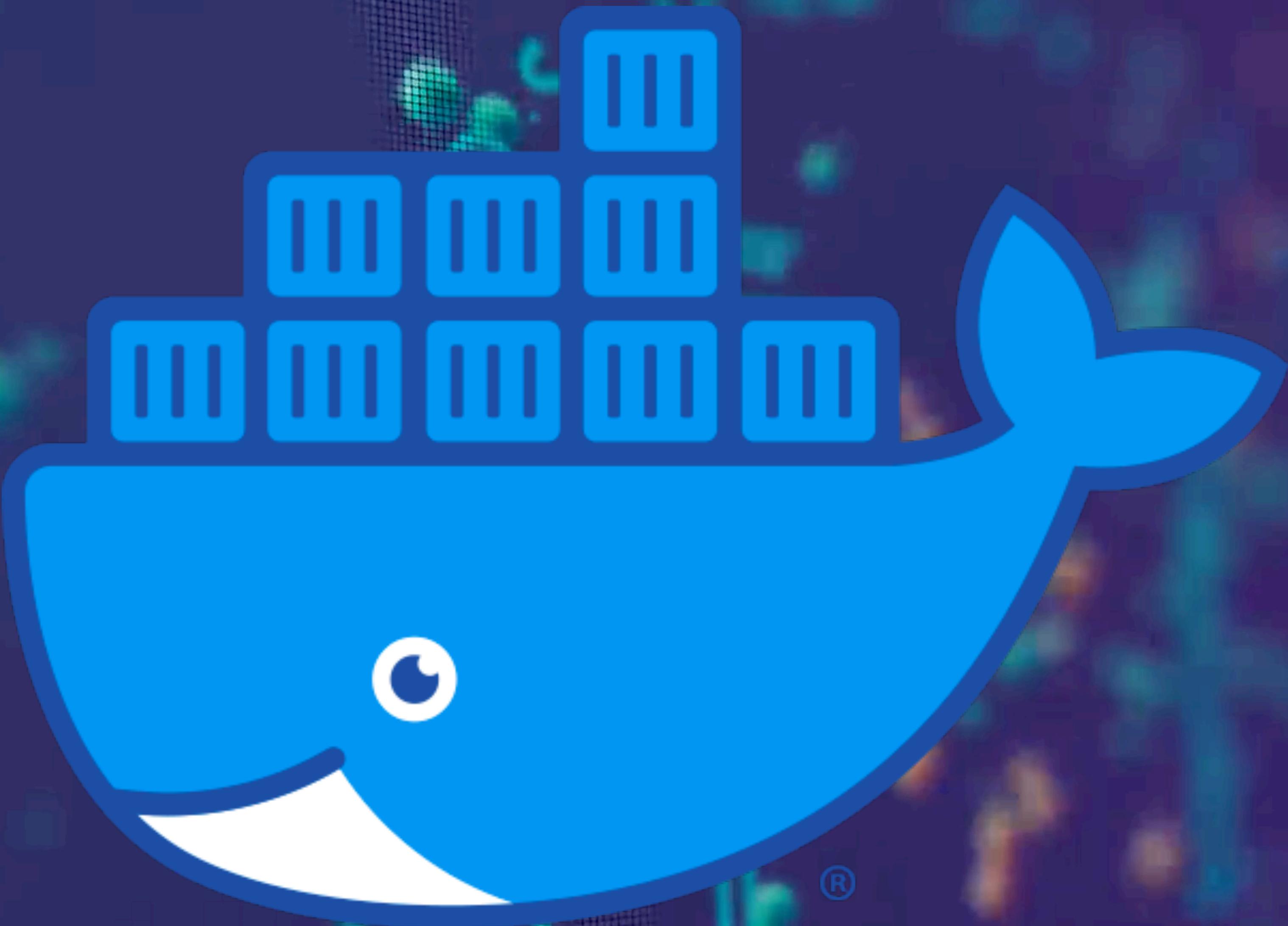


Programación Avanzada

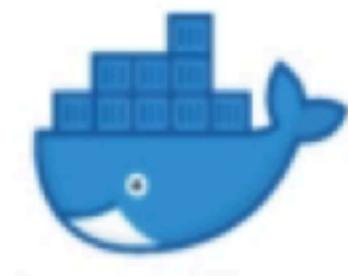
Clase 4

Dr (c).MTI.ING. Humberto Farias Aroca

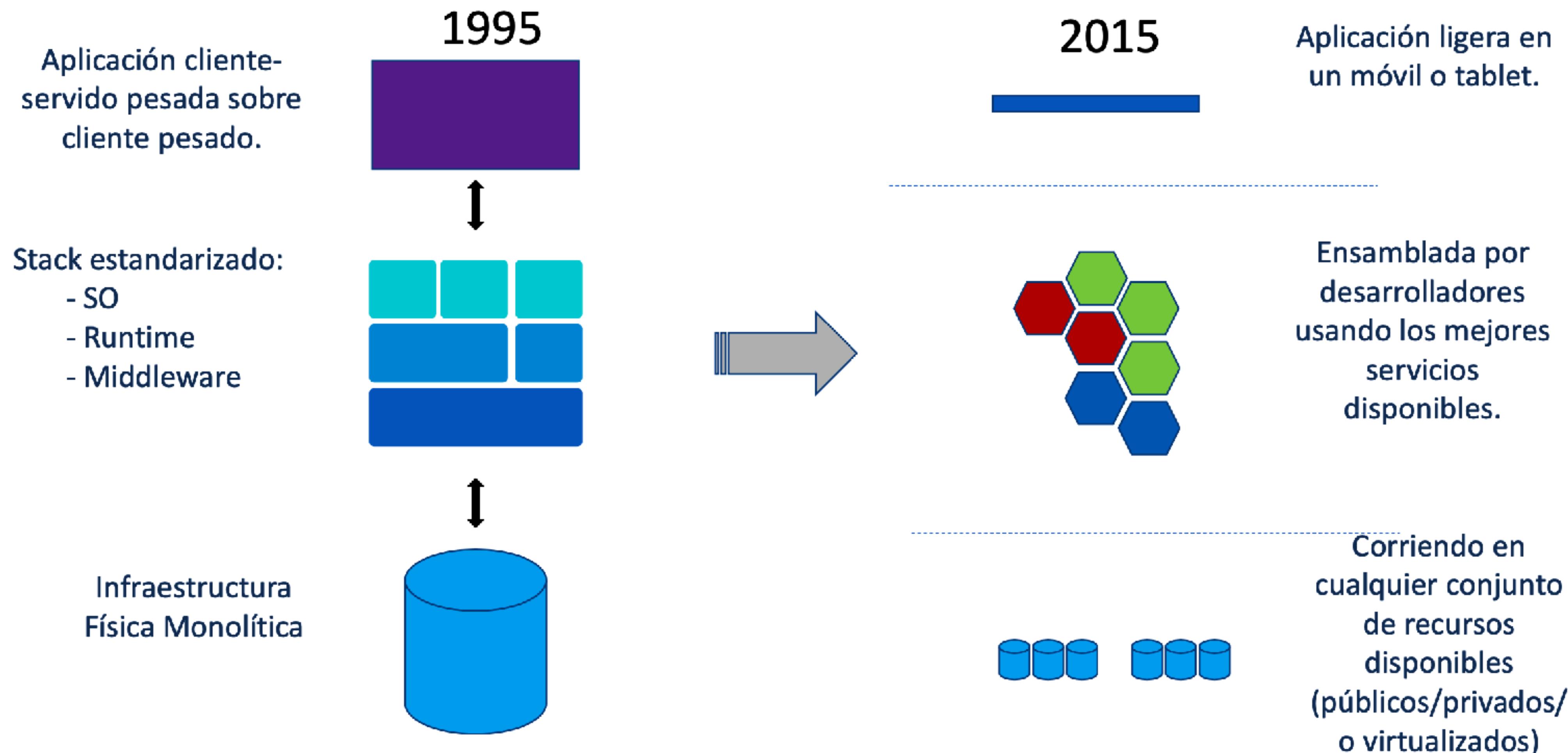


®

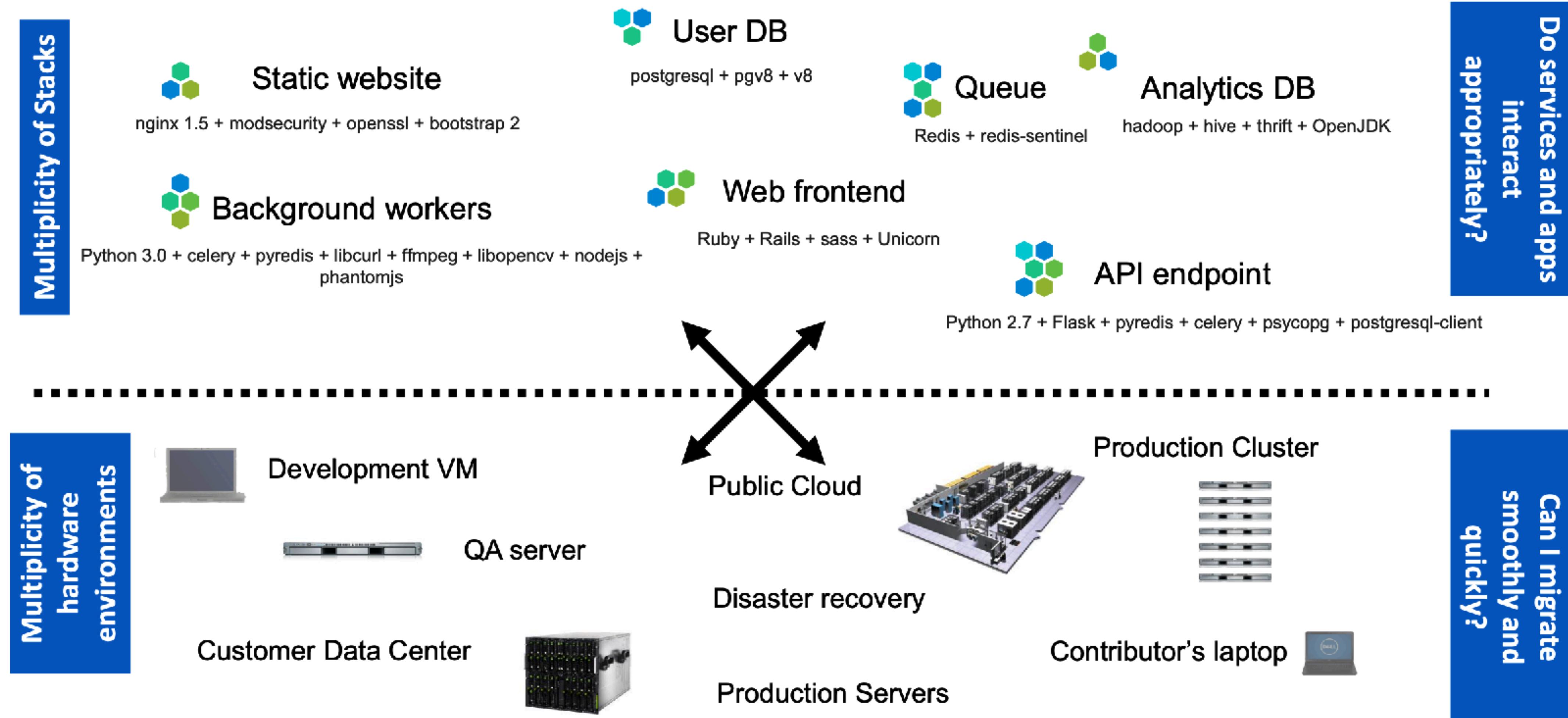
¿Qué es Docker?



Evolución: Aplicaciones



El problema que solventa Docker



El problema que solventa Docker

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
							

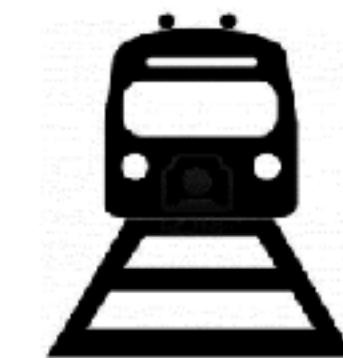
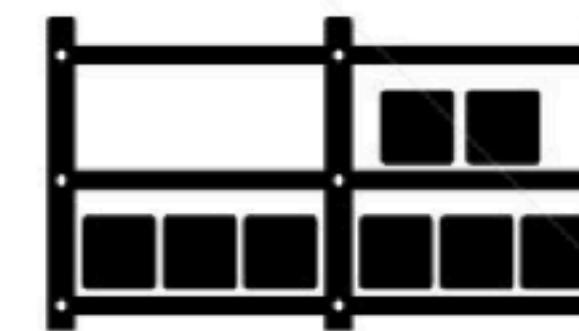
Símil: Transporte en antes de los años 60

Multiplicity of Goods



Do I worry about how goods interact (e.g. coffee beans next to spices)

Multiplicity of methods for transporting/storing



Can I transport quickly and smoothly (e.g. from boat to train to truck)

Problema $N \times M \rightarrow$ No escala

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

Solución: El Container



Multiplicity of Goods

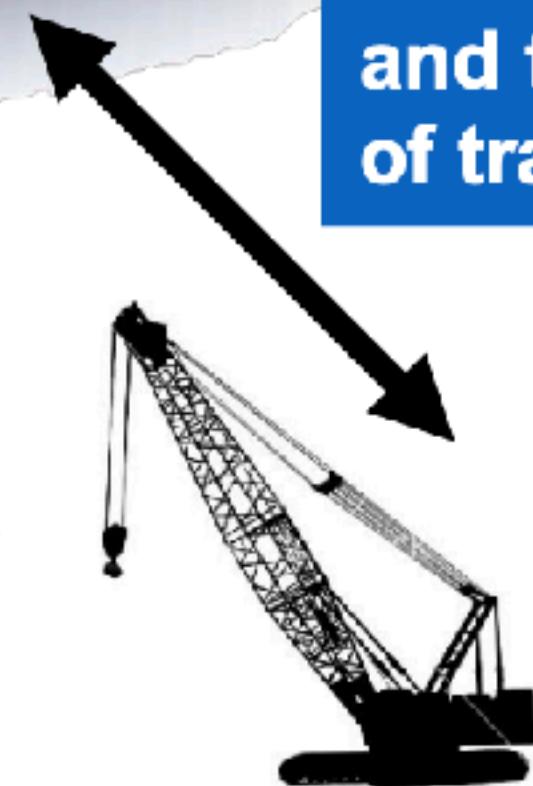
A standard container that is loaded with virtually any goods, and stays sealed until it reaches final delivery.



Do I worry about how goods interact (e.g. coffee beans next to spices)



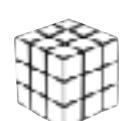
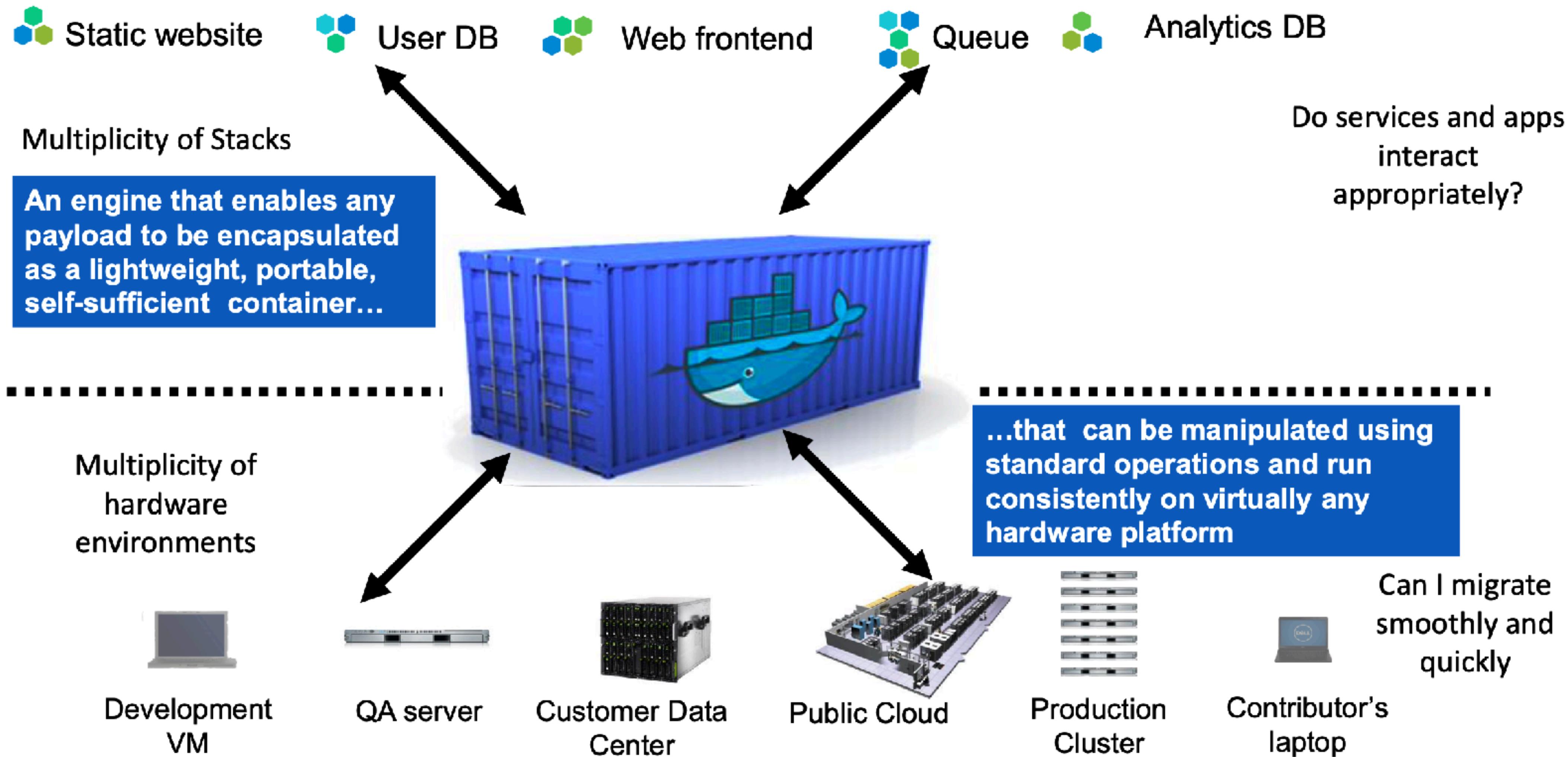
Multiplicity of methods for transporting/storing



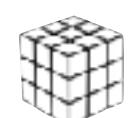
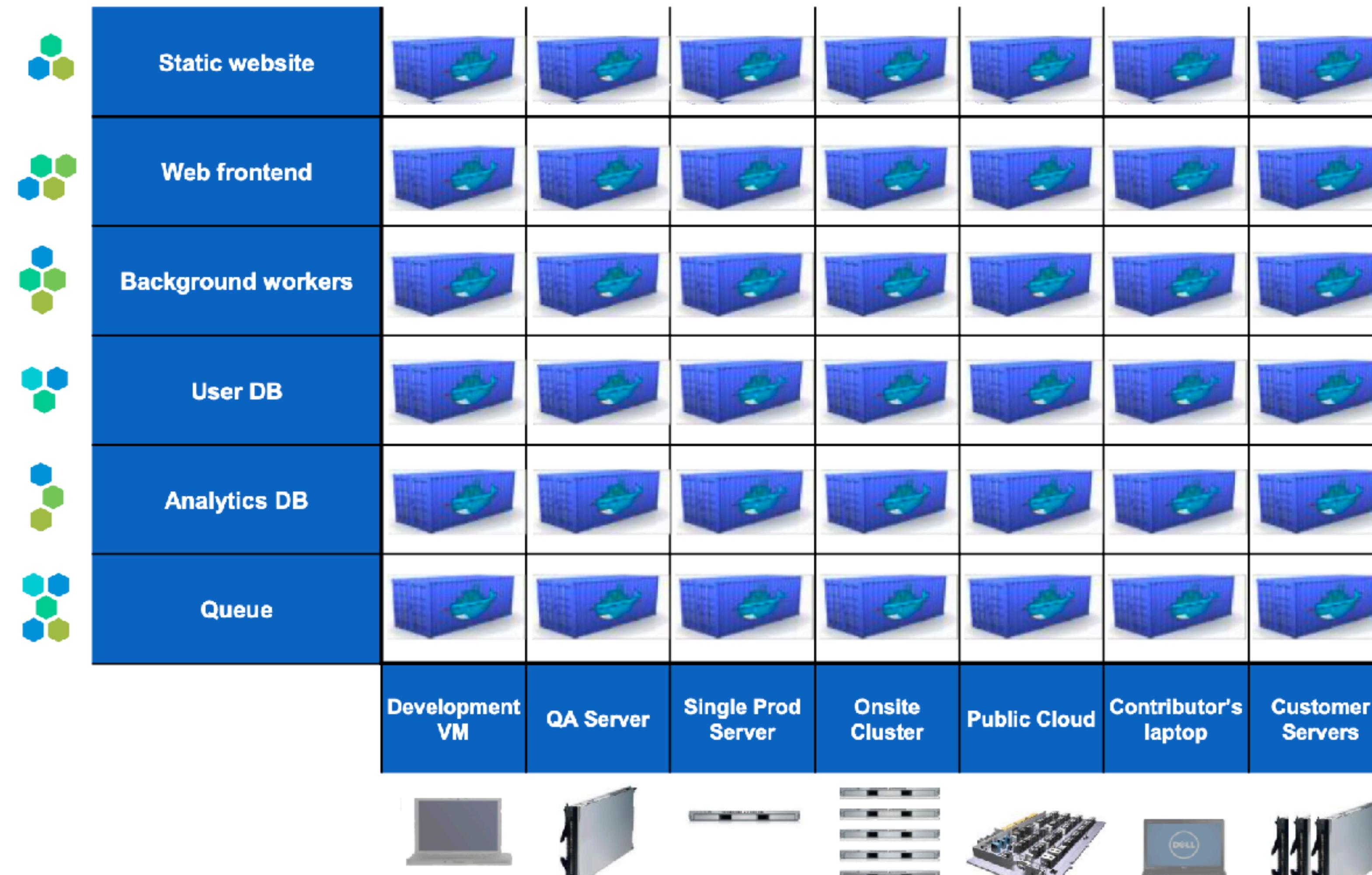
Can I transport quickly and smoothly (e.g. from boat to train to truck)

...in between, can be loaded and unloaded, stacked, transported efficiently over long distances, and transferred from one mode of transport to another

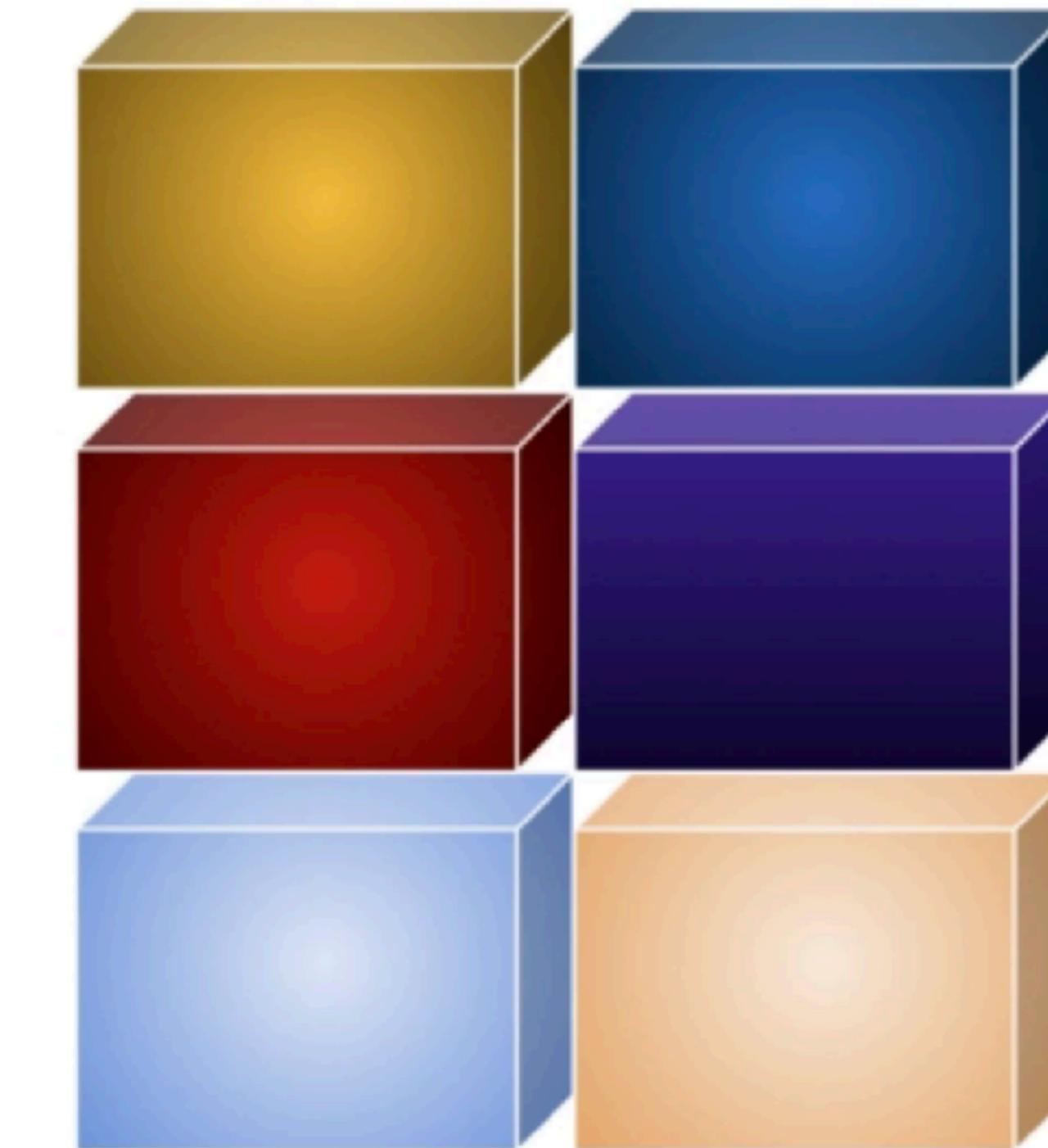
Docker



Docker elimina el problema estandarizando



Microservicios y monolitos



Microservicios

1

Retorno de la inversión y ahorro de costos



2

Estandarización y productividad



3

Eficiencia de Continuous Integration (CI)



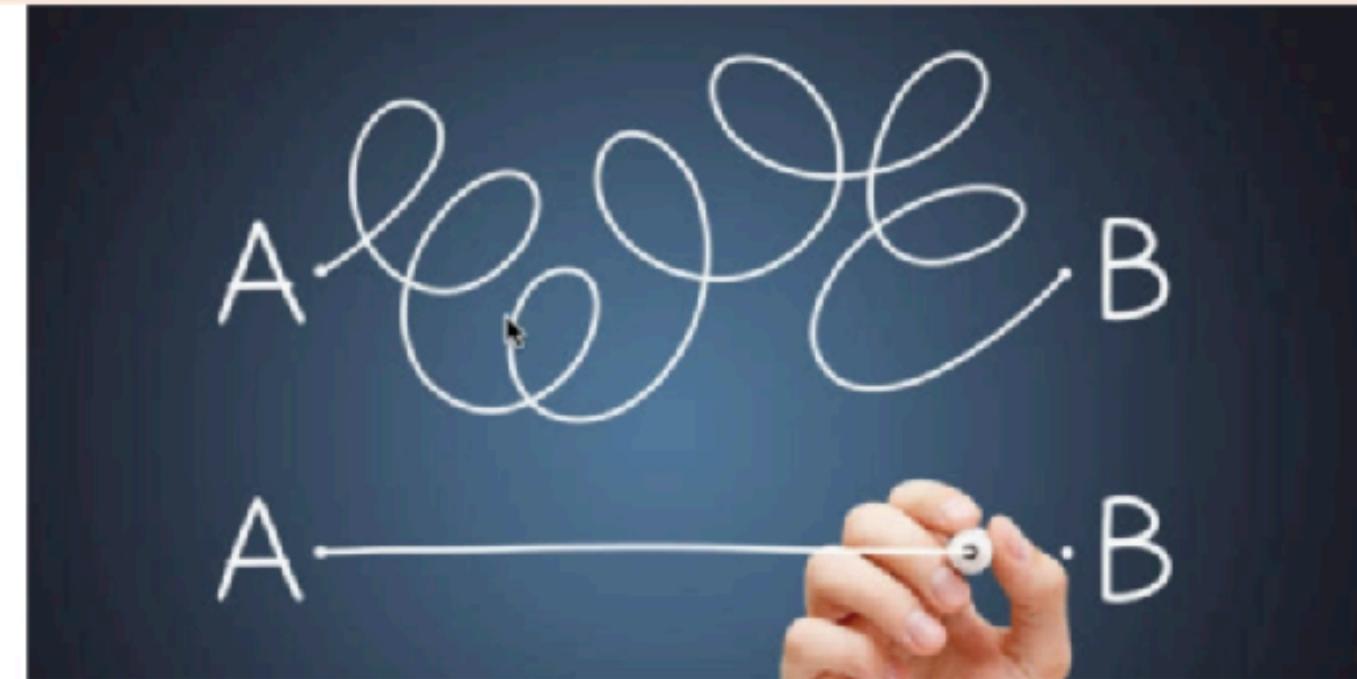
4

Compatibilidad y mantenibilidad



5

Simplicidad y configuraciones más rápidas



6

Despliegue rápido



7

Despliegue continuo y pruebas



8

Plataformas multi-nube

aws

9

Aislamiento



10

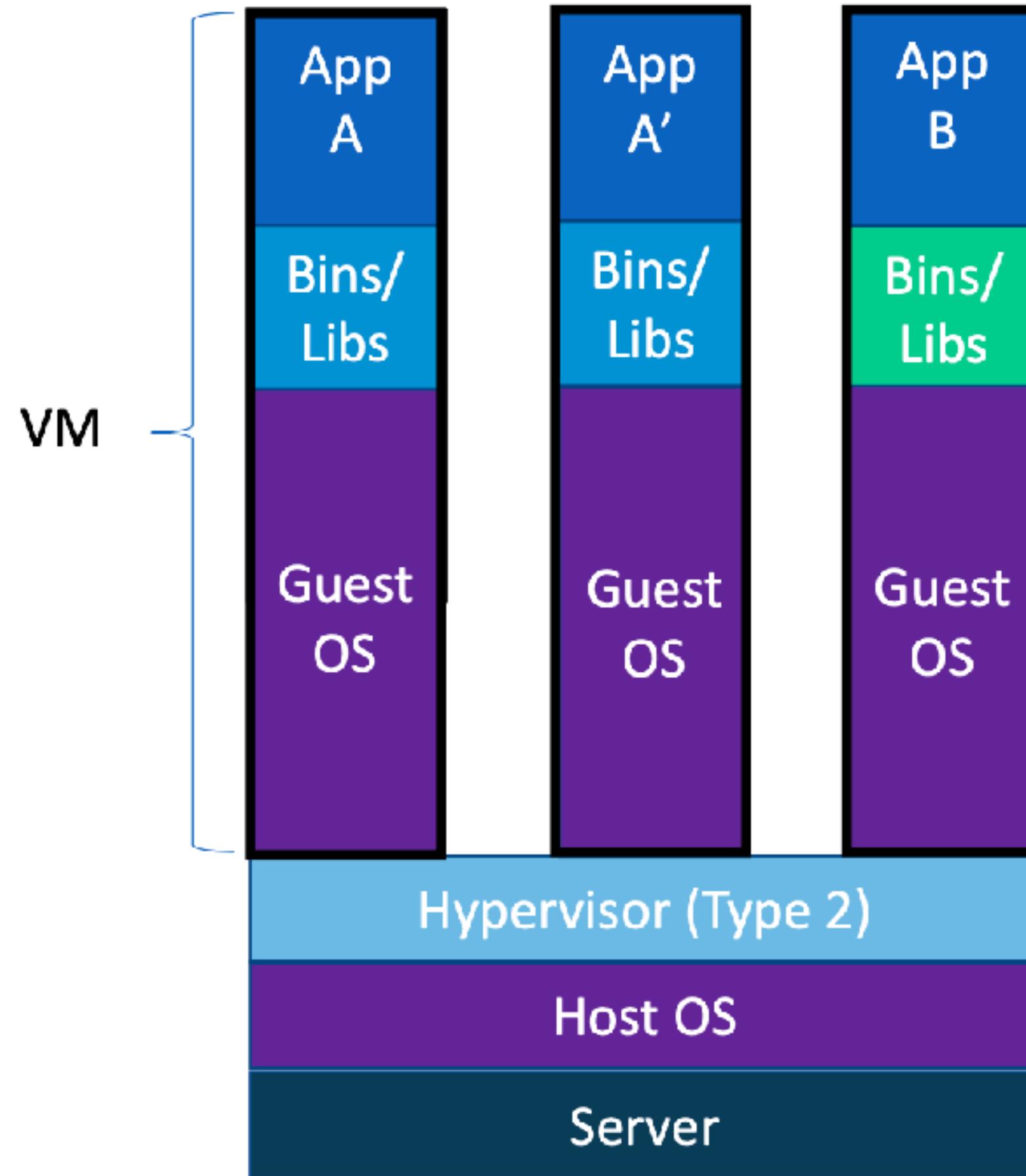
Seguridad



Docker vs Máquinas Virtuales (VMs)

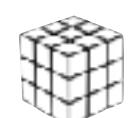
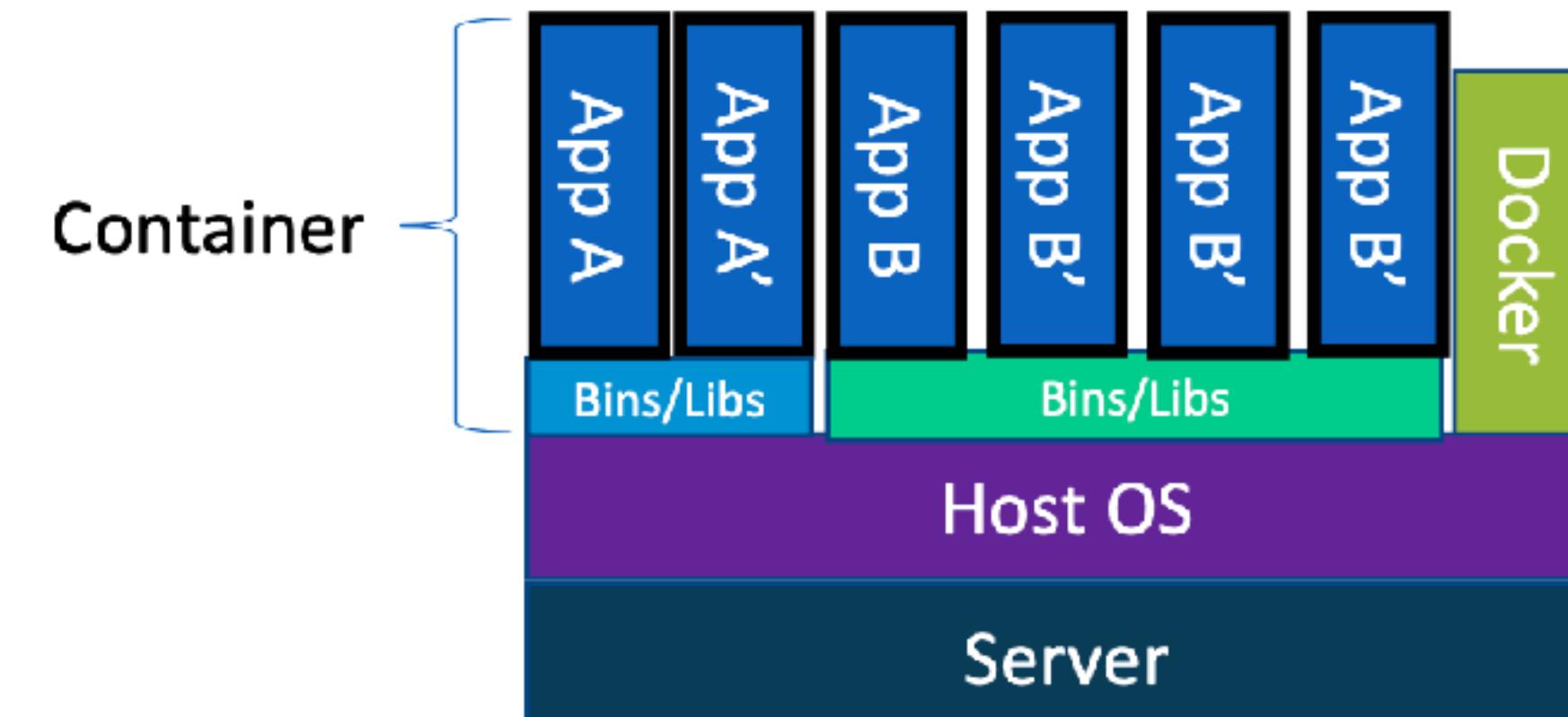


Contenedores vs Maquinas Virtuales



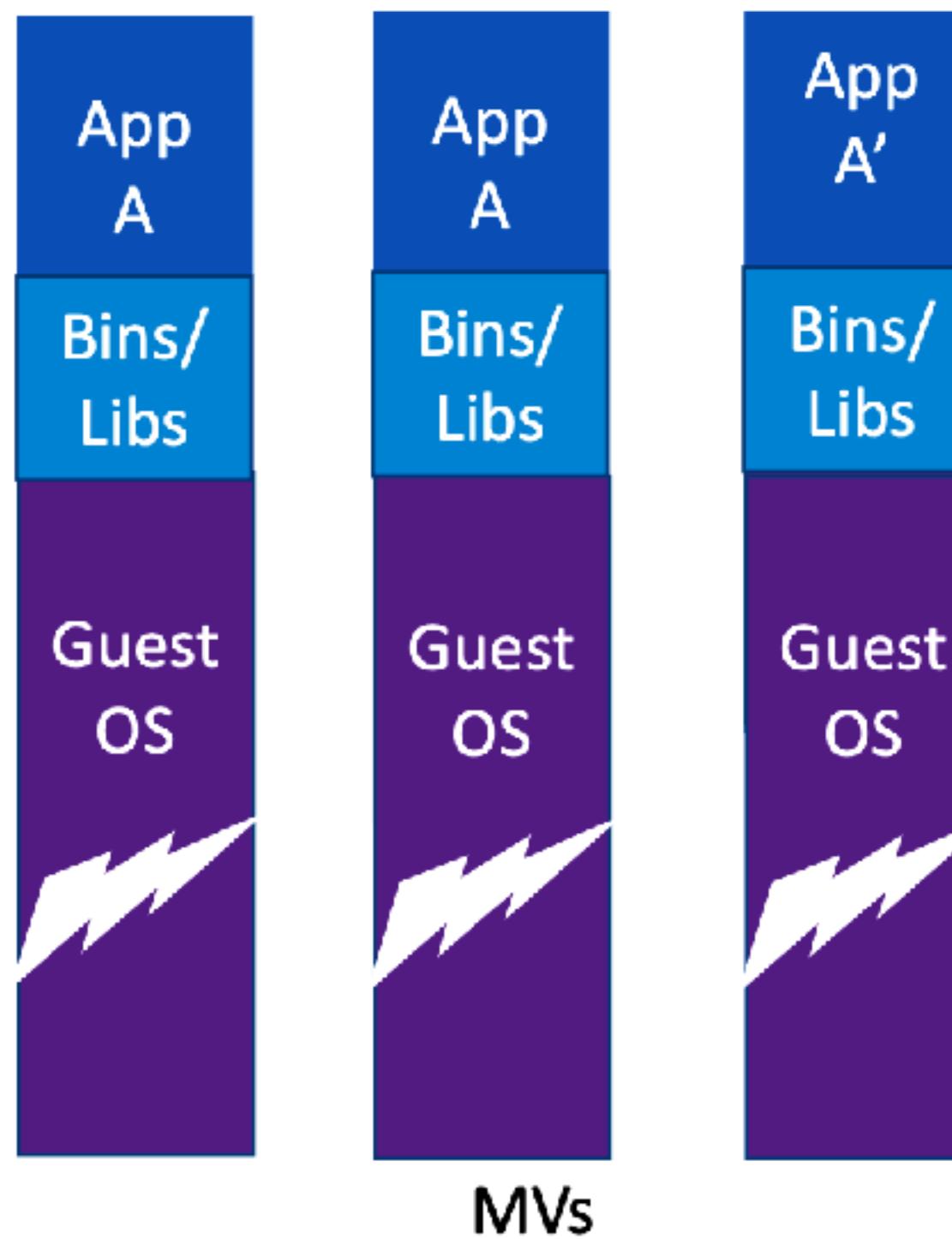
Los contenedores están aislados, pero comparten SO y opcionalmente binarios y librerías

Da como resultado despliegues más rápidos, con menos sobrecoste, más fáciles de migrar y reiniciar.



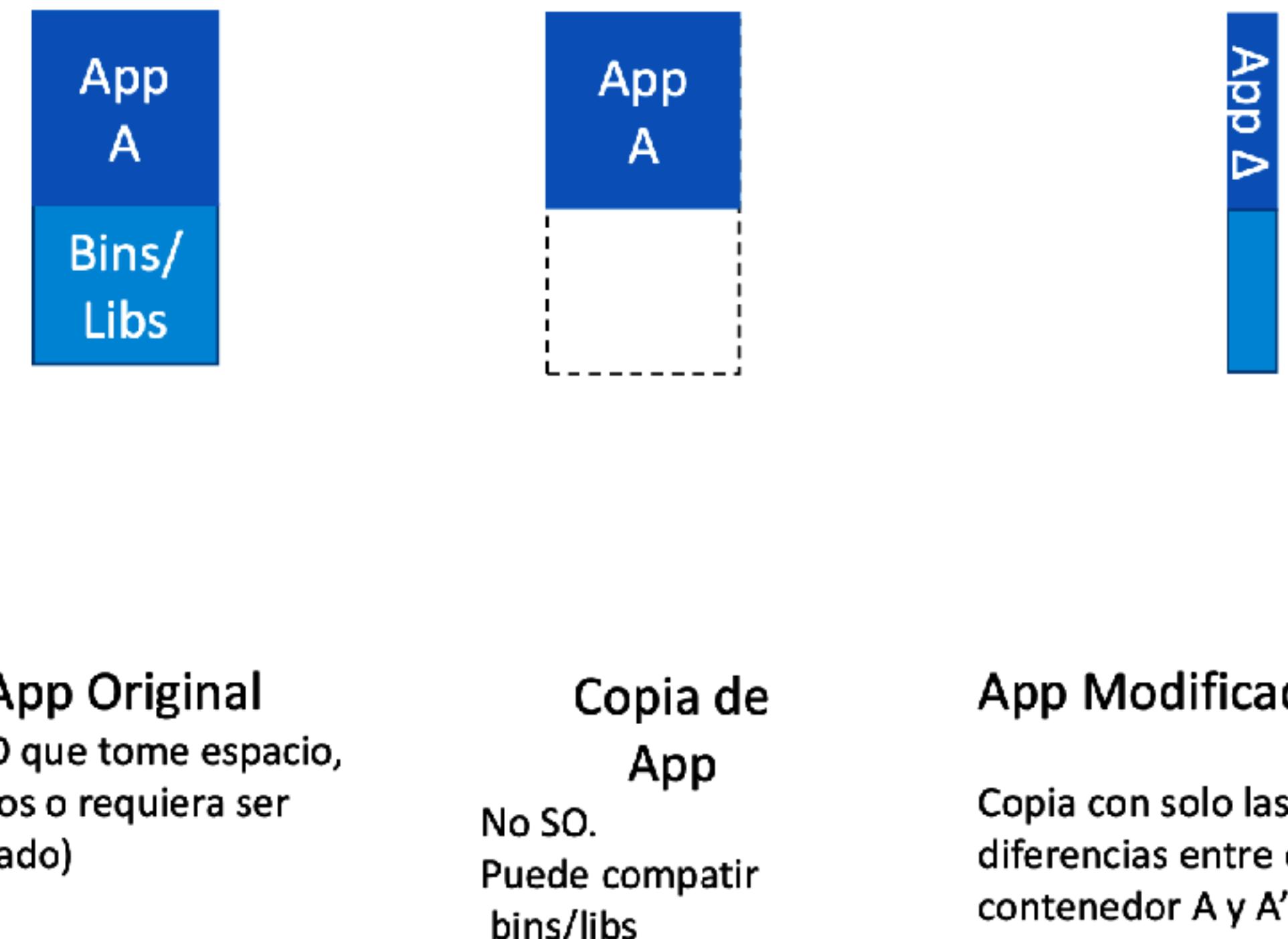
¿Porqué son contenedores Docker son ligeros?

MVs

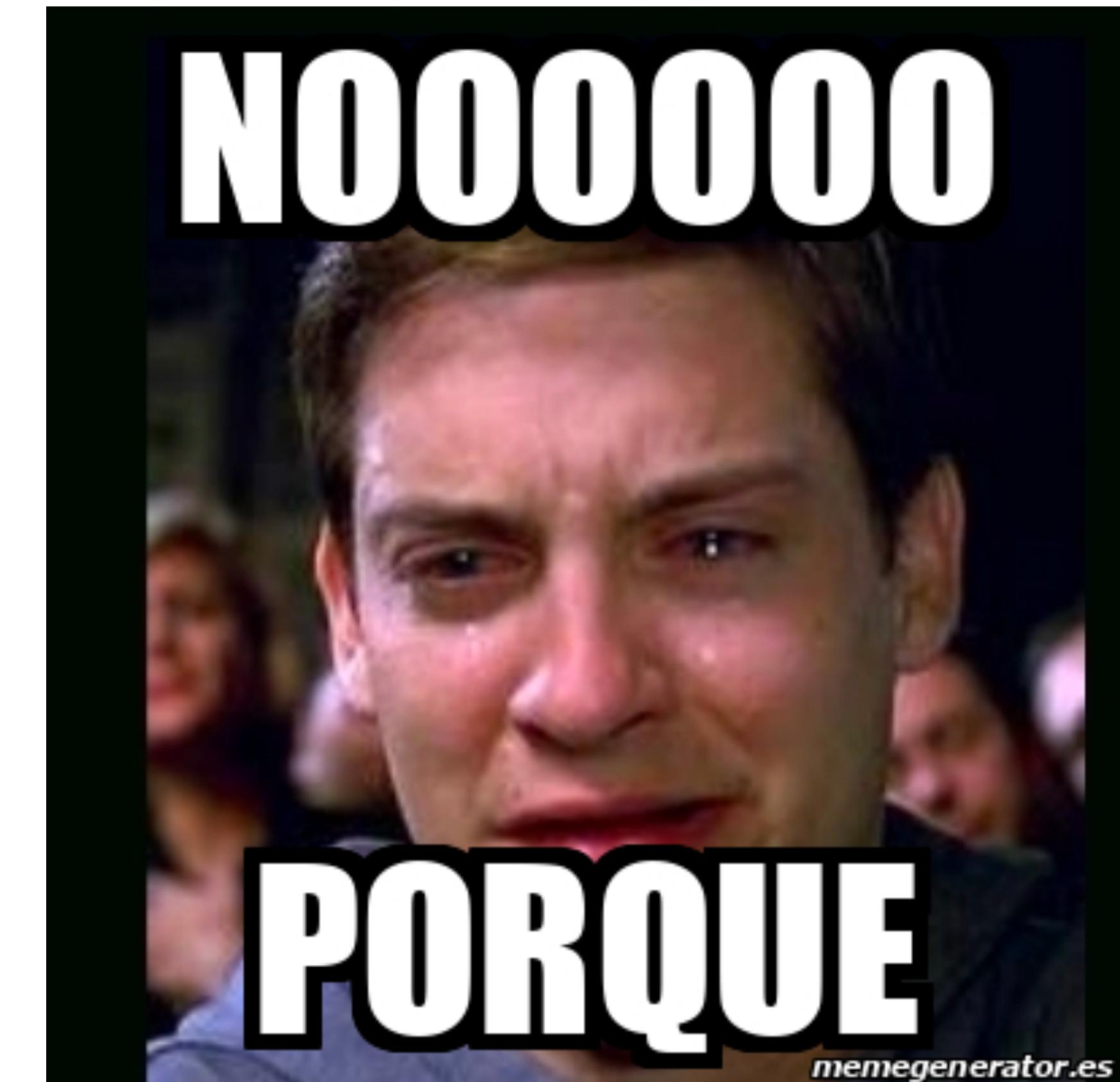


Cada pequeño cambio a una Aplicación
Requiere un nueva máquina virtual.

Contenedores

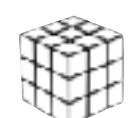


TAREA



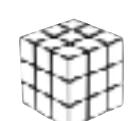
Beneficios para desarrolladores

- Entornos limpios, seguros y portables.
- Despliegues reproducibles (sin perdida de dependencias)
- Aislamiento de aplicaciones
- Tests, integración, empaquetado automatizado
- Menores problemas de compatibilidad
- Despliegues rápidos y baratos



Beneficios para devops

- Configura una vez, corre cientos
- Despliegues estandarizados y repetibles
- Elimina inconsistencias entre entornos (devel, qa, prod, etc.)
- Permite segregación de responsabilidades
- Mejora la velocidad de CI y CD
- Más ligeros que una MV



CONTINUOUS INTEGRATION (CI)

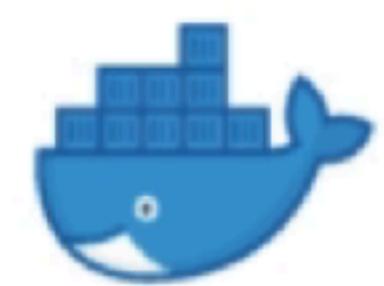


CONTINUOUS DELIVERY (CD)

LIBERACIÓN AUTOMÁTICA AL REPOSITORIO

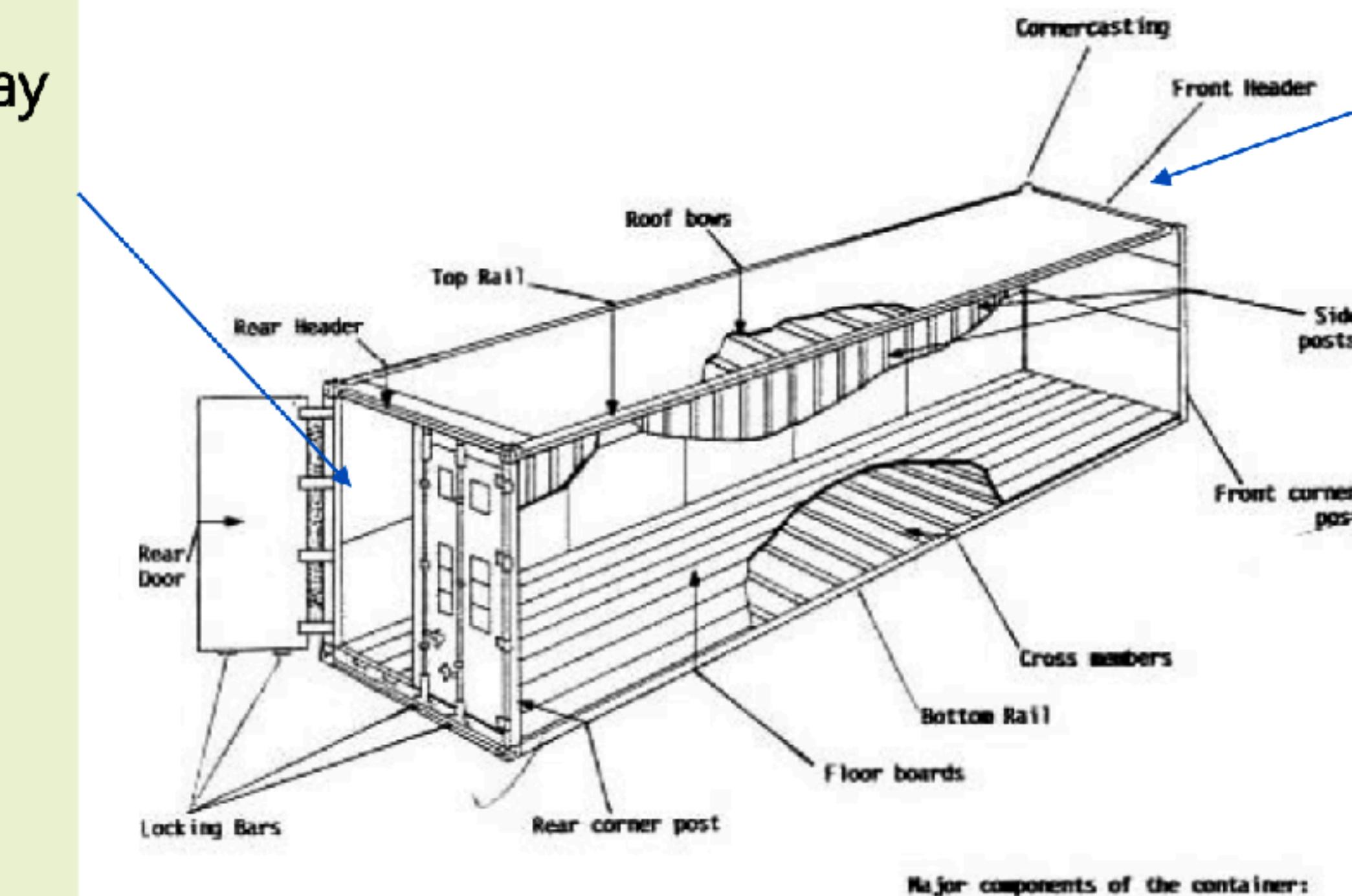
CONTINUOUS DEPLOYMENT (CD)

DESPLIEGUE AUTOMÁTICO HACIA PRODUCCIÓN



Separación de Responsabilidades (SoC)

- Alicia, la Desarrolladora
 - Precupada por "que hay dentro" del container.
 - Su código
 - Sus librerías
 - Su gestor de paquetes
 - Sus Apps
 - Sus datos
 - Todos los Servidores Linux son iguales



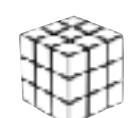
- Carmen, la Devop
 - Preocupada por lo que hay fuera del contenedor
 - Logging
 - Acceso Remoto
 - Monitorización
 - Configuración de Red
 - Todos los contenedores se arrancan, para, copian, mse migran del mismo modo.

¿Porqué?

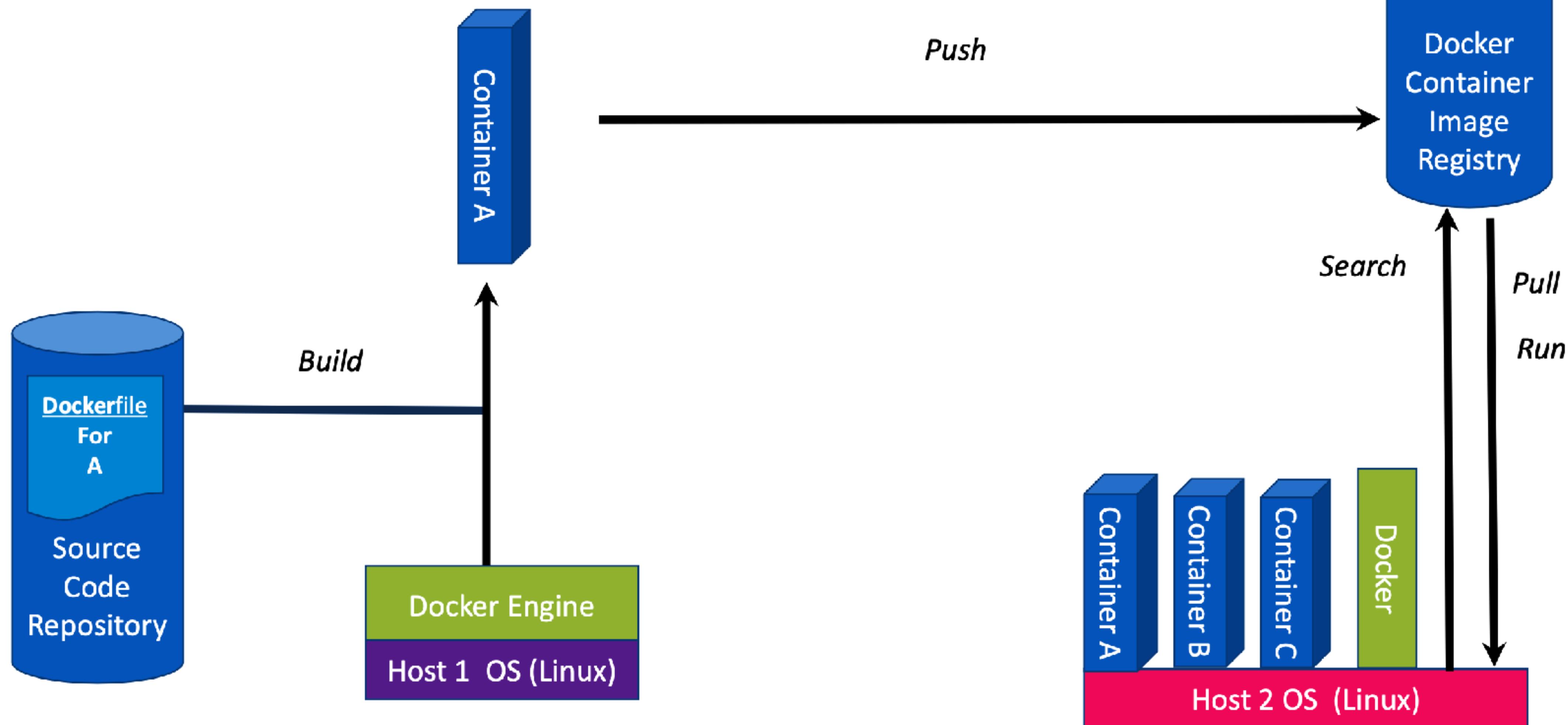
- Corre en cualquier sitio (Linux)
 - Sin importar version del kernel 2.6.32+
 - Sin importar distribución
 - Físico, virtual, nube o no.
 - Container & Arquitectura de Host deben emparejar
- Corre cualquier cosa:
 - Si corre en el host, corre en el container
 - Si corre en un kernel Linux, correrá

¿Qué es?

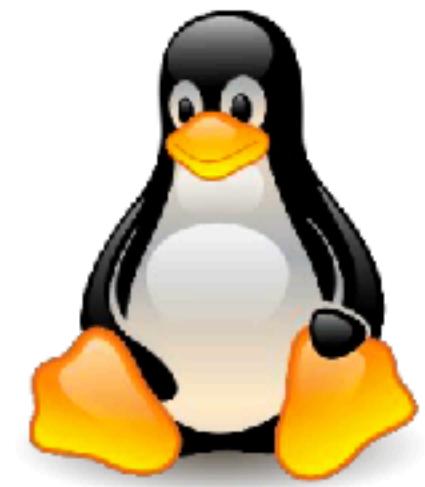
- Alto Nivel – MV ligera
 - Espacio de proceso propio
 - Configuración de red propia
 - Puede correr como `root`
 - Puede tener su propio `/sbin/init`
- Bajo Nivel – chroot con esteroides
 - Puede no tener su propio `/sbin/init`
 - Contenedor = proceso aislado
 - Comparte Kernel con el host
 - No emula dispositivo



Funcionamiento básico



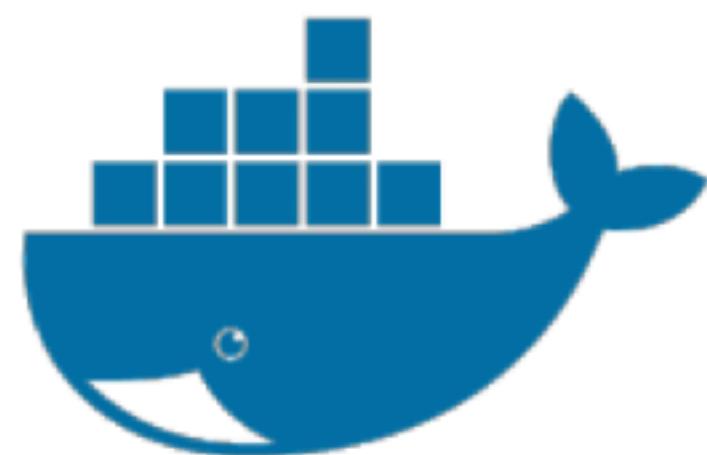
La tecnología bajo Docker



- Linux Kernel
 - Servicios de aislamiento de recursos cgroups
 - Kernel namespaces
 - Union file-systems aufs
 - Libvirt, LXC



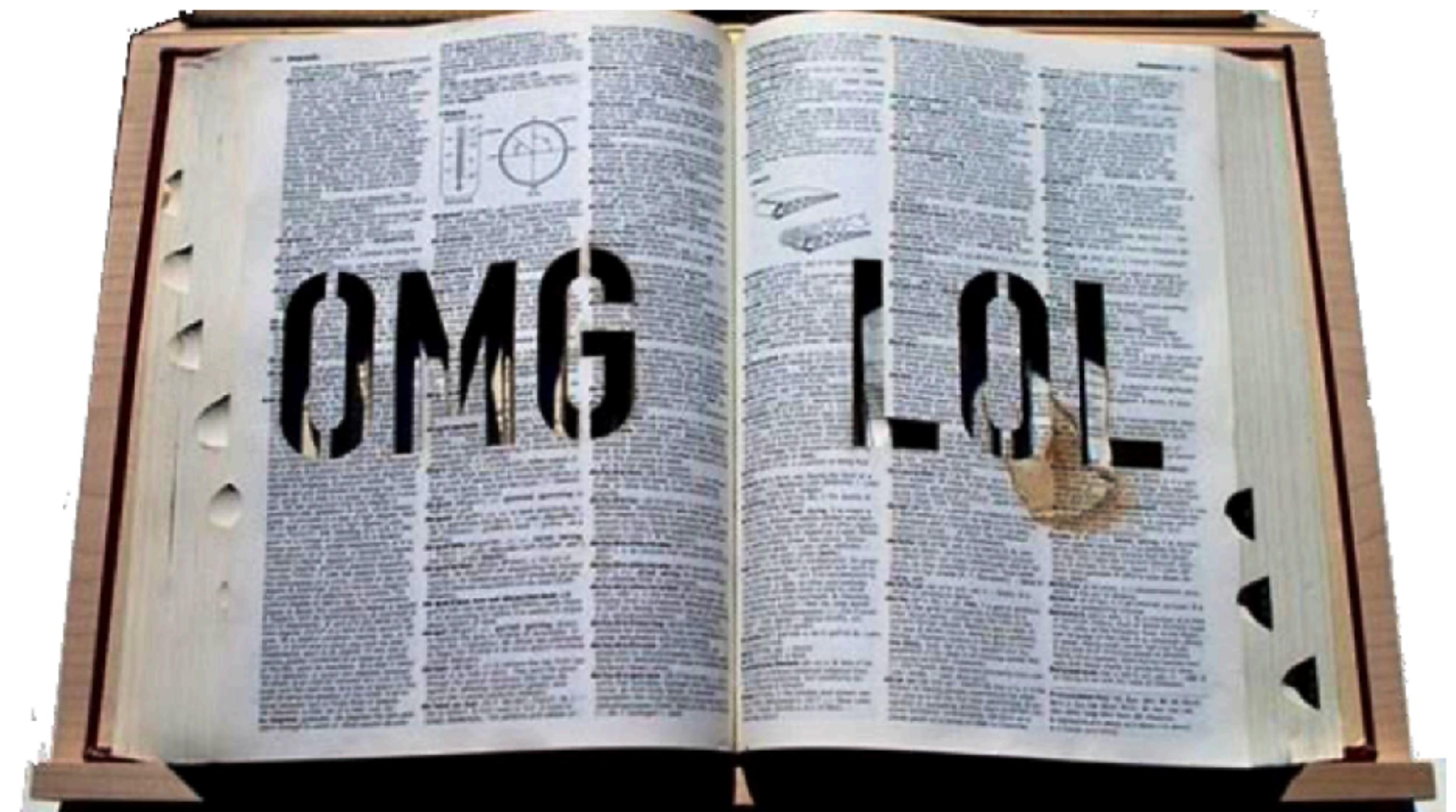
- Git
 - Control de versiones: delta a las imágenes de contenedores



- Registro (Docker Hub)

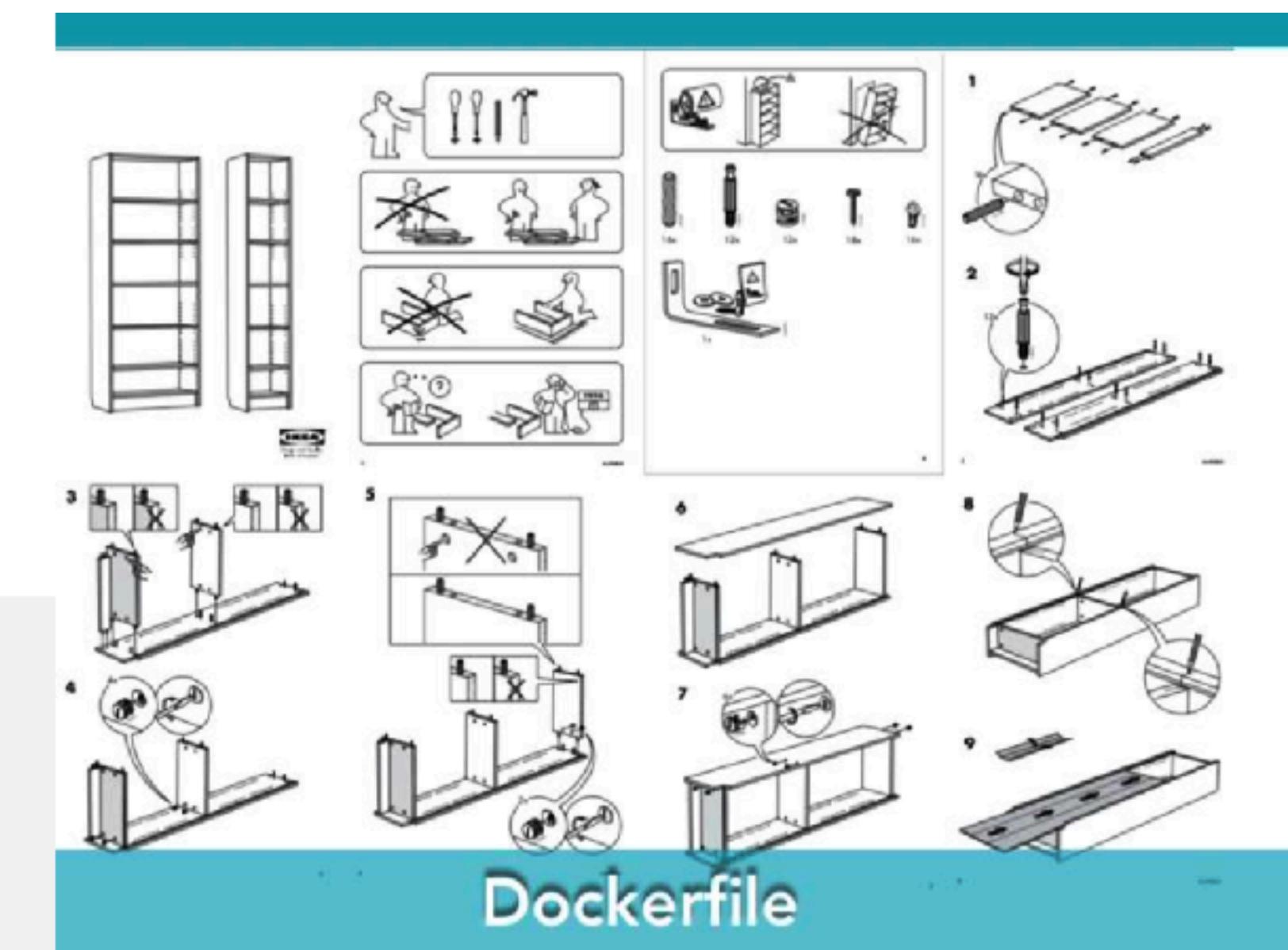
Glosario Docker:

- Veamos algo de jerga.



Glosario Docker: Dockerfile

```
# A basic apache server.  
  
FROM ubuntu:14.04  
  
MAINTAINER Kimbro Staken version: 0.1  
  
RUN apt-get update && apt-get install -y apache2  
  
ENV APACHE_LOG_DIR /var/log/apache2  
  
EXPOSE 80  
  
CMD ["/usr/sbin/apache2", "-D", "FOREGROUND"]
```



Glosario Docker: Container

- Contenedor
 - La aplicación paquetizada, aislada



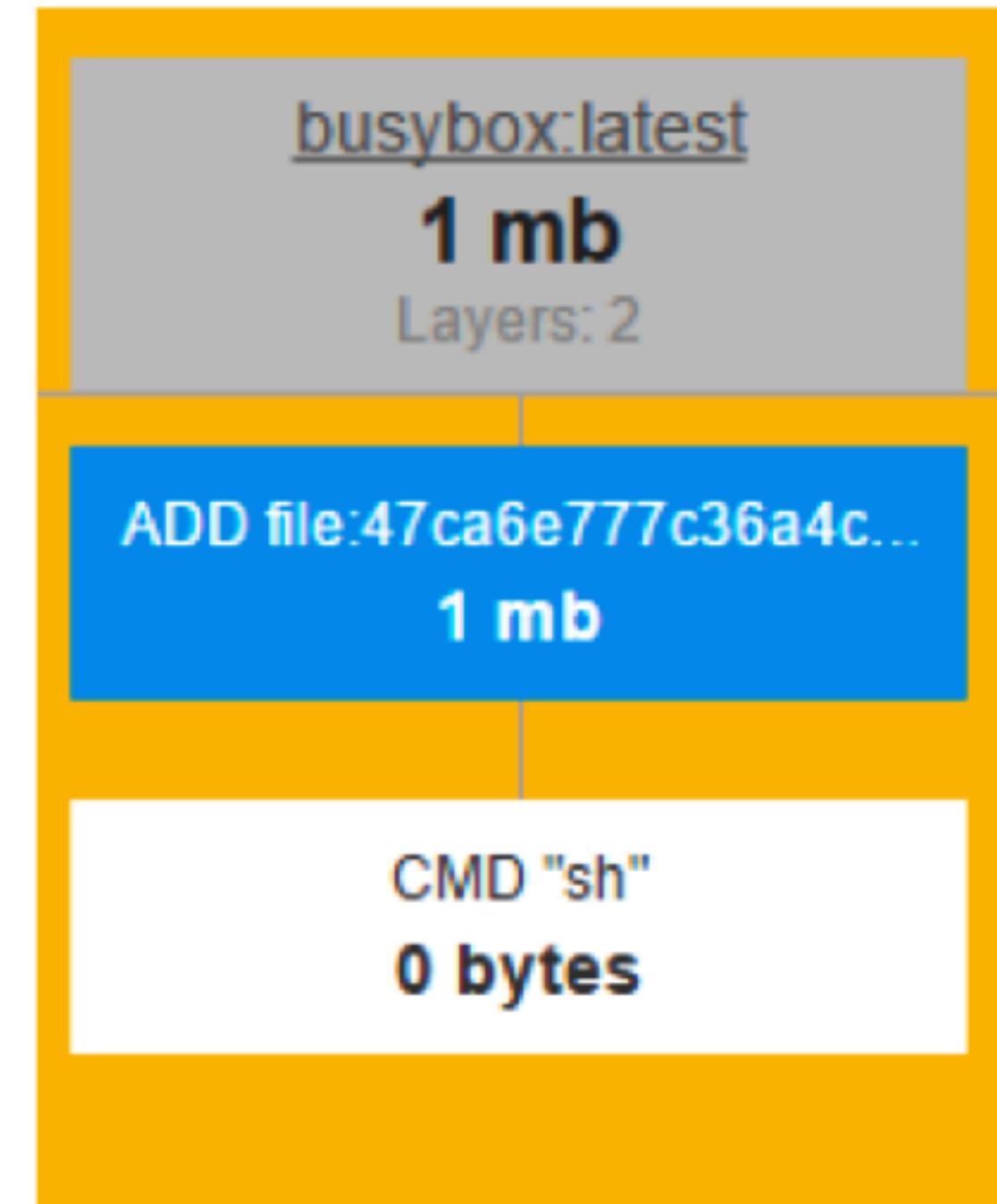
Glosario Docker: Host (*anfitrión*)

- Es el porta-contenedores
- Múltiples sabores: local, private cloud, public cloud. Virtual o físico.
- Expone los recursos



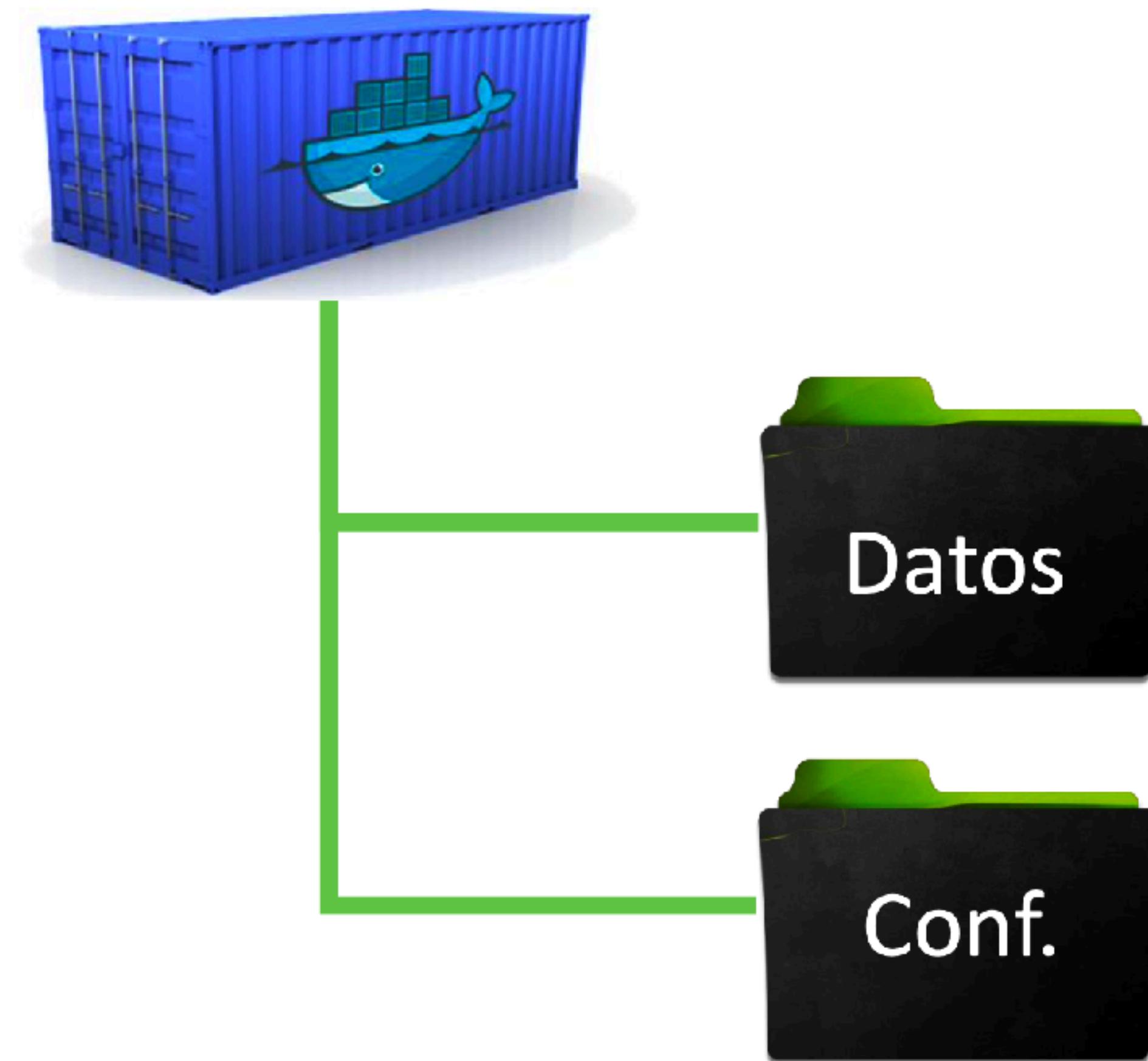
Glosario Docker: Imagen

- Fichero binario que contiene todo el sistema de ficheros de un contenedor.



Glosario Docker: Volumen

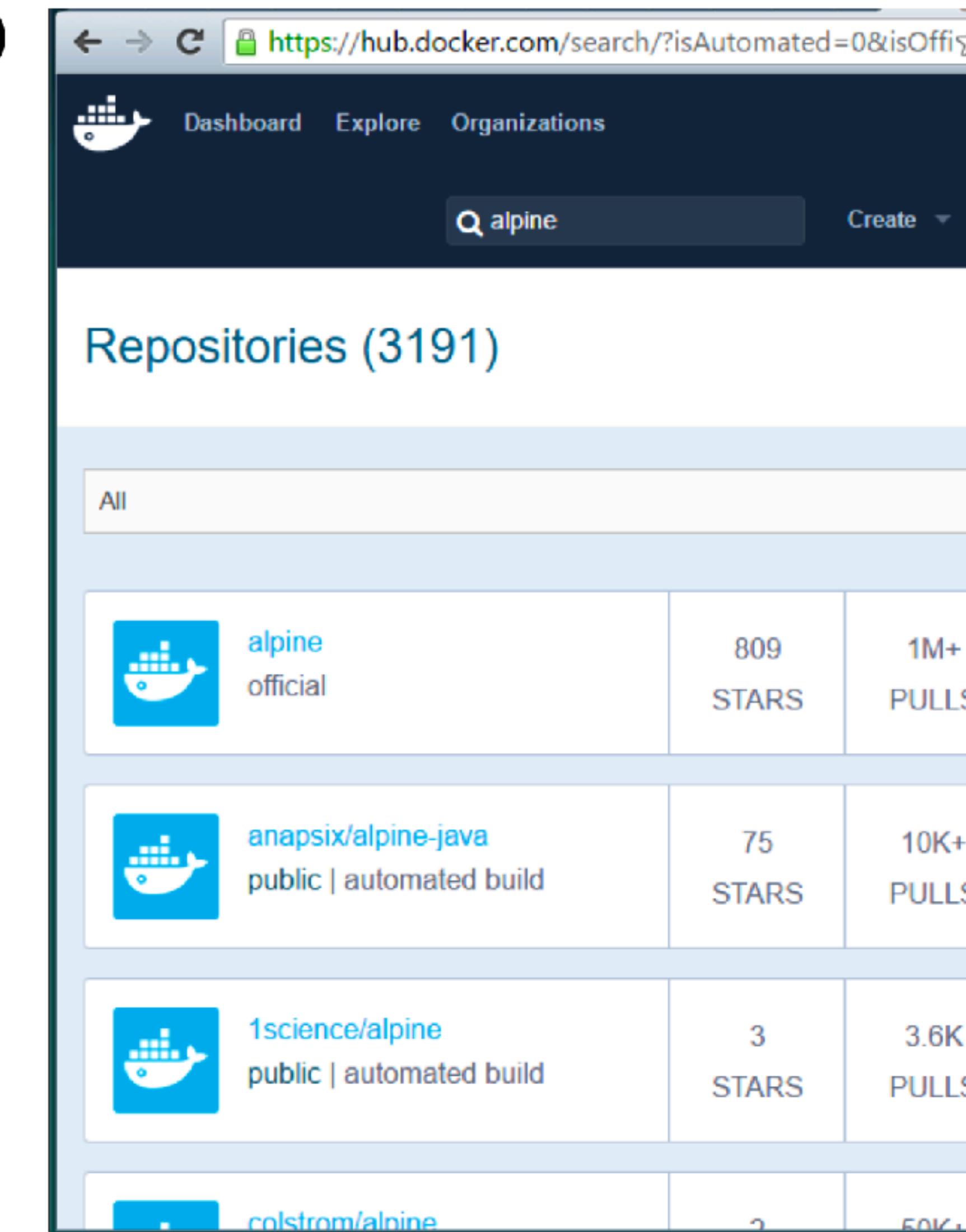
- Discos o directorios externos que podemos *montar* en el contenedor.
- Recursos externos (alojados en el *host*) que sobreviven al contenedor
- Configuración / Datos / Recursos



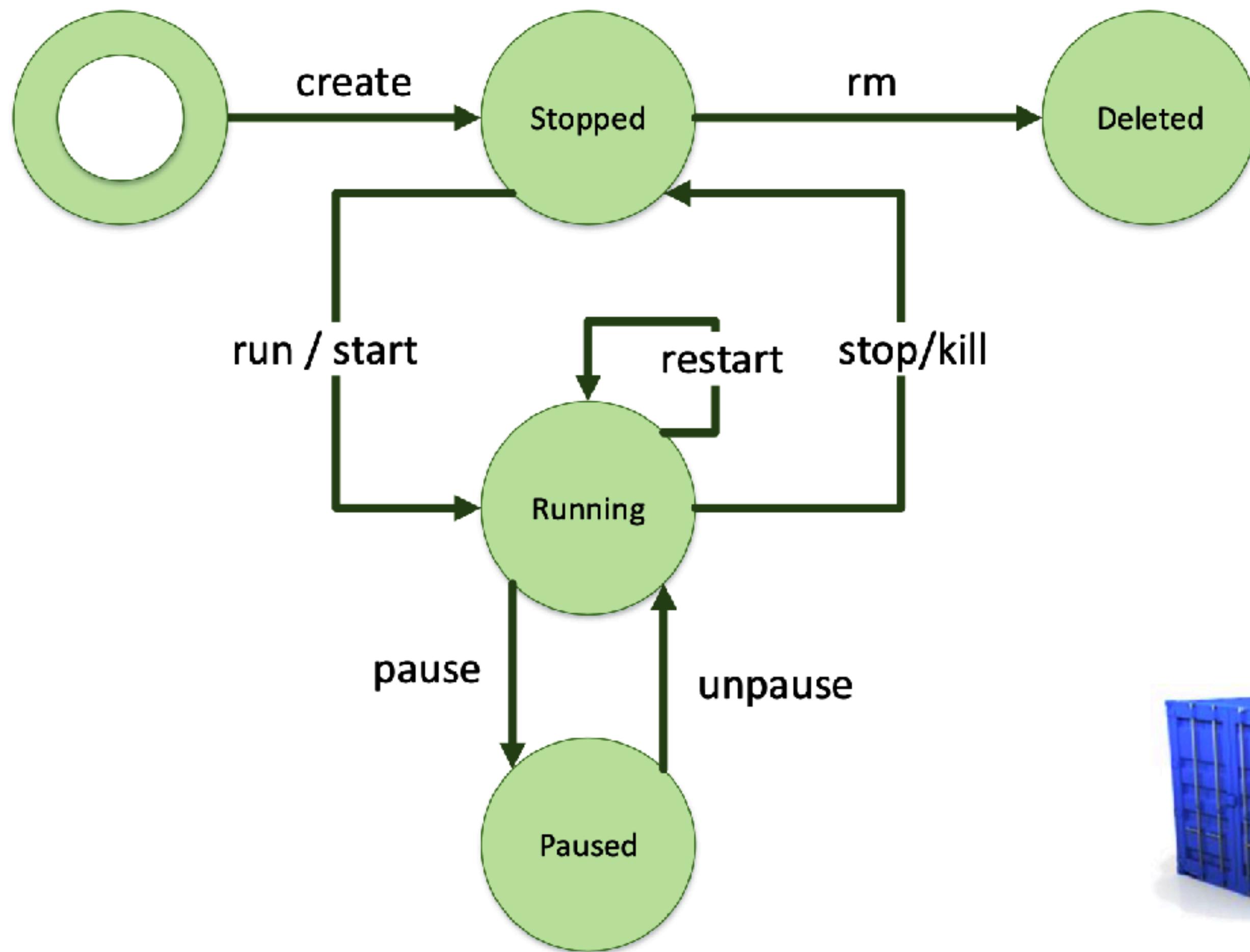
Glosario Docker: Registro

- Biblioteca de imágenes de contenedor
 - Listas para ser usadas
- Registro publico
 - Compartidas por la comunidad
 - Libre acceso
- Registro privado
 - Contenedores corporativos o privados

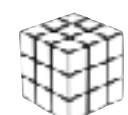
<https://hub.docker.com>

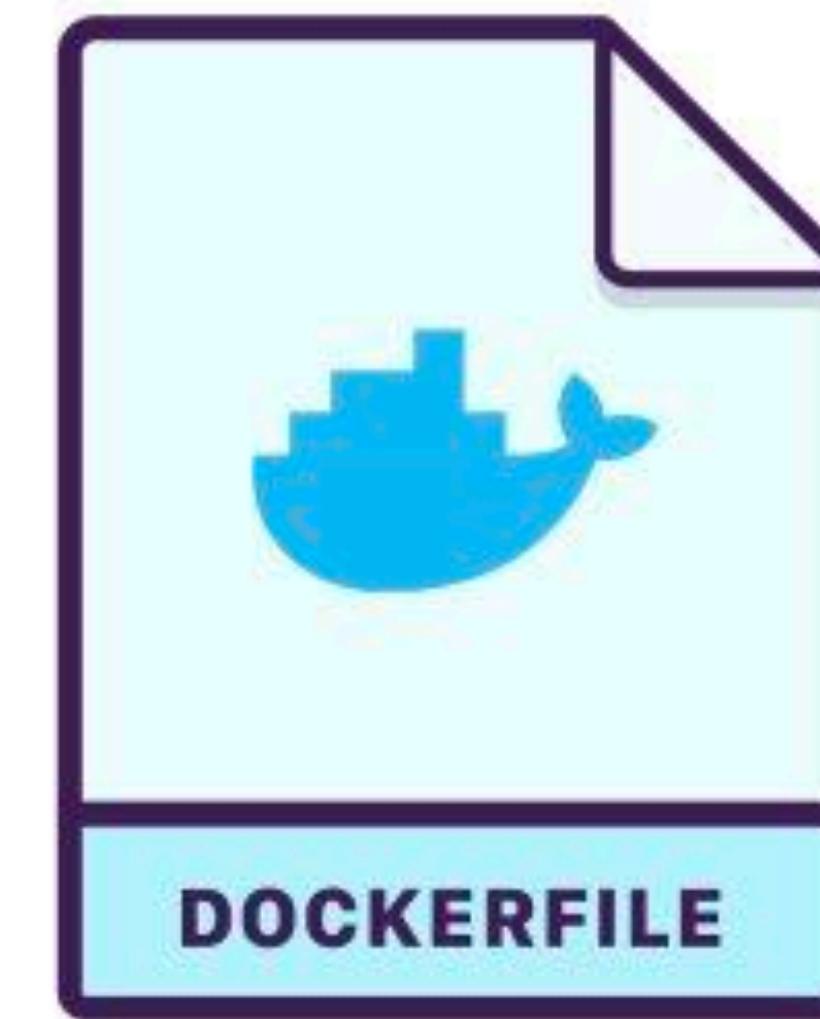


Ciclo de vida de un contenedor

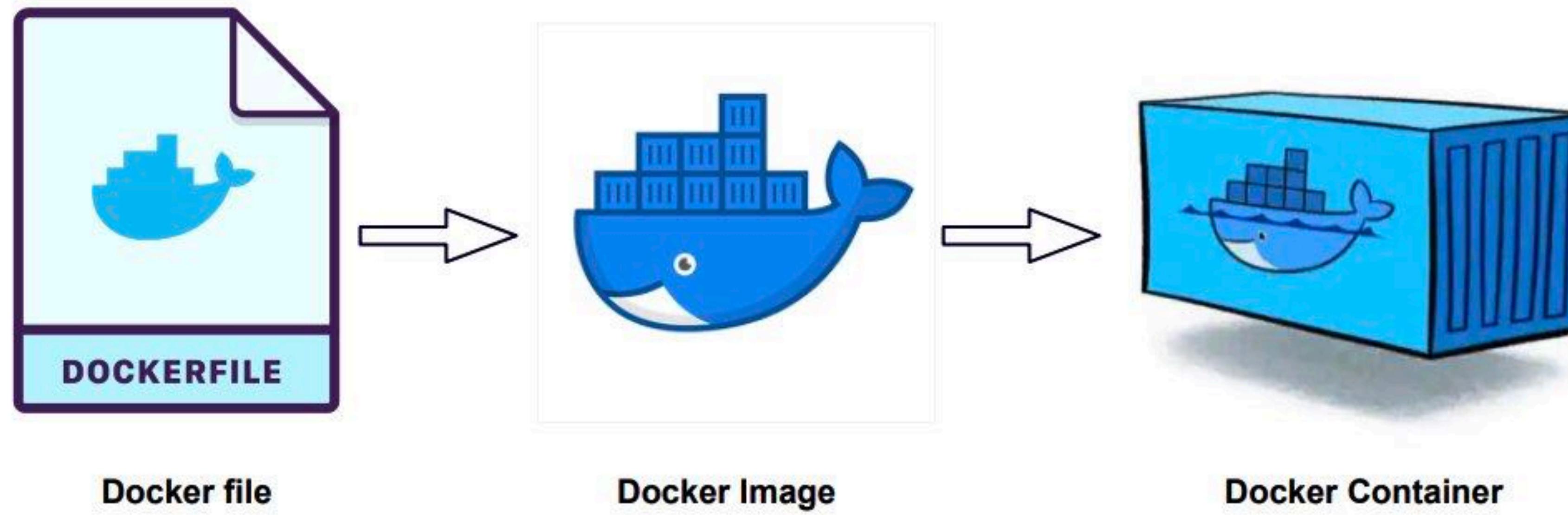


```
docker run -p 8088:80 -d --name welcome-to-docker docker/welcome-to-docker
```





Docker file



Como hacer un Dockerfile

Hay **diferentes maneras de construir nuestro dockerfile**, a continuación, se muestra un dockerfile mediante el cual construiremos una imagen con base **sistema operativo Ubuntu 14:04**, y sus sub-tareas serán:

- Instalación y actualización de paquetes del sistema operativo
- Establecer variable de entorno llamada **DEBIAN_FRONTEND**
- Instalación de git

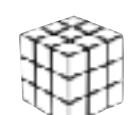
```
# Descarga la imagen de Ubuntu 14.04
FROM ubuntu:14.04

# Actualiza la imagen base de Ubuntu 14.04
RUN apt-get update

# Definir ambiente de entorno
ENV DEBIAN_FRONTEND noninteractive

# Instalar Git
RUN apt-get -qqy install git
```

Sacado de <https://github.com/harbur/docker-workshop/>



FROM: indica la imagen base sobre la que se construirá la aplicación dentro del contenedor.

Sintaxis:

```
FROM <imagen>
FROM <imagen>:<tag>
```

Por ejemplo la imagen puede ser un sistema operativo como Ubuntu, Centos, etc. O una imagen ya existente en la cual con base a esta queramos construir nuestra propia imagen.



RUN: nos permite ejecutar comandos en el contenedor, por ejemplo, **instalar paquetes o librerías** (apt-get, yum install, etc.). Además, tenemos **dos formas de colocarlo**:

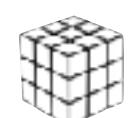
Sintaxis:

- **Opción 1 -> RUN <comando>**

Esta instrucción ejecuta comandos Shell en el contenedor.

- **Opción 2 -> [“ejecutable”, “ parametro1 ”, “ parametro2 ”]**

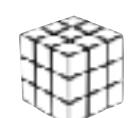
Esta otra instrucción bastante útil, que permite ejecutar comandos en imágenes **que no tengan /bin/sh**.



ENV -> establece variables de entorno para nuestro contenedor, en este caso la variable de entorno es DEBIAN_FRONTEND noninteractive, el cual nos permite instalar un montón de archivos .deb sin tener que interactuar con ellos.

Sintaxis:

ENV <key><valor>



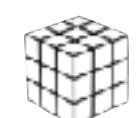
```
# Descarga la imagen de Ubuntu 16.04
FROM ubuntu:16.04

# Actualiza la imagen base de Ubuntu 16.04
RUN apt-get update

# Ejecuta el comando apt-get install y elimina determinados archivos y temporales
RUN apt-get install -y nginx \
    && rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/*

# Indica los puertos TCP/IP los cuales se pueden acceder a los servicios del container
EXPOSE 80

# Establece el comando del proceso de inicio del contenedor
CMD ["nginx"]
```



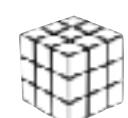
CMD -> esta instrucción nos **provee valores por defecto** a nuestro contenedor, es decir, mediante esta podemos definir una serie de comandos que solo se ejecutarán una vez que el contenedor se ha inicializado, **pueden ser comandos Shell** con parámetros establecidos.

Sintaxis:

CMD [“ejecutable”, “parámetro1”, “parámetro2”], este es el formato de ejecución.

CMD [“parámetro1”, “parámetro2”], parámetro por defecto para punto de entrada.

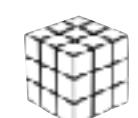
CMD comando parámetro1 parámetro2, modo shell

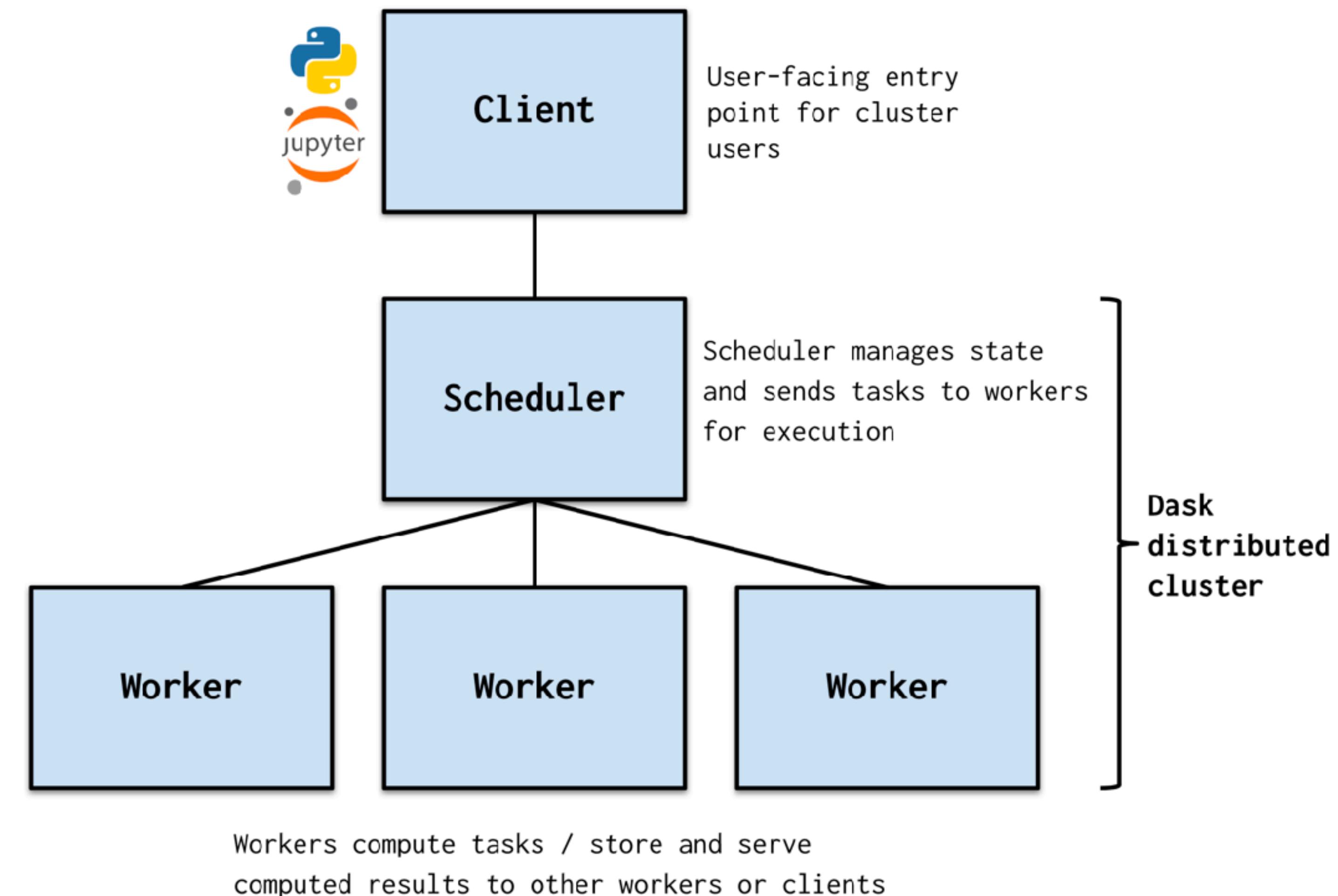


```
apple ~ /lab/docencia/avanzada/app/ cat Dockerfile
FROM node:12.22.1-alpine3.11

WORKDIR /app
COPY . .
RUN yarn install --production

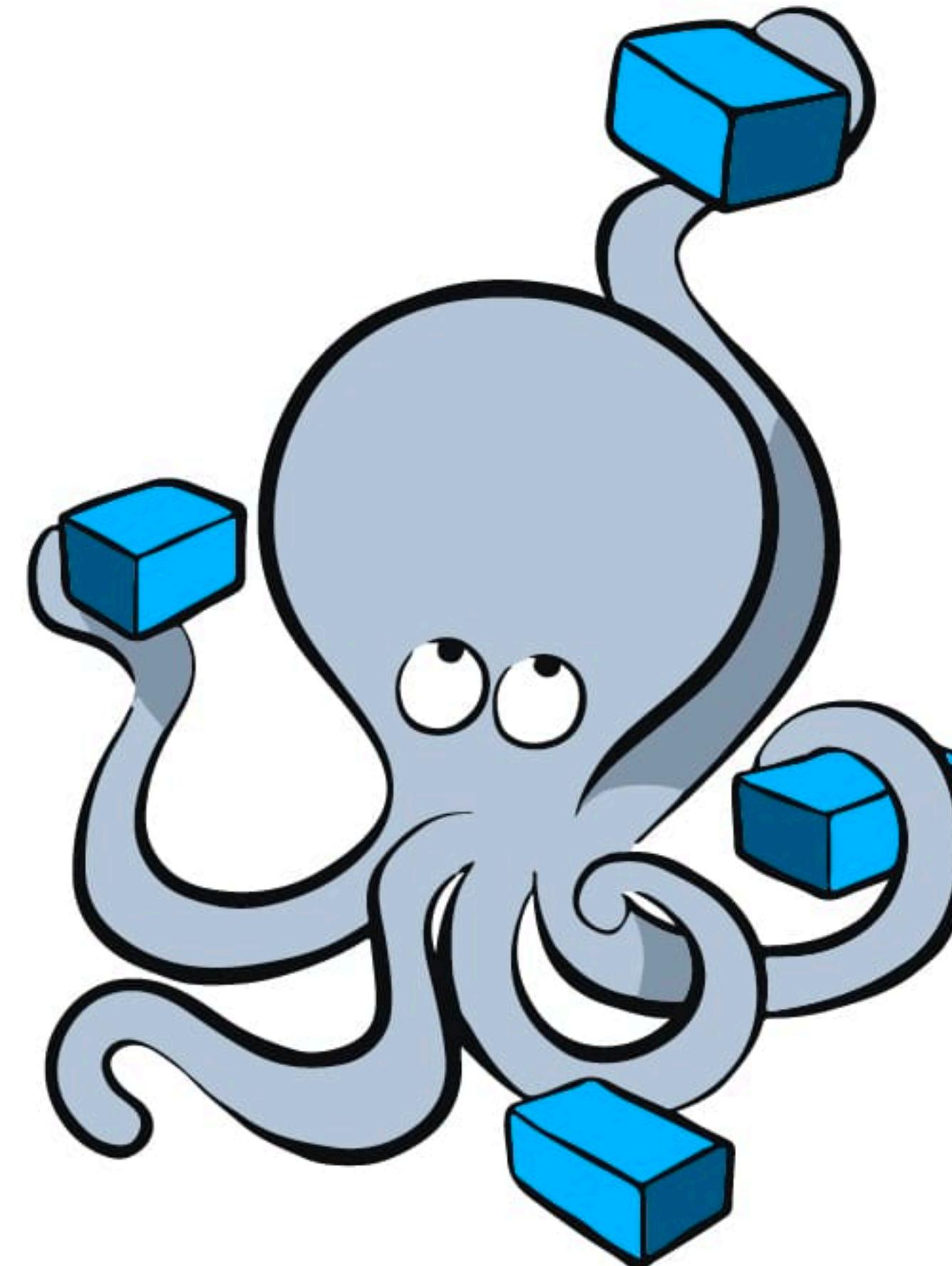
CMD ["node", "/app/src/index.js"]
```





- docker network create dask
- docker run --network dask -p 8787:8787 --name scheduler ghcr.io/dask/dask dask-scheduler # start scheduler
- docker run --network dask ghcr.io/dask/dask dask-worker scheduler:8786 # start worker
- docker run --network dask ghcr.io/dask/dask dask-worker scheduler:8786 # start worker
- docker run --network dask ghcr.io/dask/dask dask-worker scheduler:8786 # start worker
- docker run --network dask -p 8888:8888 ghcr.io/dask/dask-notebook # start Jupyter server





docker Compose

¿Qué es docker-compose?

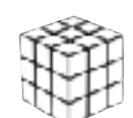
Docker Compose es una herramienta de Docker que orquesta contenedores en un mismo cliente. Consiste en un archivo de texto en formato YAML, que define de forma declarativa los contenedores que se van a desplegar, así como las dependencias entre ellos.

Al utilizar el paradigma declarativo, en este documento sólo se especifican las características de los contenedores deseados, y no cómo se despliegan. Para el despliegue, se basa en la definición de servicios, que referencian imágenes Docker de un registro y las características de los contenedores que se desean desplegar.



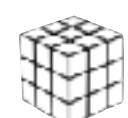
¿En qué momento podemos hacer uso de Docker Compose?

- En entornos que se encuentren en **Producción**
- En entornos que se encuentren en **Desarrollo**
- En entornos que se encuentren en **Fase de pruebas**
- En flujos de CI «**Integración Contínua**»



Servicios y recursos

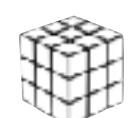
Dentro del archivo `docker-compose.yml` podemos definir diferentes recursos que desplegar. A la hora de desplegar nuestras aplicaciones, lo primero es definir los servicios que vamos a desplegar.



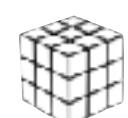
Servicios

Estos 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.

Para definir un servicio en el archivo `docker-compose.yml`, debemos definir un bloque con el nombre del servicio, y dentro una serie de propiedades que definen las características del contenedor:



- `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 nuestros 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`.

Todas las propiedades entre corchetes son opcionales, y si no se especifican, se toman valores por defecto. Es decir, sólo con definir la propiedad `image` ya se puede desplegar un contenedor con la imagen especificada.



Volúmenes

Cuando hablamos de volumen nos referimos a los volúmenes de Docker, el mecanismo que tienen los contenedores para persistir y compartir datos. Para ello, consumen o generan ficheros en un directorio del sistema de ficheros del host.

Para definir un volumen en el archivo `docker-compose.yml`, debemos definir un bloque con el nombre del volumen, y luego referenciarlo en los servicios que lo necesiten. Dentro de este bloque podemos definir las siguientes propiedades:



- **[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`.



Como podemos ver, todas las propiedades son opcionales, pero es recomendable definir al menos un nombre y la ruta en el sistema de archivos del host. Un ejemplo de volumen con estas propiedades sería:

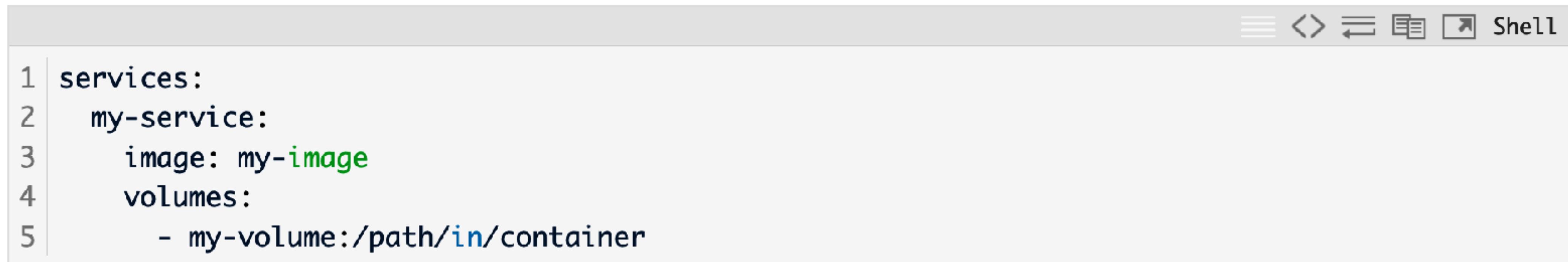


The screenshot shows a code editor window with a tab labeled "Shell". The code in the editor is a Docker volume configuration, likely defined in a Dockerfile or a separate configuration file. The code is as follows:

```
1 volumes:
2   my-volume:
3     driver: local
4     driver_opts:
5       type: none
6     device: /path/to/my-volume
7     o: bind
8     external: false
9     labels:
10    - "com.example.description=Volume for my service"
11    name: my-volume
12    scope: local
```



Referenciado en el servicio como:



A screenshot of a terminal window titled "Shell". The window contains a portion of a Docker Compose configuration file. The code is as follows:

```
1 services:
2   my-service:
3     image: my-image
4     volumes:
5       - my-volume:/path/in/container
```

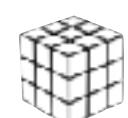
Recuerda que la ruta que definas en la propiedad `device` del bloque `volumes` debe existir en el sistema de archivos del host. No importa si es una ruta absoluta `/path/to/my-volume` o relativa `../path/to/my-volume`, siempre que exista.



Redes

Cuando desplegamos aplicaciones con múltiples módulos, por ejemplo, una aplicación web con un servidor web y una base de datos, las alojamos en contenedores separados. Para que estos contenedores puedan comunicarse entre sí, necesitamos definir una red en común, siguiendo el modelo **Container Network Model**:

- **SandBox** : Aisla 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**.



DOCKER

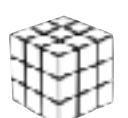
Siguiendo este modelo existen varias implementaciones en Docker:

Nombre	Alcance	Descripción
bridge	Local	Red por defecto de Docker. Crea una red virtual en el host, que conecta los contenedores por medio de un bridge virtual.
host	Local	Deshabilita el aislamiento entre los contenedores y el host, no hace falta exponer puertos
overlay	Global	Permite conectar múltiples demonios Docker entre sí y habilitar la comunicación entre servicios distribuidos .
ipvlan	Global	El usuario obtiene el control total del direccionamiento IPv4 e IPv6 de la red.
macvlan	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.
none	Local	Deshabilita toda la gestión de red, normalmente usado en conjunto con un driver de red propio.



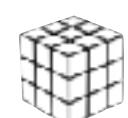
```
1 version: '3.7' # Versión de docker-compose, depende de Docker
2 services:
3   web:
4     image: nginx
5     ports:
6       - 80:80
7     volumes:
8       - web_data:/usr/share/nginx/html
9     networks:
10    - web_net
11
12   db:
13     image: postgres
14     volumes:
15       - db_data:/var/lib/postgresql/data
16     networks:
17       - web_net
18
19     volumes:
20       web_data:
21       db_data:
22
23   networks:
24     web_net:
25       driver: bridge
```

En el ejemplo podemos observar dos servicios: `web`, que despliega un contenedor `nginx` exponiendo el puerto 80, y `db`, que despliega un contenedor `postgresql`, sólo accesible desde la red que comparten ambos contenedores, `web_net`. Esta red está declarada en la sección `networks` como una red de tipo `bridge`. Por otro lado, cada contenedor tiene asociado un volumen, definido en la sección `volumes`, y referenciado en el servicio.



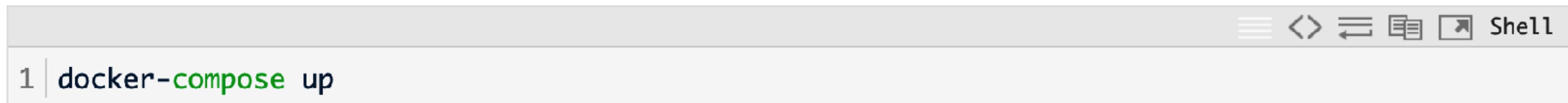
Comandos de docker-compose

Una vez definido el fichero `docker-compose.yml`, los siguientes comandos son útiles para gestionar el despliegue de nuestra aplicación y sus recursos asociados.



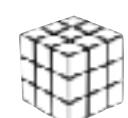
```
docker-compose up
```

Sirve para desplegar la aplicación. Si no se especifica ningún servicio, se desplegarán todos los servicios definidos en el fichero `docker-compose.yml`. Si se especifica un servicio, se desplegará sólo ese servicio y sus dependencias.



A screenshot of a terminal window titled "Shell". The window has a toolbar with icons for copy, paste, and file operations. The main area shows the command "1 | docker-compose up" entered at the prompt.

Con el parámetro `-d` el despliegue se hará en segundo plano.



`docker-compose down`

Aunque no hayamos utilizado el parámetro `-d` en el comando `docker-compose up`, para detener la ejecución de los contenedores correctamente debemos ejecutar el comando:



The image shows a screenshot of a terminal window. At the top, there is a toolbar with icons for file operations like copy, paste, and search, followed by the word "Shell". Below the toolbar, the terminal prompt shows the number "1" and the command "docker-compose down" in green text, indicating it is the current active command.

docker-compose ps

Sirve para listar los servicios desplegados, y sus contenedores asociados, y ver su estado actual junto con los puertos que tienen expuestos. Es muy útil para comprobar si el despliegue definido en el fichero `docker-compose.yml` es el esperado:

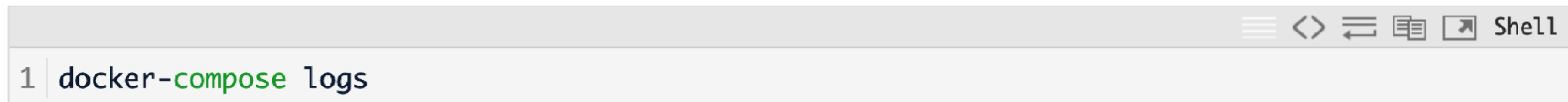


A screenshot of a terminal window titled "Shell". The window has a toolbar with icons for file operations and a tab labeled "Shell". The main area shows a command prompt with the number "1 |" followed by the command "docker-compose ps" in green text.

docker-compose logs

Sirve para ver los logs de los contenedores desplegados. Si no se especifica ningún nombre de servicio, se mostrarán los logs de todos los contenedores. Si se especifica un servicio, se mostrarán los logs de los contenedores asociados a ese servicio.

Es muy útil si un servicio ha fallado al desplegarse o simplemente para monitorizar el funcionamiento de dicho servicio.



A screenshot of a terminal window. The title bar says "Shell". The command "1 | docker-compose logs" is typed into the terminal. The terminal interface includes standard icons for file operations like copy, paste, and search.

docker-compose exec

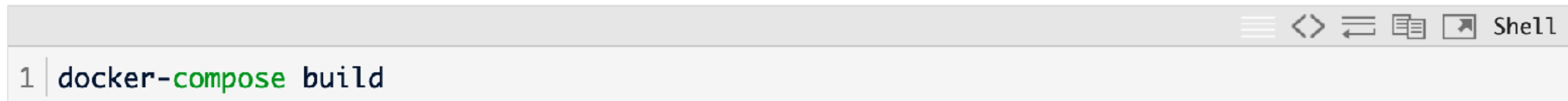
Sirve para ejecutar un comando en un contenedor desplegado. Es muy útil si queremos conectarnos al contenedor de un servicio para depurar algún tipo de errores. Por ejemplo, si queremos obtener una shell en el servicio web :



A screenshot of a terminal window titled "Shell". The window has a toolbar with icons for file operations and tabs. In the terminal area, the command "1 | docker-compose exec web bash" is typed in green, indicating it is the current command being run.

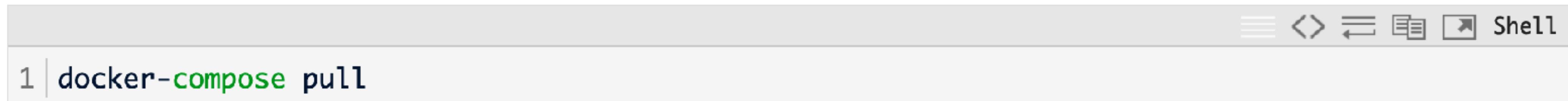
docker-compose build o docker-compose pull

`build` sirve para reconstruir las imágenes de los servicios definidos en el fichero `docker-compose.yml`. Es muy útil si hemos modificado el fichero `Dockerfile` de algún servicio y queremos reconstruir la imagen asociada a dicho servicio.



A screenshot of a terminal window titled "Shell". The window has a toolbar with icons for file operations and a tab labeled "Shell". The main area shows the command "1 | docker-compose build" entered at the prompt.

Por otro lado, si nuestras imágenes vienen de un registro, `pull` sirve para descargar las imágenes de los servicios definidos en el fichero `docker-compose.yml` desde el registro.



A screenshot of a terminal window titled "Shell". The window has a toolbar with icons for file operations and a tab labeled "Shell". The main area shows the command "1 | docker-compose pull" entered at the prompt.

DOCKER

24 lines (21 sloc) | 568 Bytes

Raw Blame   

```
1  version: "3.1"
2
3  services:
4    scheduler:
5      image: ghcr.io/dask/dask:latest
6      hostname: scheduler
7      ports:
8        - "8786:8786"
9        - "8787:8787"
10     command: ["dask-scheduler"]
11
12   worker:
13     image: ghcr.io/dask/dask:latest
14     command: ["dask-worker", "tcp://scheduler:8786"]
15     # For Docker swarm you can specify multiple workers, this is ignored by `docker-compose up`
16     deploy:
17       replicas: 2
18
19   notebook:
20     image: ghcr.io/dask/dask-notebook:latest
21     ports:
22       - "8888:8888"
23     environment:
24       - DASK_SCHEDULER_ADDRESS="tcp://scheduler:8786"
```





MUCHAS GRACIAS