**Qué es docker?**  
Docker es un programa de código abierto que permite que una aplicación Linux y sus dependencias se empaqueten como un contenedor.

La virtualización basada en contenedores aísla las aplicaciones entre sí en un sistema operativo (OS) compartido. Este enfoque estandariza la entrega del programa de la aplicación, permitiendo que las aplicaciones se ejecuten en cualquier entorno Linux, ya sea físico o virtual. Dado que comparten el mismo sistema operativo, los contenedores son portátiles entre diferentes distribuciones de Linux, y son significativamente más pequeños que las imágenes de máquinas virtuales (VM).

“Docker te permite construir, distribuir y ejecutar cualquier aplicación en cualquier lado.”

**Problemáticas del desarrollo de software**

**1. Construir -** Escribir código en la máquina del desarrollador. (Compile, que no compile, arreglar el bug, compartir código, etc. )

**Problemática:**

* Entorno de desarrollo (paquetes)
* Dependencias (Frameworks, bibliotecas)
* Versiones de entornos de ejecución (runtime, versión Node)
* Equivalencia de entornos de desarrollo (compartir el código)
* Equivalencia con entornos productivos (pasar a producción)
* Servicios externos (integración con otros servicios ejem: base de datos)

**2. Distribuir** - Llevar la aplicación donde se va a desplegar (Transformarse en un artefacto)

**Problemática:**

* Output de build heterogeo (múltiples compilaciones)
* Acceso a servidores productivos (No tenemos acceso al servidor)
* Ejecución nativa vs virtualizada
* Entornos Serverless

**3. Ejecutar** - Implementar la solución en el ambiente de producción (Subir a producción)  
El reto Hacer que funcione como debería funcionar

**Problemática:**

* Dependencia de aplicación (paquetes, runtime)
* Compatibilidad con el entorno productivo (sistema operativo poco amigable con la solución)
* Disponibilidad de servicios externos (Acceso a los servicios externos)
* Recursos de hardware (Capacidad de ejecución - Menos memoria, procesador más debil)

**Virtualizacion**  
Permite atacar en simultáneo los tres problemas del desarrollo de software profesional.

**Problemas de la virtualizacion**

* PESO: En el orden de los GBs. Repiten archivos en común. Inicio lento.
* COSTO DE ADMINISTRACION: Necesita mantenimiento igual que cualquier otra computadora.
* MULTIPLES DE FORMATO: VDI, VMDK, VHD, raw, etc

**Containerización**  
El empleo de contenedores para construir y desplegar software.

* Flexibles
* Livianos
* Portables
* Bajo acoplamiento
* Escalables
* Seguros

**Virtualizacion vs Containerización**

* Virtualización: A diferencia de un contenedor, las máquinas virtuales ejecutan un sistema operativo completo, incluido su propio kernel.
* Containerización: Un contenedor es un silo aislado y ligero para ejecutar una aplicación en el sistema operativo host. Los contenedores se basan en el kernel del sistema operativo host (que puede considerarse la fontanería del sistema operativo), y solo puede contener aplicaciones y algunas API ligeras del sistema operativo y servicios que se ejecutan en modo de usuario.

Componentes DENTRO del circulo de Docker:

* Docker daemon: Es el centro de docker, el corazón que gracias a el, podemos comunicarnos con los servicios de docker.
* REST API: Como cualquier otra API, es la que nos permite visualizar docker de forma “gráfica”.
* Cliente de docker: Gracias a este componente, podemos comunicarnos con el corazón de docker (Docker Daemon) que por defecto es la línea de comandos.

Dentro de la arquitectura de Docker encontramos:

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

**Que es un contenedor ?**

* Es una agrupación de procesos.
* Es una entidad lógica, no tiene el limite estricto de las máquinas virtuales, emulación del sistema operativo simulado por otra más abajo.
* Ejecuta sus procesos de forma nativa.
* Los procesos que se ejecutan adentro de los contenedores ven su universo como el contenedor lo define, no pueden ver mas allá del contenedor, a pesar de estar corriendo en una maquina más grande.
* No tienen forma de consumir más recursos que los que se les permite. Si esta restringido en memoria ram por ejemplo, es la única que pueden usar.
* A fines prácticos los podemos imaginar cómo maquinas virtuales, pero NO lo son. Máquinas virtuales livianas.
* Docker corre de forma nativa solo en Linux.
* Sector del disco: Cuando un contenedor es ejecutado, el daemon de docker le dice, a partir de acá para arriba este disco es tuyo, pero no puedes subir mas arriba.
* Docker hace que los procesos adentro de un contenedor este aislados del resto del sistema, no le permite ver más allá.
* Cada contenedor tiene un ID único, también tiene un nombre.

Comandos:  
$ docker run hello-world (corro el contenedor hello-world)  
$ docker ps (muestra los contenedores activos)  
$ docker ps -a (muestra todos los contenedores)  
$ docker inspect <containe ID> (muestra el detalle completo de un contenedor)  
$ docker inspect <name> (igual que el anterior pero invocado con el nombre)  
$ docker run –-name hello-platzi hello-world (le asigno un nombre custom “hello-platzi”)  
$ docker rename hello-platzi hola-platzy (cambio el nombre de hello-platzi a hola-platzi)  
$ docker rm <ID o nombre> (borro un contenedor)  
$ docker container prune (borro todos lo contenedores que esten parados)

* Combinación de teclas para salir de un contenedor y apagarlo  
  crtl + c
* Combinación de teclas para salir del contenedor sin apgarlo  
  ctrl + d
* Comando para salir de un contenedor y apargarlo, si el proceso es un noOps  
  $ exit
* Iniciar un contenedor ya creado  
  $ docker start <nombre-contenedor o id-contenedor>

Ejemplo  
$ docker start contenedor-ubuntu

* Detener un contenedor ya creado e iniciado  
  $ docker start <nombre-contenedor o id-contenedor>

Ejemplo  
$ docker stop contenedor-ubuntu

* Iniciar bash un contenedor ya creado  
  $ docker exec -it <nombre-contenedor o id-contenedor>
* docker exec

Ejecutar dentro del contenedor

-it

interactivo; terminal. Inicia un terminal modo interactivo

<contenedor ID OR nombre contenedor >

nombre del contenedor o el id

bash

bas o terminal

Ejemplo  
$ docker exec -it contenedor-ubuntu

Cada vez que un contendor se ejecuta, en realidad lo que ejecuta es un proceso del sistema operativo. Este proceso se le conoce como **Main process**.

**Main process**  
Determina la vida del contenedor, un contendor corre siempre y cuando su proceso principal este corriendo.

**Sub process**  
Un contenedor puede tener o lanzar procesos alternos al main process, si estos fallan el contenedor va a seguir encedido a menos que falle el main.

**Ejemplos manejados en el video**

* Batch como Main process
* Agujero negro (/dev/null) como Main process

docker run --name alwaysup -d ubuntu tail -f /dev/null

\_el ouput que te regresa es el ID del contentedor \_

Te puedes conectar al contenedor y hacer cosas dentro del él con el siguiente comando (sub proceso)

docker exec -it alwaysup bash

Se puede matar un Main process desde afuera del contenedor, esto se logra conociendo el id del proceso principal del contenedor que se tiene en la maquina. Para saberlo se ejecuta los siguientes comandos;

docker inspect --format '{{.State.Pid}}' alwaysup

\_El output del comando es el process ID (2474) \_

Para matar el proceso principal del contenedor desde afuera se ejecuta el siguiente comando (solo funciona en linux)

Kill 2474

Comandos:

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

Comandos:

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

**Crear un volumen:**  
docker volume create [nombre\_del\_volument]

**Listar los volúmenes**  
docker volume ls

**Host:** Donde Docker esta instalado.  
**Bind Mount:** Guarda los archivos en la maquina local persistiendo y visualizando estos datos (No seguro).  
**Volume:** Guarda los archivos en el area de Docker donde Docker los administra (Seguro).  
**TMPFS Mount:** Guarda los archivos temporalmente y persiste los datos en la memoria del contenedor, cuando muera sus datos mueren con el contenedor.

\*\*Montamos el contenedor especificándole el volumen \*\*  
entre la coma en src y dst no poner espacio. Puede que nos de un error docker: invalid refence format  
docker run -d --name db --mount src=dbdata (que\_queremos\_montar),dst=(destino) /data/db mongo

**Entramoss en el cliente de mongo**  
docker exec -it db bash  
mongo

**Borramo el contenedor anteriormente creado**  
docker rm -f db

**Montamos otro contenedor con el mismo nombre, mismo lugar (src,dst)**  
docker run -d --name db --mount src=dbdata,dst=/data/db mongo  
docker exec -it db bash  
mongo

Comandos:

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

**15. Insertar y extraer archivos de un contenedor**  
Existen dos formas de manejar datos con docker, un método es bind mount y otro es volumen (volume), el primero consiste en compartir un directorio de la máquina host con el contenedor y el segundo es más práctico consiste en crear un Volume y este último tiene un problema que los archivos son visibles por el contenedor y por otro volumen y no por nosotros.  
Lo siguiente que vamos a ver es que independientemente de que usemos un método u otros podremos introducir archivos o sacar archivos. Es importante resaltar que no es necesario que el contenedor este encendido cuando se intenta introducir o sacar archivos del contenedor

Pasos para copiar un archivo de la máquina host al contenedor

1. Crear un archivo para realizar prueba de copia
2. Comprobar que el archivo existe y ver que no tiene nada de contenido
3. Crear un contenedor basado en la imagen de ubuntu e iniciar este contenedor con un proceso NoOps
4. Verificar que el contenedor esté corriendo o activo.
5. Entrar al bash del contenedor
6. Crear una carpeta y comprobar que el directorio fue creado. Este directorio dentro del contenedor alojará el archivo prueba.
7. Salir del contenedor
8. Copiar el archivo local (host) al contenedor. se puede apreciar que a parte de copiar se está cambiado el nombre del archivo.
9. Entrar al bash del contenedor
10. Entrar al directorio donde se copió el archivo y comprobar 11. que existe allí
11. Salir del contenedor

Ejecución de los pasos para copiar un archivo de la máquina host al contenedor

1. $ touch prueba.txt
2. $ cat prueba.txt
3. $ sudo docker run -d --name copytest ubuntu tail -f /dev/null
4. $ sudo docker ps
5. $ sudo docker exec -it copytest bash
6. $ mkdir testing && ll
7. $ exit
8. $ sudo docker cp prueba.txt copytest:/testing/test.txt  
   9.$ sudo docker exec -it copytest bash
9. $ cd testing && ll
10. $ exit.

Pasos para copiar un archivo del contenedor al host

1. Comprobar que el contenedor existe, no importa si está encendido o apagado, ya que el proceso se ejecutará igual
2. Copiar el archivo/carpeta del contenedor al local (host), como se puede ver también se está cambiando el nombre de la carpeta al mismo tiempo que se copia
3. verificar que el archivo existe

Ejecución de los pasos para copiar un archivo del contenedor a la máquina host  
$ sudo docker ps -a  
$ sudo docker cp copytest:/testing local-testing  
$ ll

Explicación de la extracción o introducir de datos a un contenedor

Introducir archivos

sudo docker cp <path/nombre-archivo> <nombre Contenedor>:<path/nombre-archivo>

extraer archivos  
sudo docker cp <nombre Contenedor>:<path/nombre-archivo> <path/nombre-archivo>

flag

cp :  
Instrucción para Copiar

Parámetros

<nombre Contenedor>:

Nombre del contenedor

<path/nombre-archivo>

Directorio del archivo y ubicación del archivo

**16. Conceptos fundamentales de Docker: imágenes**  
¿Qué son las imágenes?  
Son moldes o plantillas que tienen empaquetado todo lo necesario para funcionar. Docker usa estás misma para construir contenedores.  
¿Para qué sirven las imágenes en la construcción de software?  
Las imágenes sirven para crear contenedores y también es como docker intenta solucionar el problema de construcción de y distribución de software.

**$ sudo docker image ls**  
Listar imágenes locales de docker. Esta instrucción de terminal mostrará las siguientes columnas

* TAG:  
  Es la version de la imagen, cuando no especificamos que versión de descargar docker asume que es la última
* CREATE AT:  
  tiempo de creación
* SIZE:  
  Tamaño de la imagen
* IMAGE ID:  
  Este es el identificador de la imagen. Si descargamos dos imágenes que son iguales al realizar un pull, docker lo que hará es tener un puntero lógico a la misma imagen ya descargada y no descargará la otra imagen. Existe la posibilidad que tenga TAG diferentes, pero serán la misma imagen. Esto último se puede comprobar viendo el IMAGE ID, debería ser los mismos.

**$ sudo docker pull**  
Si no colocamos un parámetro después de pull entenderá que deberá realizar la descarga de docker hub. EL parámetro extra después de pull es para indicarle el repositorio

Comandos: IMAGENES

$ mkdir imagenes (creo un directorio en mi máquina)  
$ cd imagenes (entro al directorio)  
$ touch Dockerfile (creo un Dockerfile)  
$ code . (abro code en el direcotrio en el que estoy)

##Contenido del Dockerfile##  
FROM ubuntu:latest  
RUN touch /ust/src/hola-platzi.txt (comando a ejecutar en tiempo de build)  
##fin##

$ docker build -t ubuntu:platzi . (creo una imagen con el contexto de build <directorio>)  
$ docker run -it ubuntu:platzi (corro el contenedor con la nueva imagen)  
$ docker login (me logueo en docker hub)  
$ docker tag ubuntu:platzi miusuario/ubuntu:platzy (cambio el tag para poder subirla a mi docker hub)  
$ docker push miusuario/ubuntu:platzi (publico la imagen a mi docker hub)

La importancia de entender el sistema de capas consiste en la optimización de la construcción del contenedor para reducir espacio ya que cada comando en el dockerfile crea una capa extra de código en la imagen.

Agregando Capas

Con docker commit se crea una nueva imagen con una capa adicional que modifica la capa base.

Ejemplo: crear una nueva imagen a partir de la imagen de Ubuntu.

docker pull ubuntu

docker images

docker run -it cf0f3ca922e0 bin/bash

(modificar el contenedor: Ej apt-get install nmap)

docker commit deddd39fa163 ubuntu-nmap

Comandos:

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

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

La clave está en estructurar nuestro Dockerfile de manera de que primero se copien todas las dependencias y posteriormente nuestro código fuente, que es el mas suceptible a cambios.

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

**Éxito total éxito roootundo!!!**  
Comandos:

➜ docker network ls  
Permite que otros contenedores se conecten al network  
➜ docker network create --attachable platzinet  
Inspeciona el network creado  
➜ docker network inspect platzinet  
Crea contenedor de mongo  
➜ docker run -d --name db mongo:4  
Conectar el contenedor db a la red platzinet  
➜ docker network connect platzinet db  
Contenedor usando la imagen platziapp, ademas conecta con el contenedor de mongo base de datos test, con una variable de entorno  
➜ docker run -d --name app -p 3000:3000 --env MONGO\_URL=mongodb://db:27017/test platziapp  
Conectar el contenedor app a la red platzinet  
➜ docker network connect platzinet app

Docker Compose es una herramienta que permite simplificar el uso de Docker. A partir de archivos YAML es mas sencillo crear contendores, conectarlos, habilitar puertos, volumenes, etc.

Con Compose puedes crear diferentes contenedores y al mismo tiempo, en cada contenedor, diferentes servicios, unirlos a un volúmen común, iniciarlos y apagarlos, etc. Es un componente fundamental para poder construir aplicaciones y microservicios.

Docker Compose te permite mediante archivos YAML, poder instruir al Docker Engine a realizar tareas, programáticamente. Y esta es la clave, la facilidad para dar una serie de instrucciones, y luego repetirlas en diferentes ambientes.

**Docker compose** es una herramienta que nos permite describir de forma declarativa la arquitectura de nuestra aplicación, utiliza compose file (docker-compose.yml).

Comandos:  
$ docker network ls (listo las redes)  
$ docker network inspect docker\_default (veo la definición de la red)  
$ docker-compose logs (veo todos los logs)  
$ docker-compose logs app (solo veo el log de “app”)  
$ docker-compose logs -f app (hago un follow del log de app)  
$ docker-compose exec app bash (entro al shell del contenedor app)  
$ docker-compose ps (veo los contenedores generados por docker compose)  
$ docker-compose down (borro todo lo generado por docker compose)

Los comandos docker-compose solo funcionan si se ejecutan dentro de la carpeta que contiene la definicion docker-compose.yml, si tratas de ejecutarlo desde afuera no lo reconoce.

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

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

Comandos:  
$ docker-compose build (crea las imágenes)  
$ docker-compose up -d (crea los servicios/contenedores)  
$ docker-compose logs app (veo los logs de “app”)  
$ docker-compose logs -f app (hago un follow de los logs de “app”)

**docker-compose.override.yml** es un archivo que se encarga de sobreescribir tu configuración de **docker-compose.yml** , se puede usar para tener segura tu configuración y para no guardar los cambios en el repositorio de git.  
Un equivalente podría ser los archivos de declaración de variables de entorno, donde hay un archivo **.env** declarando su nombre y valor, y hay una copia **.env.example** con solo las variables sin valor. En **.gitignore** se declara que los cambios en **.env** no serán guardados, pero mandamos el archivo de ejemplo al repositorio.

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

**Administrando tu ambiente de Docker**  
La idea administrar el ambiente de docker como programador es estar conciente de los recursos asignados en nuestro entorno y llevar un seguimiento de estos porque puede ocurrir que en algún momento nos quedemos sin recursos.

Listar recursos y contenedores en docker

* $ sudo docker ps -a  
  Lista y muestra todos los contenedores tantos inactivos o inactivos
* $ sudo docker ps -aq  
  lista y muestra todos los contenedores tantos inactivos o inactivos
* $ sudo docker network ls  
  Lista y muestra todas las redes
* $ sudo docker volume ls  
  Lista y muestra todos los volumenes
* $ sudo docker image ls  
  Lista y muestra todas las imágenes

Liberar o eliminar recursos de docker

* $ sudo docker system prune  
  Delete todo lo que no se esté usando
* $ sudo docker image prune  
  Elimina las imágenes
* $ sudo docker container prune  
  Elimina los contenedores inactivos
* $ sudo docker network prune  
  Elimina los redes inactivos
* $ sudo docker volume prune  
  Elimina los volume que se usan
* $ sudo docker rm -f $(docker ps -aq)  
  Elimina todos los contenedores que estén activos o inactivos

Mostrar estadísticas y asignar recursos

* $ sudo docker run -d --name container\_platzi\_app --memory 1g image\_platzi\_app

–name <nombre del container>  
nombre del contenedor  
–memory <cantidad de memoria a usar>  
Limito el uso de memoria

* $ sudo docker stats  
  Muestra el monitor de recursos de los contenedores
* $ sudo docker inspect container\_platzi\_app  
  Inspecciona un contenedor

**SIGKILL**

* SIGKILL es la mejor forma de matar un proceso. Siempre matará un proceso y lo matará abruptamente, generando un error fatal. SIGKILL siempre debería funcionar. Si no funciona, el sistema operativo ha fallado.  
  `

**SIGTERM:**

* SIGTERM intenta matar un proceso, pero a diferencia de SIGKILL, puede bloquearse o manejarse de otra manera. Puede considerarse una forma más suave de intentar finalizar un proceso.  
  Para la mayoría de los propósitos, SIGKILL será el método más rápido y efectivo para terminar el proceso.

Al usar cmd "/loop.sh" se ejecuta el comando en "shell form". De esta forma, Docker utiliza el shell que se encuentre configurado en el dockerfile y ejecuta el comando como un subproceso del shell. Es así que el shell queda como proceso principal y el comando como subproceso. Al usar cmd ["/loop.sh"] se ejecuta el comando en "exec form". De esta forma, Docker lo ejecuta como proceso principal.

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

El **ENTRYPOINT** es el comando que se va a ejecutar por defecto, y en **CMD** va el parametro del comando.  
Este parametro se puede modificar desde el run del contenedor.

Comandos:  
$ docker buils -t ping . (construyo la imagen)  
$ docker run --name pinger ping <hostname> (ahora le puedo pasar un parámetro, previamente tengo que agregar el ENTRYPOINT en el Dockerfile)

**.dockerignore file deberia ser in un subset de .gitignore file?**

La respuesta es: *no deberia*

Porque el archivo .gitignore es para ignorar **archivos locales** en tu entorno de desarrollo.  
y el build de una imagen no deberia hacerse de forma local, el build debe ser hecho en **el pipeline** de tu repositorio de forma automatica, asi que por ende los archivos locales que deseas ignorar no deberian estar en tu repo

Al igual que **.gitignore** en git, en Docker existe el **.dockerignore** que lista archivos o carpetas que se deben ignorar al momento de tomar en cuenta el contexto de build para construir una imagen.

Comandos:  
$ docker build -t prueba .(creo la imagen)  
$ docker run -d --rm --name app prueba (corro el contenedor)  
en el archivo .dockerignore puedo poner todo lo que no quiero que copie del contexto de build  
$ docker exec -it app bash (entro al contenedor y verifico que no se haya copiado lo que está en el .dockerignore)

* El Dockerfile de producción contiene 2 “fases de build” que se pueden pensar como hacer 2 build seguidos, en donde al final la imagen construida contendrá lo especificado en el ultimo de los build.
* El primer build corre 1 test que verifica que todo funcione bien  
  El segundo build construye la imagen final aprovechando el caché de las capas del primer build.
* Al final el 2do build es solo una extracción de lo que nos interza del primer build.
* Lo importante en este caso especifico es que si el test falla, entonces el build 2 no se corre, lo que significa que la imagen no se construye.

Comandos:  
$ docker build -t prodapp -f Dockerfile . (ahora le especifíco el Dockerfile)  
$ docker run -d --name prod prodapp

**DOCKER-IN-DOCKER**

Existe la Posibilidad de usar Docker desde otros contenedores, se logra usando el Docker socket con bind mount se accede a el archivo docker sock a la maquina anfitriona y accediendo a el desde el otro Docker el cliente puede accederlo puede hablarle directamente