# ¿Qué es Docker?

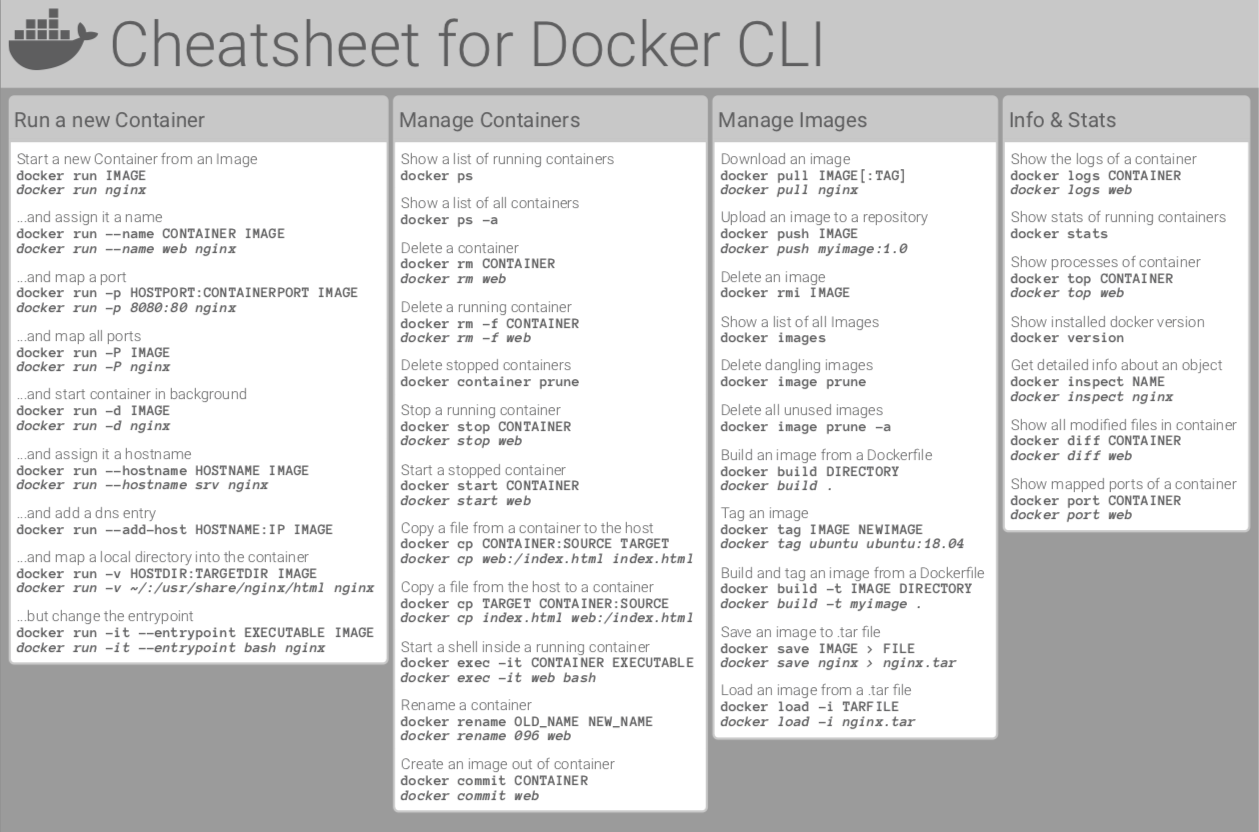
Docker es una plataforma de software que nos permite crear, probar e implementar aplicaciones rápidamente.

Durante el desarrollo de una aplicación se requiere librerías, variables de entorno y otras dependencias que complementan y apoyan su funcionamiento.

Cuando una aplicación es alojada en el servidor donde va a vivir para ser accedida por sus usuarios, es importante que todas estas cosas existan. Docker nos ayuda a que esas necesidades se cumplan.

# ¿Por qué Docker?

Docker nos facilita a que las aplicaciones cuenten con sus recursos necesarios dentro de sus aplicaciones y que este de manera aislada de otros contenedores, esto nos permite resolver los grandes problemas del desarrollo de software muy fácilmente, además de que nos ayuda a trabajar más rápido.



# Las tres áreas en el desarrollo de software profesional

**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 heterogéneo (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)

# Virtualización

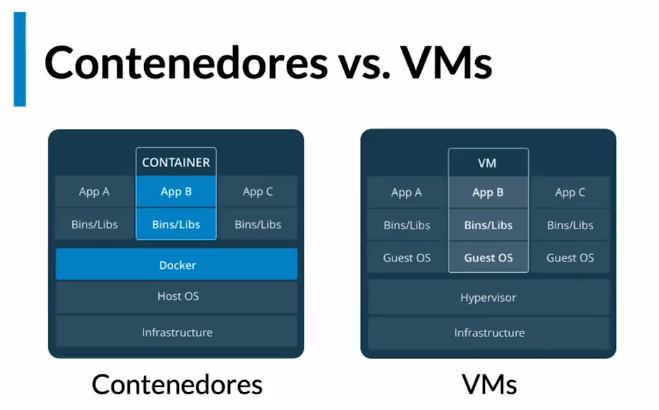
Permite atacar en simultáneo los tres problemas del desarrollo de software profesional.

**Problemas de la visualización**

* Peso: En el orden de los GBs. Repiten archivos en común. Inicio lento.
* Costo de administración: Necesita mantenimiento igual que cualquier otra computadora.
* Múltiples formatos: VDI, VMDK, VHD, raw, etc.

**Containerización**

El empleo de contenedores para construir y desplegar software.

* Flexibles: No importa que app se quiera meter en un contenedor, se puede, es decir, prácticamente toda app se puede dockerizar.
* Livianos: Los contenedores reutilizan el kernel de la maquina anfitriona, es decir, la parte donde corren los contenedores. Todos los contenedores van a reutilizar un motón de código que ya esta disponible en el SO.
* Portables: Están diseñado para correr en cualquier maquina.
* Bajo acoplamiento: ES decir, con el SO que esta corriendo. Cada contenedor tiene lo necesario para poder correr lo que necesita.
* Escalables: Es muy fácil crear otro contenedor igual al que ya tengo corriendo con la garantia que el código va a ser igual, va a funcionar igual.
* Seguros: Las herramientas que corren contenedores se aseguran que solamente pueda acceder aquellas partes que necesita acceder y que no pueda acceder a otras.

**Visualización 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.

# Qué es y cómo funciona Docker

Docker es una plataforma que permite **construir**, **ejecutar** y **compartir** aplicaciones mediante contenedores.

**Docker daemon**

Es un servicio del sistema operativo que maneja todas las entidades con las que trabaja Docker.

**Arquitectura de Docker**

* El corazón de Docker es **docker daemon**
  + El que estará trabajando con el sistema operativo para que Docker funcione.
* La forma en la que docker daemon expone una interfaz para comunicarse consigo es con una REST API.
* Encima de la REST API, esta **docker-cli** que es el cliente.
  + El **cliente** le habla al **docker daemon** con las instrucciones que nosotros escribimos en el cliente.
* El cliente podría ser otro aparte de nosotros, como ser un cliente que deseemos que hable con el docker daemon
  + Ej: Un paquete de Python para que hable al docker daemon a través del cliente.
* Principalmente trabajaremos con 4 entidades
  + **Contenedores**
  + Es el corazón del Docker y lo que utilizaremos todo el tiempo.
  + **Imágenes**
  + Artefactos que utiliza Docker para empaquetar contenedores y para poderlos distribuirlos a través de distintas instalaciones de Docker.
  + **Volúmenes de datos**
  + Es la manera en la que Docker nos permite acceder con seguridad y de manera flexible al sistema de archivos de la máquina anfitriona sin comprometer la seguridad del mismo.
  + **Network**
  + Permite a los contenedores comunicarse entre si o con el mundo exterior.

# Conceptos fundamentales de Docker: 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.

# Ciclo de vida de un contenedores

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

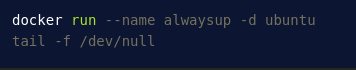
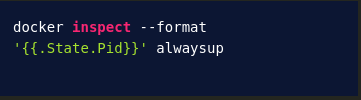
**Main process**

Determina la vida del contenedor, un contenedor 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 encendido a menos que falle el main.

Ejemplo

* Batch como **Main process**
* Agujero negro (/dev/null) como Main process
* El ouput que te regresa es el ID del contenedor.
* Te puedes conectar al contenedor y hacer cosas dentro del él con el siguiente comando (**sub proceso**)
* 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.
* 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)

En resumen, todo contenedor va a correr al menos un proceso y ese proceso se llama **Proceso Principal** o **Main Process**, y siempre desde el punto de vista del contenedor, ese proceso va a tener el **process\_id 1**, independientemente que dentro del contenedor haya otros process o no, el contenedor va a seguir vivo y funcionando, siempre y cuando el **Main Process** este funcionando. Si ese Main Process es un proceso nativo de la máquina que corre docker, se muere, el contenedor, también se morirá, es decir, parará de ejecutarse.

# Bind mounts

**Bind mounts** es ligar una ruta dentro de la máquina que ejecuta el contenedor con una ruta dentro del contenedor, esto permitirá espejar ese contenido, es decir, todo lo que se guarde en el contenedor, también se guardará en el disco de la máquina anfitriona.

Para hacer esto, se debe obtener la ruta completa o absoluta del path en cuestión.

**Ventajas**

* Es muy práctico, nos permite meter o sacar cosas a un contenedor.

**Desventajas**

* A un contenedor se le da acceso a una parte del disco de la máquina anfitriona, lo cual el contenedor va a tener acceso total a esa parte del disco y si se ejecutan contenedores de dudosa procedencia, esto puede ser un problema.

**Volúmenes**

**Vincular montaje y volúmenes**

Cuando usamos un contenedor, los datos generados se eliminan cuando el contenedor se detiene o elimina, por lo tanto, ¿cómo podemos guardar los datos? Hay dos formas de hacerlo vinculando el montaje y los volúmenes.

* **Bind mount**: Nos ayuda a vincular un directorio en nuestra máquina a un directorio en un contenedor. Esto permite guardar datos en nuestro directorio y usar los mismos datos cada vez que usamos el contenedor. Pero puede ser peligroso porque estamos permitiendo que el contenedor tenga acceso a un lugar en nuestra máquina y si el contenedor es malicioso, puede usar de mala manera cualquier información que tengamos en ese directorio.
* **Los volúmenes**: Son la evolución de la montura bind. Los volúmenes resuelven los problemas de seguridad y privacidad que se encontraban en los montajes de enlace.

**¿Como funciona?**

La parte del disco que compartimos con docker está completamente administrada por docker, y no tenemos acceso a menos que tengamos privilegios, pero es muy conveniente porque solo tenemos un lugar donde se almacenan los datos de nuestro docker, y solo docker puede realizar modificaciones en él.

**Ventajas**

* No se tienen archivos tirados en el disco.
* Nadie puede acceder a ellos.
* Los maneja únicamente docker.
* No hay que acordarse la ruta.

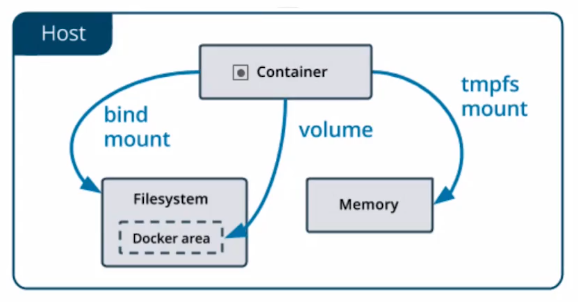
**Desventajas**

* Sí se quiere ver esos archivos para tocarlos, cambiarlos desde la maquina local, es bastante más complicado.

# 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**) y un tercero sólo en Linux y es **tmpfs mount**, el cual sólo existe en memoria del contenedor, el primero consiste en compartir un directorio de la máquina host con el contenedor y el segundo es más práctico consiste en crear un Volumen 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
10. cd testing && ll
11. 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

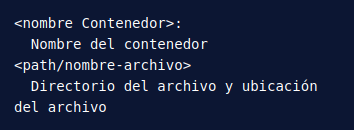
* sudo docker cp <nombre Contenedor>:<path/nombre-archivo> <path/nombre-archivo>

**flag**

cp :

**Instrucción para Copiar**

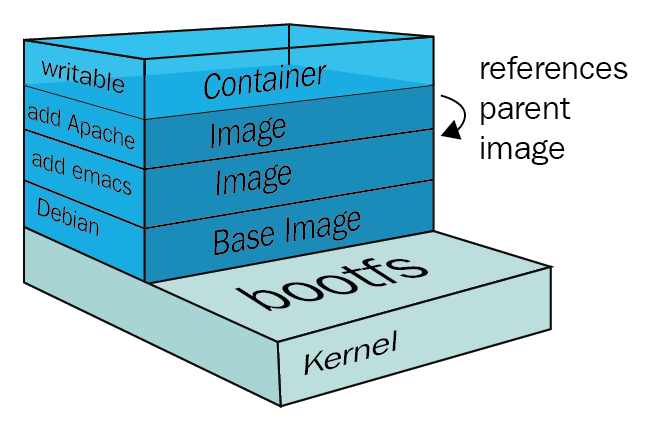
Parámetros

**Conceptos fundamentales de Docker: imágenes**

Una imagen contiene distintas capas de datos (distribución, diferente software, librerías y personalización).

Podemos llegar a la conclusión, que una imagen se conforma de distintas capas de personalización, en base a una capa inicial (**base image**), la dicha capa, es el más puro estado del SO.

La siguiente ilustración nos mostraría la representación gráfica, del concepto de una imagen en Docker.

Si observamos, partimos desde la base del SO, y vamos agregando capas de personalización hasta obtener la imagen que necesitamos:

* Distribución debian.
* Se agrega el editor emacs.
* Se agrega el servidor Apache.
* Se agregan los permisos de escritura para la carpeta **/var/www** de apache.

Hay que tener en cuenta, que todo parte del Kernel de Linux, en caso de utilizar alguna distribución de Linux.

**Histórico de una imagen**

Podemos observar la historia de nuestra imagen, con el siguiente comando.

De esta manera podemos ver las capas de personalización que fueron agregadas, para la construcción de la imagen que conocemos.

# El sistema de capas

Las imágenes son un conjunto de capas, y a partir del **Dockerfile** se puede saber como está construido una imagen. Existen distintas maneras de ver las capas que conforman una imagen.

1. Si la imagen es pública se puede visitar Dockerhub y buscar una imagen para ver su Dockerfile. Una vez encontrar la imagen buscamos en los tag y haciendo clic en los tag se mostrará el docker file para construir dicha imagen.
2. Se puede hacer a través de la línea, esta opción no es muy cómoda. Es importante destaca que los cambios se presenta por filas y las filas más inferiores son los cambios más viejos y las que están más arriba son los cambios más recientes. el comando es:

**sudo docker history <nombre imagen: nombre tag>**

**Ejemplo:**

**sudo docker history ubuntu:platzi**

Otra opción es una herramienta de terminal creada por un usuario llamada **dive https://github.com/wagoodman/dive**, al ejecutar esta herramienta se puede visualizar que del lado izquierdo las capas que tiene la imagen y del lado derecho se visualiza el detalle de cambios que genera cada capa. el comando es el siguiente.

**sudo dive <nombre de imagen:nombre tag>**

Ejemplo:

**sudo dive ubuntu:platzi**

Los cambios que se muestran en la terminal son de la siguiente forma, los **más** **antiguos** abajo y los **más** **recientes** arriba.

En la construcción de una imagen, es importante tomar en cuenta:

* En la construcción de una imagen, se puede optar por instalar paquetes o crear archivos en una capa y luego desinstalar los paquetes o eliminar los archivos en otra capa. Esto es poco óptimo porque las capas son inmutables y en este caso estaríamos desperdiciando espacios. El motivo de que las capas son inmutables es para que al momento de crear una imagen docker no agrega capas ya creadas.
* Una estrategia para evitar desperdiciar espacio es que en la misma capa se cree o instalen los paquetes y luego de terminar la operación que esto realizan se desinstale o eliminen.

Otra estrategia de optimizar es que una vez creado el creada la imagen todas las operaciones se hagan desde un contenedor.

**Docker Compose: la herramienta todo en uno**

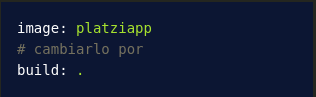
**Docker Compose** es una herramienta que permite simplificar el uso de Docker. A partir de archivos **YAML** es mas sencillo crear contenedores, conectarlos, habilitar puertos, volúmenes, etc.

Con Compose puedes crear diferentes contenedores y al mismo tiempo, en cada contenedor, diferentes servicios, unirlos a un volumen 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.

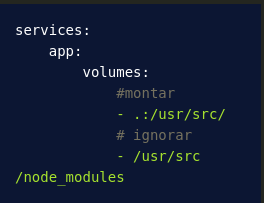
**Docker Compose como herramienta de desarrollo**

* Hacer que Docker Compose, en lugar de correr una imagen, en su lugar cree una imagen y la use.

**Tips**: docker-compose build app para buildear solo un servicio y el compose lo detecte.

* Si quiero que el código sea responsivo, hay que hacer cambios.

Como **docker-compose** se fija en el cambio de imagen, pero la imagen **no es recreada ante un cambio**. Hay que agregar al Docker Compose lo siguiente:

Cambiar el comando de inicio.

**Compose en equipo: override**

**Contexto del archivo.override.yml**

El **archivo.yml** que utiliza docker compose normalmente iría versionando, pero surge un problema, el problema es que a veces se realizan cambios en el archivo.yml (archivo base) para probar ciertas cosas, y luego se desea hacer commit en el proyecto sin que se commitee los cambios en el archivo.yml (archivo base). La solución que propone docker es utilizar un

**docker-compose.override.yml** que contenga esas variaciones del archivo.yml (archivo base). Al momento de ejecutar docker-compose, este se da cuenta qué, en nuestro **docker-compose.override.yml**  va a sustituir o tomar ciertas configuraciones de nuestro archivo.yml (archivo base)

**¿Qué es un docker-compose.override.yml?**

**docker-compose.override.yml** es un archivo que sirve para personalizar o hacer pequeños cambios propios para nuestro ambiente sobre el **docker-compose** original sin la necesidad de alterar el mismo.

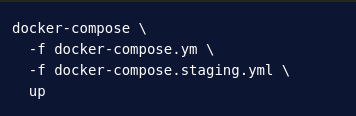
Pueden tener diferentes overrides por entorno por ejemplo:

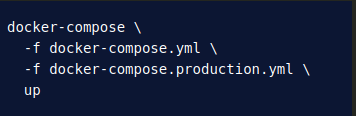
* **docker-compose.override.yml** para development
* **docker-compose.staging.yml** para staging
* **docker-compose.production.yml** para production

Y debes correrlos de la siguiente forma:

**development**

**staging**

**production**

**¿Cómo funciona un docker-compose.override.yml?**

Se puede agregar variables de entorno en ambos archivo y lo que hará **docker-compose** es una especie de **merge**, como ocurre en git, es decir, creará ambas variables de entorno. Se puede realizar un **build** también en el archivo base.

* Funcionamiento de los **port**

En el caso de crear los puertos, si es recomendable hacerlo solamente en nuestro archivo.yml (archivo base). Esto se debe a que los puertos se procesan en etapas distintas.

* Funcionamiento de **depends\_on**

Este caso también se toma desde el archivo.yml (archivo base)

**Pasos para configurar un “docker-compose.override.yml” con el ejemplo anterior**

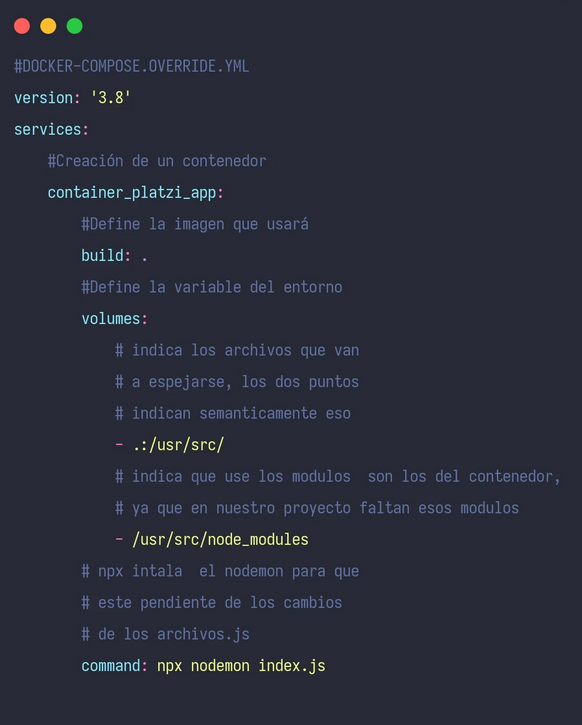
1. Crear el archivo con el nombre docker-compose.override.yml
2. Verificar que el archivo existe.
3. Configurar el anterior archivo base como estaba e intentar lo siguiente:
   * Los puertos (**port**) y las dependencias (**depend\_on)** deberían ir en el archivo base y, el archivo orverride los tomará de allí.
   * Si declaramos variables de entorno (**enviroment**) tanto en nuestro archivo base (**docker-compose.yml**) y en nuestro archivo override (**docker-compose.override.yml**), lo que hará docker compose es que tomará las dos variables de entornos y las creara.
   * Otro punto importante es que tenemos en nuestro archivo base (doker-compose) un servicio con **image** y en un archivo override (docker-compose-override) un **build**. El nombre que tomará nuestro servicio será el nombre de la carpeta donde está y el nombre del servicio.

5.Luego levantar los servicios.  
6. Realizar la prueba en el navegador.

Ejecución de lo pasos para configurar un **docker-compose.override.yml** con el ejemplo anterior

1. touch docker-compose.override.yml
2. ls -al
3. Archivo **docker-compose.yml**



1. Archivo **docker-compose.override.yml**
2. sudo docker-compose up -d
3. localhost:3000

**Pasos para escalar contenedores**

Existe la posibilidad de escalar los contenedores para realizar distintas pruebas para ello se utiliza los siguientes pasos:

1. Eliminamos los anteriores servicios para modificar el archivo base, añadiendo un rango de puertos.
2. En nuestro archivo **docker-compose.yml** en la sección de puerto, definimos un rango de puerto para nuestros contenedores. La explicación es la siguiente:
   * Dos contenedores se asocian a un puerto físico, pero internamente cada contenedor tiene su propio puerto virtual. (Dos servicios no pueden escuchar en el mismo puerto físico)
3. Levantamos los servicios escalandola a 2 instancias.
4. Realizamos las pruebas en el navegador.

**Ejecución de los pasos para escalar contenedores**

1. docker-compuse down
2. Archivo docker-compose.yml
3. sudo docker-compose up -d --scale <name\_container>=2 esto nos permitirá ver en una pestaña **localhost:3000** , en otra pestaña **localhost:3001**

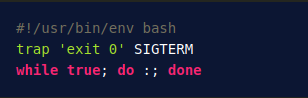
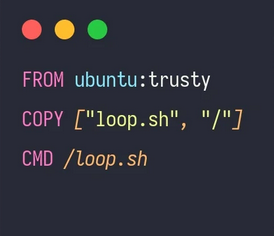
# Deteniendo contenedores correctamente: SHELL vs. EXEC

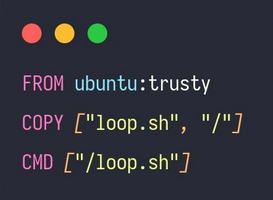
Al estar activo o recién crear un contenedor, este debería estar ejecutando un proceso **principal** o **main process** para mantenerse en funcionamiento. En caso de que el **main process** se detenga el contenedor debería dejar de funcionar.

Docker tiene una manera de manejar los procesos de los contenedores de manera estándar, cuando se manda a detener el contenedor con la instrucción **docker stop <name-container>;** con esta instrucción Docker manda una señal estándar de linux llamada **SIGTERM** al proceso, después de un tiempo el proceso deberá detenerse, en caso de no detenerse el proceso enviará otra señal llamada **SIGKILL**, con esta señal se garantiza que el proceso se ha terminado de manera forzada.

Existen problemas en la terminación de procesos ejecutados en los contenedores, una de las causas está asociado a **¿Cómo se debería declarar en Dockerfile el proceso a ejecutar?**

Digamos que tenemos estos dos archivos configurados de la siguiente manera.

El primero es el dockerfile, sirve para construir nuestra **imagen** y el segundo es el archivo **bash** que se ejecutará como un **main process** o proceso principal del contenedor. Es importante destacar que la línea de CMD ejecuta nuestro archivo **bash**, a través de un **shell**, esta forma de expresar esta línea se conoce como **shell form** y esto genera que al momento de detener un contenedor que esté ejecutando un proceso por **shell form** necesariamente tendrá que usar un **SIGKILL**. Para evitar esto, el **CMD** debe ejecutar nuestro **main process** de **form exec form** , es decir, nuestro archivo debe estar configurado de esta manera.

La idea de que el archivo esté configurado de esta manera es que le de tiempo a que el contenedor procese las solicitudes restantes y pueda hacer un **Great full shutdown**. Otra nota importante es que los códigos de salida mayor a **128** es el resultado de una salida por un código de excepción.

# Contenedores ejecutables: ENTRYPOINT vs CMD

En docker podemos construir imagines a partir de un archivo llamado **dockerfile**.

El archivo nos permite ejecutar un comando que se ejecutara una vez inicie nuestro contenedor.

**¿Qué ocurre al definir un CMD sin definir un ENTRYPOINT?**

Si sólo especificas un CMD, entonces Docker ejecutará dicho comando usando el ENTRYPOINT por defecto, que es **/bin/sh -c**.

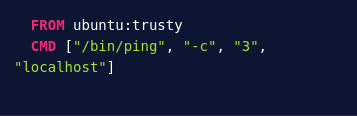
Respecto al “entrypoint” (punto de entrada) y “cmd” (comando), puedes sobrescribir ninguno, ambos, o sólo uno de ellos.

Si especificas ambos, entonces:

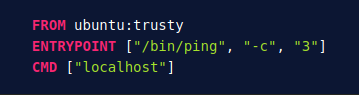
* **ENTRYPOINT**: Especifica el ejecutable que usará el contenedor.
* **CMD**: Se corresponde con los parámetros a usar con dicho ejecutable.

**Ejemplo sin entrypoint**

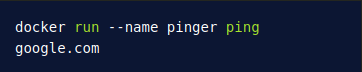
Al no contener el entrypoint toma **/bin/ping** y toma como parámetros los demás elementos "-c", "3", "localhost", sin embargo el argumento **localhost** es fijo por lo que no podemos hacer **ping** a otros destinos.

**Ejemplo con entrypoint y cmd**

Al usar **entry point** toma el ejecutable **/bin/ping** como aplicación de entrada y lo que esta en **CMD ["localhost"]** se puede cambiar por lo que podemos hacer **ping** a diferentes destinos.

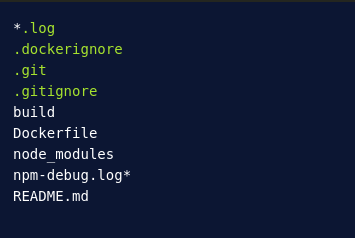
**Ping a localhost**

**Ping a cualquier otro host**

**El contexto de build**

Cuando hacemos un **build** debemos tener en cuenta los archivos que no queremos agregar a la imagen,

para ignorar debemos usar **.dockerignore**

* Volver a hacer el build de la imagen

**docker build -t <image\_name>** .

* Volver a ejecutar el contenedor

**docker run -d --name <container\_name> <image\_name>**, Si ya existe un contenedor con ese nombre hay que eliminar la antigua versión primero.

**Multi-stage build**

En ambientes productivos y de desarrollo a veces, en el entorno de desarrollo tenemos código que no queremos que esten en producción y viceversa, para ello existen las **fases de build**.

* 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 interesa 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.

**Docker-in-docker**

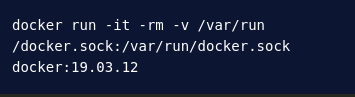
Docker-in-Docker, también conocido como **dind**, es una técnica que permite ejecutar contenedores Docker **dentro** de otros contenedores Docker. Esto significa que puedes **crear** y **ejecutar** un entorno Docker completo **dentro** de otro entorno Docker.

NOTA: Esto sólo funciona en Linux y Mac.

Esto se logra gracias al funcionamiento de linux en base a archivos. Donde se le puede hacer un **bind mount** al archivo socket de docker con el interno de un contenedor. Docker deamon expone una interfaz para hablarle con el socket **/var/run/docker.sock**.

Esto se realiza con **-v /var/run/docker.sock:/var/run/docker.sock**.

Ejecutemos un contenedor:

Este contenedor esta conectado su docker al de la maquina. Se puede ejecutar por ejemplo: **docker ps** y verse a si mismo.

**Docker-in-Docker Dive**

Esta es una herramienta de análisis de imágenes.

Se podría montar en un contenedor el **dive** y revisar las imágenes en la maquina principal.

Pero también habrá que compartirle el cliente de docker porque no tiene el suyo propio. Con **-v $(which docker):/bin/docker**.

Esto se logra con:

