio de trabajo.
- **USER**: usuario y grupo.

Las acciones:

- **COPY**: copia dentro del container (ver ADD).
- **RUN**: ejecuta un comando.
- **VOLUME**: crea un volumen.
- **CMD**: define el programa que ejecutará el container (ver ENTRYPOINT).

### 5.3.5. ¿Cuál es el circuito de Docker?



- **Dockerfile:** El Dockerfile se construye (**build**) para crear una "imagen"

```
$ docker build [options]
```

- **Imagen:** La "imagen" se ejecuta (**run**) para crear un *container*

```
$ docker run [options] image-name
```

- **Container:** En la definición (Dockerfile) or al iniciarlos, los "containers" incluyen o se asocian a "volúmenes"

```
$ docker run [options] -v .web:/var/www/html img-n
```

### 5.3.6. Comandos Básicos

- **build:** CONSTRUIR una imagen a partir del Dockerfile del directorio actual.

```
$ docker build
```

- **run:** CREAR un container.

```
$ docker run [options] [containerIDoName]
```

- **ps:** VER los containers en funcionamiento (con "-a" para verlos todos, incluso los detenidos)

```
$ docker ps
```

```
$ docker ps -a
```

- **stop:** DETENER un container

```
$ docker stop [containerIDoName]
```

- **start:** LEVANTAR un container

```
$ docker start [containerIDoName]
```

- **logs:** Ver los últimos REGISTROS.

```
$ docker logs [containerIDoName]
```

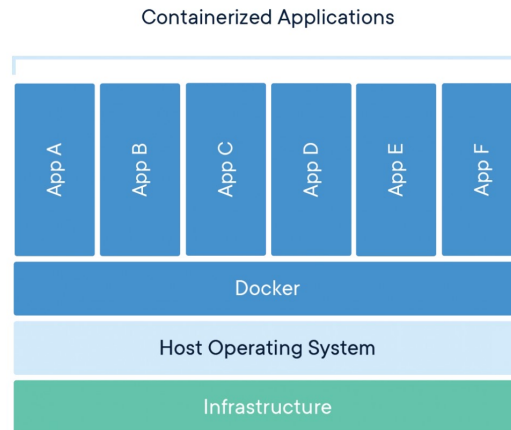
- **rm:** BORRA un container detenido.

```
$ docker rm [containerIDoName]
```

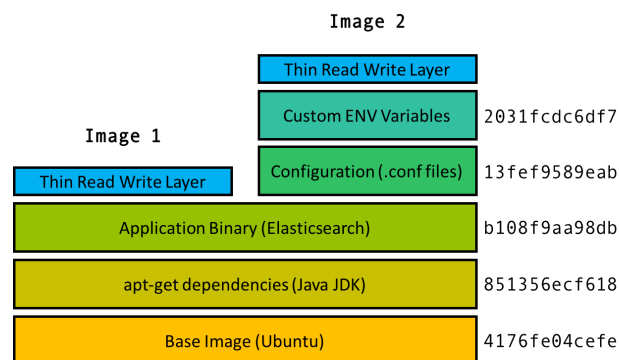
### 5.3.7. Imágenes de Docker

- Una imagen de Docker es un archivo, que contiene librerías de sistema, herramientas, y otros archivos y dependencias para la aplicación que va a correr.
- Un *container* es una instancia *corriendo* de una imagen.

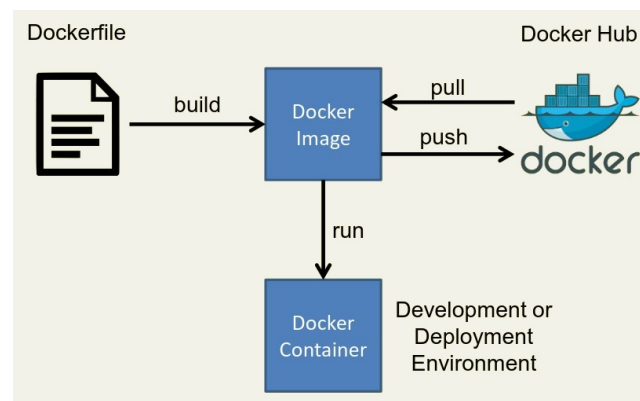
- Las imágenes se construyen agregando capas sobre imágenes previas.
- Cada imagen y capa se construyen siguiendo las instrucciones en un **Dockerfile**.



- Las imágenes diferentes pueden compartir las capas.
- La imagen base contiene el Sistema Operativo de un *Container*, que puede ser diferente del Sistema Operativo del *host*.



- Las imágenes puede ser *pusheadas* y *pulleadas* a/desde repositorios públicos o privados para un *deployment* fácil.

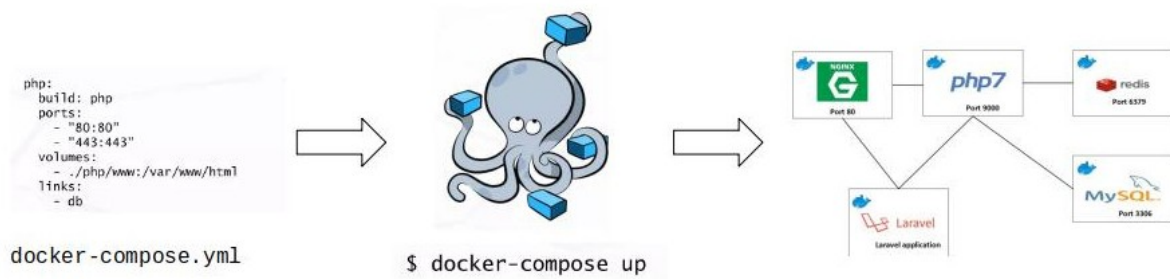


## 5.4. Orchestration

Las herramientas de *orchestration* que manejan el ciclo de vida de los servicios ejecutándose en *containers*:

- Proveen máquinas que ejecutan el container.
- Ejecutan y detienen grupos de *containers* como una aplicación simple.
- Proveen redundancia y disponibilidad.
- Balancean la carga.
- Escalan el número de *containers* que combinan la carga actual.
- *Alocan* recursos entre *containers*.
- Manejan la exposición de servicios.

El **Docker Compose** es una de las primeras herramientas de *orchestration*. Se usa para definir y ejecutar aplicaciones *multi-container* como si fuera código.



## 5.5. Cluster Management

Adicionalmente a la *orchestration*, algunas herramientas proveen un nivel extra de funcionalidad permitiendo la creación de *clusters* de máquinas múltiples:

- **Docker Swarm**
- Kubernetes

### 5.5.1. Docker Swarm



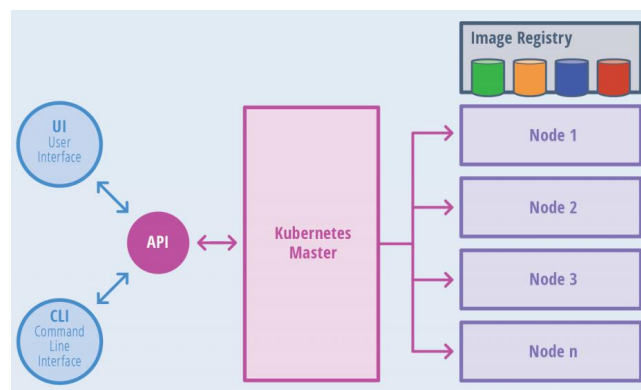
- Es una *API* compatible con todas las herramientas de Docker estándares, por ejemplo, *docker-compose*.
- Es fácil de instalar y manejar. Es bueno para unos pocos nodos de containers.
- Tiene características limitadas.
- Tiene una comunidad *open source* más pequeña que la de *Kubernetes*.



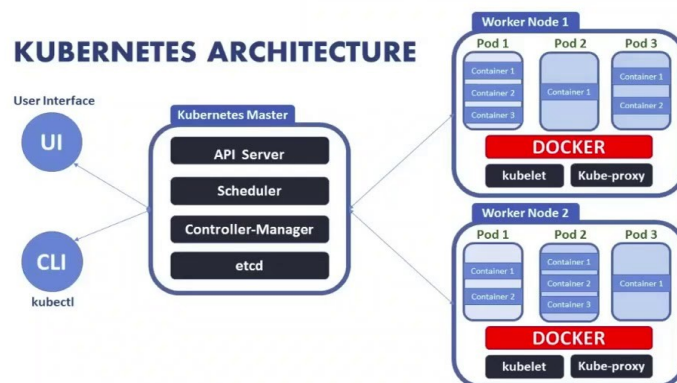
## 5.5.2. Kubernetes



- *Kubernetes* fue desarrollado por Google basado en 15 años de experiencia trabajando con *containers*.
- Tienen muchas características.
- Diseñado para ser tolerante a fallas.
- Es más difícil de instalar/configurar.



### Architecture



## 5.6. Cloud

En la actualidad hay proveedores de *cloud* que ofrecen soluciones para ejecutar *containers* de **Docker**.

- *Google Cloud: Kubernetes Engine*
- *AWS: ECS, EKS, Fargate, etc.*