Docker

David Granados Zafra

V 0.4

Docker & Big Data

Docker y Big Data

- Docker es una herramienta de contenerización que permite realizar las siguientes operaciones sobre los contenedores.
  - Empaquetar
  - Distribuir
  - Ejecutar

Docker y Big Data

- Aunque docker no está directamente relacionado con una casuística concreta como es Big Data, juega un papel decisivo en la arquitectura:
  - Desarrollo
  - Implementación
  - Gestion

Docker y Big Data

- Facilita el despliegue de las aplicaciones de Big Data
  - Encapsula todas las dependencias y configuraciones en un contenedor.
  - Esto permite mover las aplicaciones a distintos entornos.
  - Está totalmente vinculado con los sistemas cloud como AWS

Docker y Big Data

- Nos aumenta la consistencia del entorno de desarrollo.
  - Proporciona entornos de de desarrollo consistentes y reproducibles.
  - Elimina las diferencias entre los entornos locales y los entornos de producción.
  - Los desarrolladores pueden crear imágenes de contenedor que contengan todas las bibliotecas, herramientas y dependencias necesitarias.

Docker y Big Data

- Aislamiento y seguridad.
  - Docker utiliza tencnologías de virtualización a nivel de sistema operativo para proporcionar aislamiento seguro entre los contenedores.
  - Las aplicaciones Big Data pueden ejecutarse de manera segura y aislada sin interferir ni afectar a sistemas de otros host.

Docker y Big Data

- Gestión de recursos y Escalabilidad.
  - Docker permitela configuración dinámica de recursos, tales como CPU y memoria para cada uno de los contenedores.
  - Existen otras herramientas específicas como son Kubernetes y Docker Swarm que nos permiten orquestar y gestionar clústeres de contenedores.

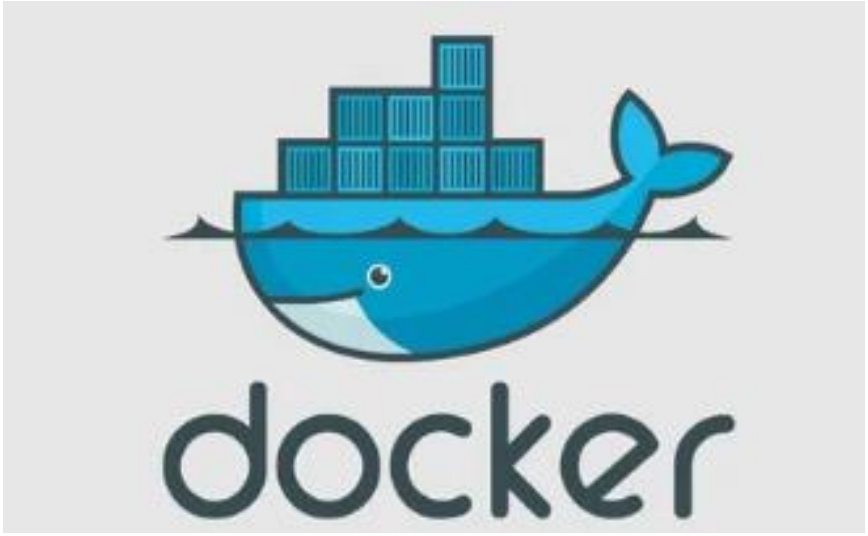
Docker y Big Data

- Integración con ecosistemas de big data:
  - Muchas tecnologías de big data como Hadoop, Spark, Kafka, HBASE,… están disponibles como imágenes de contenedor en Docker Hub

Docker

Build, Ship and Run, any App, Anywhere

Docker



Introducción a Docker.

- Es open source
- Es un proceso aislado ya que corre su propio:
  - Sistema de ficheros
  - Procesos
  - Interfaces de red
  - ...
- Es una plataforma que proporciona software en contenedores usando virtualización a nivel de sistema operativo
- Usamos el termino de empaquetar cuando juntamos todo lo necesario para que la aplicación funcione como una unidad atómica.
- Este empaquetado se hace de forma independiente de la infraestructura subyacente.

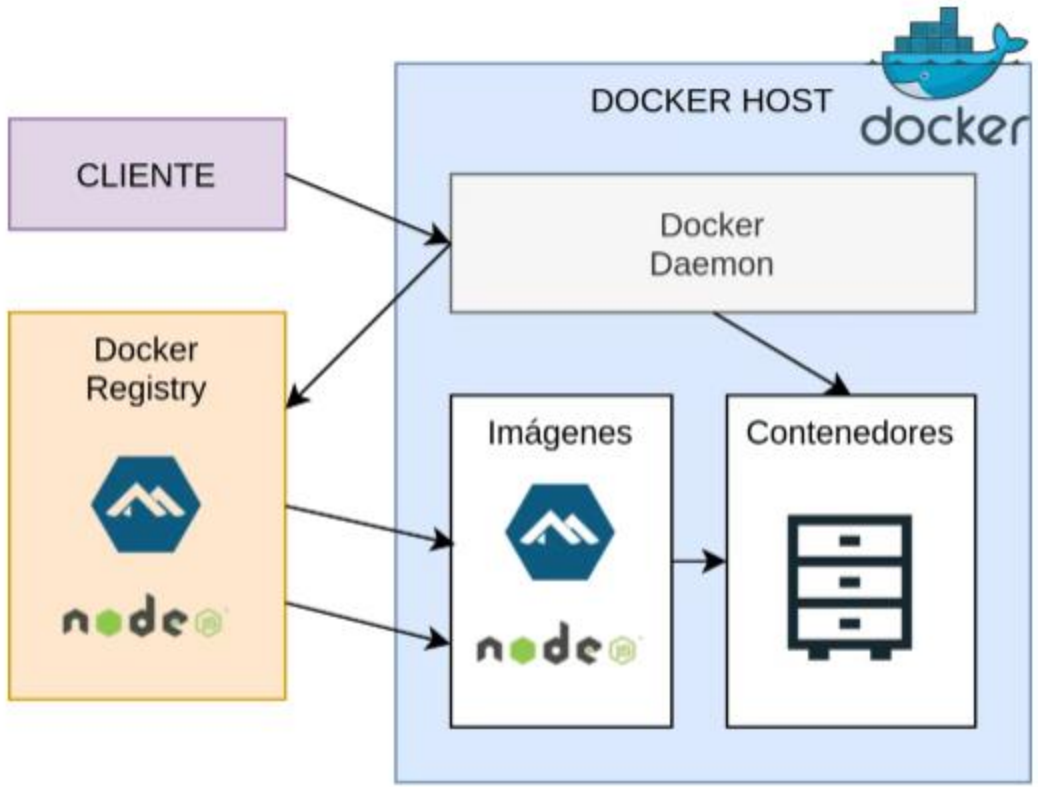
Arquitectura

- Tiene una arquitectura cliente-servidor. El cliente se comunica con Docker Daemon, que es la pieza con la responsabilidad de construir, deplegar y distribuir los contendores.
- Ambos componentes se pueden ejecutar en el mismo sistema o de forma remota (no tienen porqué estar en la misma máquina).
- El cliente es la interfaz:
  - Bien la interfaz Gráfica
  - Bien la consola.

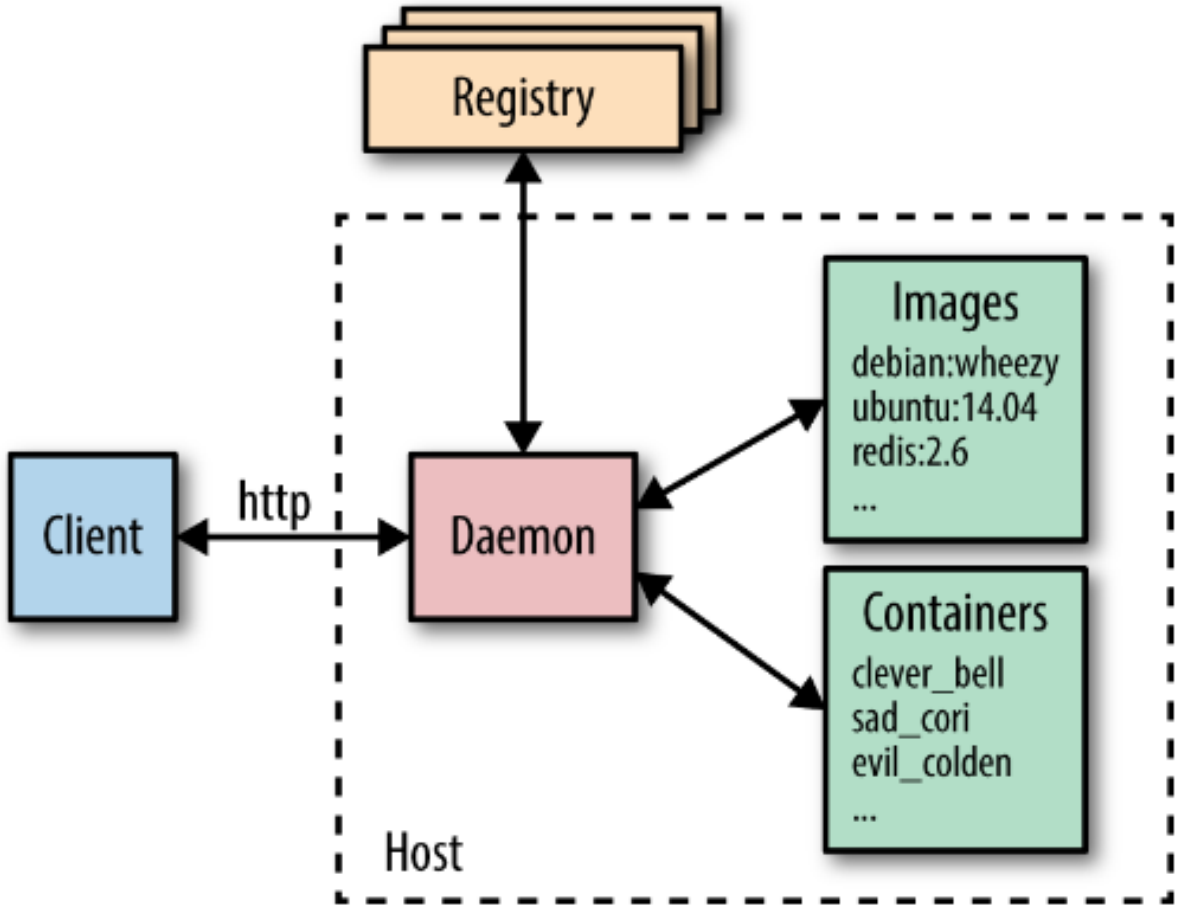
Componentes

- Es una arquitectura muy modular
- Tiene muchísimos componentes, pero a nivel de desarrollo no hace falta entrar en tanto detalle.
- Demonio levanta una API para manipular.

Arquitectura



Arquitectura



Vocabulario básico

- Contenedor.
  - Es una unidad estandarizada de software que incluye todo lo necesario para ejecutar una aplicación:
    - Bibliotecas
    - Herramientas del sistema
    - Código
    - Dependencias
  - Es un entorno aislado y portatil que se ejecuta en el sistema operativo pero no depende de el.

Vocabulario básico

- Imagen.
  - Es una paquete de solo lectura con todo lo necesario para poder crear un contenedor, podríamos pensar que es una plantilla para crear contenedores.
  - Un contenedor es por tanto una instancia en ejecución de una imagen.

Vocabulario básico

- La diferencia entre contenedor e imagen es que un contenedor es una imagen en ejecución con un estado e identificador único.
- Un contenedor puede tener cambios en sus datos o configuración los cuales son específicos del contenedor y por tanto, nunca se verán en la imágen.
- Una imagen puede usarse para crear varios contenedores, bien de forma simultánea o independiente.

Vocabulario básico

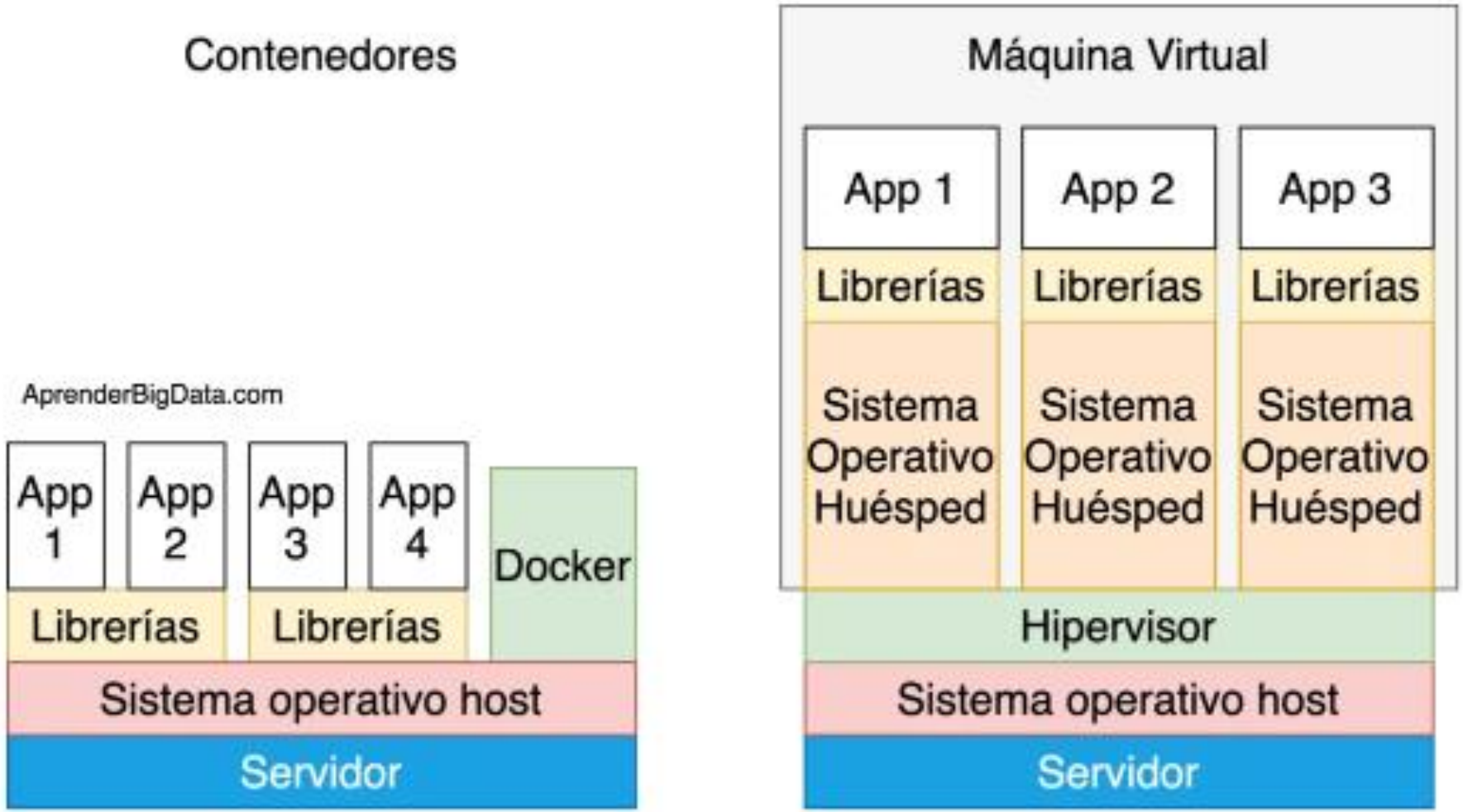
- Dockerfile.
  - Es un archivo de texto que contiene instrucciones para construir una imagen de docker de forma automatizada a lo que el VagrantFile era para crear las máquinas en Vagrant.

Vocabulario básico

- Registro (Hub).
  - Es un repositorio en línea donde se almacenan y distribuyen las imágenes. Hay registros públicos como Docker Hub y registros privados en los que solo podemos acceder de forma autorizada.

Docker

Máquinas Virtuales



Esquema de componentes en Docker y en Máquinas Virtuales

Virtualización. Máquinas virtuales.

- Las VMs emulan HW físico, lo que permite ejecutar múltiples sistemas operativos en un único servidor físico.
- Hipervisor. Software que permite la creación y gestión de las VMs
- Las VM consumen muchos más recursos de la máquina donde se alojan debido a que hacen una virtualización completa del sistema.
- Las VMs suelen ser más grandes en tamaño y tener tiempos de arranque más largos
- Todo lo anterior es lógico porque lo que estamos haciendo es una emulación de una máquina real, pero sin tener hadrware dedicado, sino que es hardware compartido.

Virtualización. Docker

- Virtualización a nivel de Sistema Operativo
  - Los contenedores comparten el Kernel del sistema operativo subyacente, lo que los hace mas livianos y eficientes.
  - Utiliza recursos del sistema de forma eficiente.
    - Los contenedores comparten recursos con el sistema anfitrión.
  - Tiempo de arranque rápido:
    - Se inician más rápidamente debido a que son mas livianos.
  - Mayor portabilidad y escalabilidad.
    - Son altamente portátiles y escalables, lo que faciliza su implementación en distintos entornos.

Docker

Inmutabilidad

Capas

- Docker implementa una arquitectura en capa que tiene ciertas características:
  - Cada capa representa un cambio incremental en la imagen base.
  - Las capas permiten la reutilización de imágenes así como creación y descarga más rápida, ya que únicamente se baja o crea aquellas capas que han cambiado.

Inmutabilidad

- Una vez creada una imagen de docker es inmutable y no se puede cambiar.
- La inmutabilidad garantiza que los contenedores creados a partir de una imagen siempre sean consistentes y reproducibles, independiente del entorno de implementación.
- Nos facilita el control de versiones ya que las imágenes inmutables perimten la implementación de cambios controlados en la infraestructura de la aplicación.

Capas Avanzado.

- Docker tiene dos tipos de capas: de lectura y de escritura
- Cuando creamos un contenedor a partir de una imagen, copiamos las capas necesarias en modo solo lectura en la caché de docker
- Se crean N contenedores y comparten esas capas de lectura. Las capas propias del contenedor, son capas de escritura y esas son únicas de cada contenedor.

Capas avanzado

- Cuando eliminamos una imagen nos exige que eliminemos los contenedores, ya que es una buena práctica.
- Podemos forzar a eliminar la imagen manteniendo los contenedores, en este caso se denominan contenedores huérfanos, lo que no es bueno ya que:
  - No podremos volver a crear el contenedor en otro lado hasta que no tengamos la imagen.
  - Si descargamos una imagen con versión mas nueva, los contenedores no lo sabrán y tendrán las vulnerabilidades de la versión anterior.
  - Se dificulta la gestión que se hace a partir de la imagen.

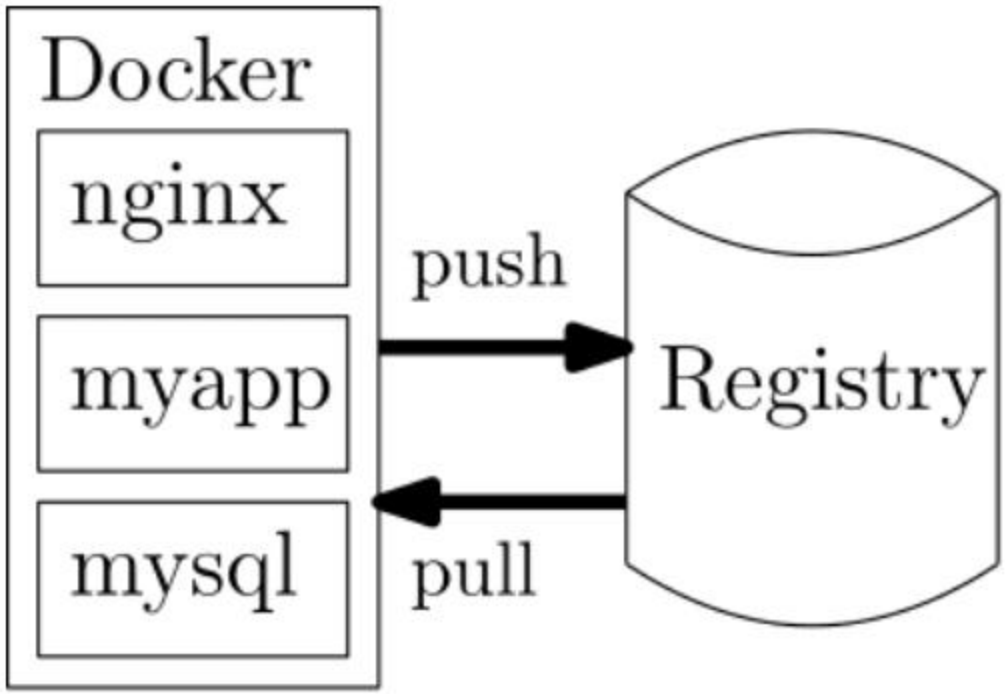
Capas Avanzado

- Los contenedores siguen funcionando sin la imagen porque la referencia no es hacia la imagen sino hacia la caché de docker.
- Para eliminar la caché usamos docker system prune.
- Si hacemos eso veremos que elimina los contenedores huérfanos.

Docker

Ciclo de desarrollo

Ciclo de desarrollo usual



Docker

Instalación

Instalación

- Podemos seguir los pasos que aparecen en:
  - https://docs.docker.com/engine/install/

Docker

Imágenes

La Caché de Docker

- Es un mecanismo que permite reutilizar las capas de las imágenes de docker que se han construido previamente.
- Cada comando que se ejecuta en un dockerfile crea una nueva capa que contiene los cambios respecto a la capa anterior.
- Docker almacena estas capas en una caché local y las reutiliza.
- Almacenamiento en caché de Docker: introducción a las capas de Docker (ichi.pro)

Docker Build y Dockerfile

```
FROM ubuntu:latest
RUN apt-get update -y
RUN apt-get install -y python-pip python-dev
WORKDIR /app
ENV DEBUG=True
EXPOSE 80
VOLUME /data
COPY . /app
RUN pip install -r requirements.txt
ENTRYPOINT ["python"]
CMD ["app.py"]
```

Caché

```
➜  docker-caching-blog git:(master) ✗ docker build -t hello-world-react-docker:latest ./
Sending build context to Docker daemon  716.8kB
Step 1/7 : FROM node:12.13.1-buster-slim AS dev
12.13.1-buster-slim: Pulling from library/node
000eee12ec04: Already exists
d8fdfd9a7a73: Already exists
7f1fa99d89fa: Already exists
019be8afd290: Already exists
fc4a5846a37d: Already exists
Digest: sha256:1d1028c639a31211a16ca39832f388375d933e913141b76f7f42ada3cf9f4b8f
Status: Downloaded newer image for node:12.13.1-buster-slim
 ---> c3dc5946eb6f
Step 2/7 : WORKDIR /home/app
 ---> Running in 9dc5abb2bc36
Removing intermediate container 9dc5abb2bc36
 ---> a6b034f19bd2
Step 3/7 : COPY ./package.json ./package-lock.json* ./
 ---> f48ff16b7965
Step 4/7 : RUN npm install
 ---> Running in 66840ccfa916

> core-js@2.6.11 postinstall /home/app/node_modules/babel-runtime/node_modules/core-js
> node -e "try{require('./postinstall')}catch(e){}"

Thank you for using core-js ( https://github.com/zloirock/core-js ) for polyfilling JavaScript standard library

The project needs your help! Please consider supporting of core-js on Open Collective or Patreon:
```

Caché

```
→  docker-caching-blog git:(master) ✗ docker build -t hello-world-react-docker:latest ./
Sending build context to Docker daemon  716.8kB
Step 1/7 : FROM node:12.13.1-buster-slim AS dev
 ---> c3dc5946eb6f
Step 2/7 : WORKDIR /home/app
 ---> Using cache
 ---> a6b034f19bd2
Step 3/7 : COPY ./package.json ./package-lock.json* ./
 ---> Using cache
 ---> f48ff16b7965
Step 4/7 : RUN npm install
 ---> Using cache
 ---> 3b8e9d61a414
Step 5/7 : COPY public /home/app/public
 ---> Using cache
 ---> bf158f61c0e1
Step 6/7 : COPY src /home/app/src
 ---> Using cache
 ---> b6be0f20bedd
Step 7/7 : CMD ["npm", "start"]
 ---> Using cache
 ---> 4c2e20f75095
Successfully built 4c2e20f75095
Successfully tagged hello-world-react-docker:latest
→  docker-caching-blog git:(master) ✗
```

Caché

```
→  docker-caching-blog git:(master) ✗ docker build -t hello-world-react-docker:latest ./
Sending build context to Docker daemon  716.8kB
Step 1/7 : FROM node:12.13.1-buster-slim AS dev
 ---> c3dc5946eb6f
Step 2/7 : WORKDIR /home/app
 ---> Using cache
 ---> a6b034f19bd2
Step 3/7 : COPY ./package.json ./package-lock.json* ./
 ---> Using cache
 ---> f48ff16b7965
Step 4/7 : RUN npm install
 ---> Using cache
 ---> 3b8e9d61a414
Step 5/7 : COPY public /home/app/public
 ---> Using cache
 ---> bf158f61c0e1
Step 6/7 : COPY src /home/app/src
 ---> 9d9d9024438c
Step 7/7 : CMD ["npm", "start"]
 ---> Running in 682b7a3f8cf3
Removing intermediate container 682b7a3f8cf3
 ---> d74f3d5b06c4
Successfully built d74f3d5b06c4
Successfully tagged hello-world-react-docker:latest
→  docker-caching-blog git:(master) ✗
```

Ejercicio1

- En este ejercicio vamos a Practicar algunos comandos:
  - docker pull nginx -> Se descargará la imagen de DockerHub
  - docker images -> Nos mostrará las imágenes que tenemos instaladas.
  - docker run -d -p 8080:80 --name mi-nginx nginx -> Lanzamos un contenedor.
  - docker ps -> Verificar el estado de los contenedores.
  - docker stop mi-nginx -> Paramos el contenedor
  - docker rm mi-nginx -> Eliminamos el contenedor.

DockerFile

Imágenes

Introducción a Dockerfile.

- Definición: Un Dockerfile es un archivo de texto que contiene una serie de instrucciones para construir una imagen Docker.
- Propósito: Automatizar el proceso de creación de imágenes Docker.
- Ubicación: Generalmente se coloca en la raíz del proyecto.

Estructura básica de un dockerfile

```
FROM ubuntu:20.04
RUN apt-get update && apt-get install -y python3
COPY . /app
WORKDIR /app
CMD ["python3", "app.py"]
```

Instrucciones importantes de dockerfile

- **FROM**: especifica la imagen base a partir de la cual se construye la nueva imagen. Es la primera instrucción que debe aparecer en un docker file. Por ejemplo: FROM ubuntu
- **RUN**: ejecuta un comando y sus argumentos dentro de la imagen. Se puede usar para instalar paquetes, crear directorios, modificar archivos, etc. Por ejemplo: RUN apt-get update && apt-get install -y python
- **CMD**: define el comando y los argumentos que se ejecutarán cuando se inicie el contenedor a partir de la imagen. Solo puede haber una instrucción CMD por docker file. Por ejemplo: CMD ["python", "app.py"]
- **ENTRYPOINT**: define el comando y los argumentos que se ejecutarán como el programa principal del contenedor. Se puede combinar con CMD para proporcionar argumentos por defecto. Por ejemplo: ENTRYPOINT ["ping"] CMD ["localhost"]

Instrucciones importantes dockerfile

- **COPY**: copia archivos o directorios desde el contexto de construcción (el directorio donde se encuentra el docker file) al sistema de archivos del contenedor. Por ejemplo: COPY app.py /app/
- **ADD**: similar a COPY, pero también puede copiar archivos desde una URL o descomprimir archivos comprimidos. Por ejemplo: ADD https://example.com/file.tar.gz /app/
- **ENV**: establece variables de entorno dentro de la imagen. Se pueden usar en otras instrucciones o en el contenedor. Por ejemplo: ENV APP_NAME my_app
- **EXPOSE**: indica el puerto o los puertos en los que el contenedor escucha conexiones. No publica el puerto, solo sirve como documentación. Por ejemplo: EXPOSE 80

Instrucciones Importantes dockerfile

- **WORKDIR**: establece el directorio de trabajo para las instrucciones RUN, CMD, ENTRYPOINT, COPY y ADD. Si el directorio no existe, se crea. Por ejemplo: WORKDIR /app
- **VOLUME**: crea un punto de montaje para un volumen externo o anónimo. Se puede usar para persistir datos o compartirlos entre contenedores. Por ejemplo: VOLUME /data
- **LABEL**: agrega metadatos a la imagen en forma de pares clave-valor. Se pueden usar para proporcionar información adicional sobre la imagen. Por ejemplo: LABEL version="1.0" description="This is a custom image"

Ejemplo de docker file para flask.

```
FROM python:3.9
COPY . /app
WORKDIR /app
RUN pip install -r requirements.txt
EXPOSE 3000
CMD ["flask", "run", "--host=0.0.0.0", "--port=3000"]
```

Otro Ejemplo

```
FROM python:3.9-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY . .
EXPOSE 5000
ENV NAME World
CMD ["python", "app.py"]
```

Buenas Prácticas

- Usa imágenes base oficiales y específicas
- Minimiza el número de capas
- Agrupa comandos RUN usando &&
- Usa .dockerignore para excluir archivos innecesarios
- Evita instalar paquetes innecesarios
- Usa COPY en lugar de ADD cuando sea posible
- Especifica versiones exactas de las dependencias

ARG vs ENV

- ARG:
- Se usa durante la construcción de la imagen.
- No está disponible en el contenedor en tiempo de ejecución.
- Útil para valores que pueden cambiar durante la construcción.
- ENV:
- Se usa tanto durante la construcción como en tiempo de ejecución.
- Persiste en el contenedor y puede ser usado por las aplicaciones.
- Ejemplo:

```
ARG VERSION=latest
FROM ubuntu:${VERSION}
ENV APP_VERSION=${VERSION}
```

Entrypoint vs CMD

- ENTRYPOINT:
  - Define el ejecutable principal del contenedor.
  - Es más difícil de sobrescribir (requiere --entrypoint en docker run).
  - Útil para crear contenedores que actúen como ejecutables.
- CMD:
  - Proporciona argumentos por defecto para el ENTRYPOINT o especifica el comando a ejecutar si no hay ENTRYPOINT.
  - Fácilmente sobrescribible en docker run.

- Ejemplo:

```
ENTRYPOINT ["python3"]
CMD ["app.py"]
```