# Ultimate Docker

Creado por: Kharoonn Makvhidar

## Introducción a Docker.

### ¿Qué es Docker?

Es una plataforma para poder Construir, Ejecutar y Desplegar aplicaciones.

¿Tu aplicación no funciona en Producción? Puede ocurrir por 1 de 3 razones:

- Falta 1 o más archivos (la app se encuentra incompleta en producción).
- Las configuraciones son distintas (tal vez una variable de entorno es distinta en tu máquina local a la que está en producción).
- Hay discordancia en el Software (tú tienes instalada la versión 3.11 de Python, pero en producción tienes la versión 3.6).

Esto se soluciona con Docker. ¿Cómo? generando paquetes que contengan todas las dependencias de tu proyecto. De esa forma, cualquier persona con una computadora que tenga instalado Docker, podrá descargar ese paquete desde la nube y ya tendrá tu aplicación funcionando, igual que en tu máquina.

Esto se hace con el comando ***docker-compose up***, y no tendrán que descargar ninguna dependencia, ni configurar ninguna variable de entorno, entre otras muchas cosas.

Además, Docker nos permite tener más de una versión de nuestra aplicación en la misma máquina. Cada paquete se encuentra en ambientes aislados entre sí.

**¿Qué es un contenedor?**
Es un ambiente aislado para poder ejecutar aplicaciones. Se puede detener y reiniciar, y además podemos crear múltiples contenedores en base a la misma imagen. (Ya veremos lo que son las imágenes)

**¿Qué es una máquina virtual?**
Podríamos decir, que a diferencia de un contenedor ésta es una abstracción de Hardware Físico a la cual le podemos instalar un Sistema Operativo.

¿Podemos crear más de una máquina virtual y cada una un Sistema Operativo distinto? la respuesta es Sí.

¿Qué necesitamos para hacerlo? Un Hypervisor, y ¿Qué es un Hypervisor?
En términos simples, diremos que es una aplicación que se instala en nuestra computadora y nos permite crear máquinas virtuales.

Tipos de Hypervisors (los más relevantes) que existen en el mercado:

- Virtual Box
- VMWare
- Parallels → Exclusivo de MacOS
- Hyper-V → Exclusivo de Windows

**¿Qué problemáticas tienen las máquinas virtuales?**

1. Cada máquina tendrá una copia completa del S.O, qué, dependiendo de ese S.O, cada imágen puede llegar a pesar varios Gb.
2. Lentas para iniciar
3. Muchos recursos (hay que asignarle de forma estática recursos: CPU, RAM, Almacenamiento, etc.)

Ahora bien, **¿Cómo son los contenedores, respecto de las máquinas virtuales?**

1. Como dijimos, también nos permiten correr aplicaciones en ambientes aislados.
2. Ocupan mucho menos espacio en disco (esto, por lo general es porque trabajas con versiones recortadas de los S.O)
3. Utilizan el mismo S.O de la máquina Host.
4. Inicio más rápido
5. Necesitan menos recursos

## Arquitectura de Docker.

Se componen de un cliente y un servidor (Docker Engine), y este cliente se comunica con el servidor mediante una API Rest.

Los contenedores que nosotros ejecutamos con Docker son procesos.

```
graph TD
    A["Cliente Docker"]
    B["Docker Engine"]
    C["API REST"]
    D["Contenedor 1"]
    E["Contenedor 2"]
    F["Contenedor N"]
    G["Sistema Operativo Host"]

    A -->|"Comandos"| C
    C -->|"Comunicación"| B
    B -->|"Gestiona"| D
    B -->|"Gestiona"| E
    B -->|"Gestiona"| F
    D --> G
    E --> G
    F --> G

    %% Los contenedores comparten el SO del host
    %% Cada contenedor es un proceso aislado
    %% El Docker Engine gestiona los contenedores
```

Este diagrama representa la arquitectura de Docker, mostrando:

- El Cliente Docker que envía comandos.
- La API REST que facilita la comunicación.
- El Docker Engine que gestiona los contenedores.
- Múltiples contenedores ejecutándose como procesos aislados.
- El Sistema Operativo Host que es compartido por los contenedores.

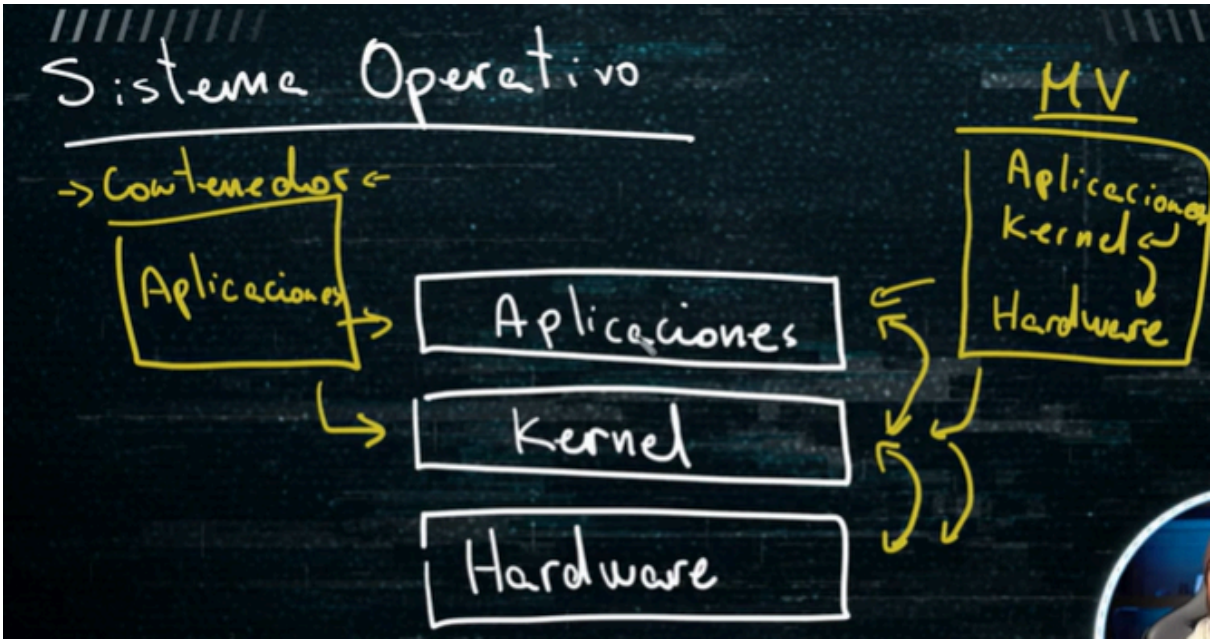
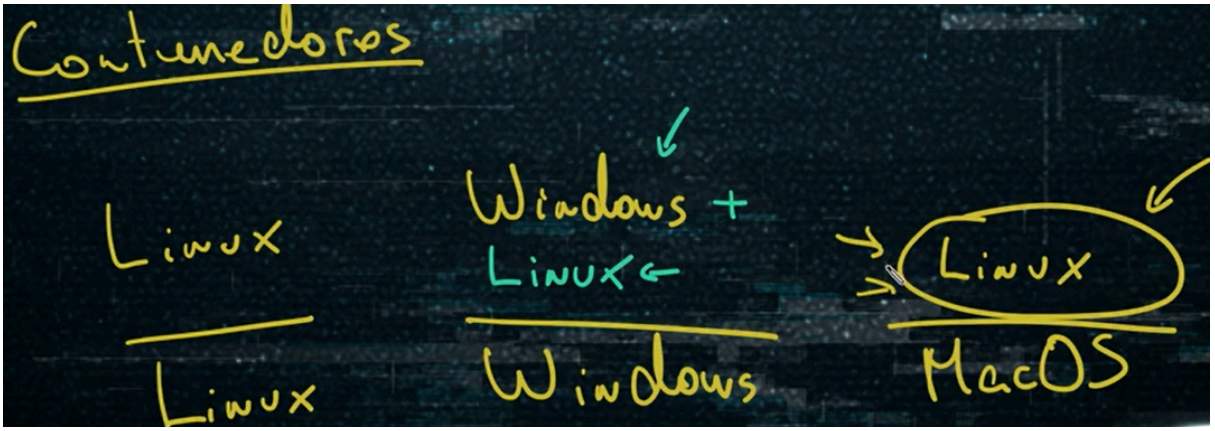



Diagrama que representa dónde se instalan tanto las máquinas virtuales (MV) como los contenedores y como se comunican con el kernel del S.O Host.




Este gráfico nos muestra por debajo, los S.O que tienen nuestra máquina Host, y por arriba los contenedores que pueden ejecutar esas máquinas.

En MacOS, su kernel no soporta Docker, pero esto se soluciona cuando instalamos Docker, ya que él mismo se va a encargar de crear una máquina virtual de Linux y así poder ejecutar contenedores de Linux.

## Proceso de desarrollo con Docker.

Para empezar, supongamos que ya tenemos una aplicación desarrollada. Ahora hay que “dockerizarla”, y esto se hace agregando un archivo “Dockerfile” en la raíz de nuestro proyecto.

Dockerfile es un archivo de texto plano el cual Docker va a utilizar para poder empaquetar nuestras aplicaciones y de esta manera construir imágenes en base a nuestras aplicaciones.

¿Qué contienen esas imágenes?

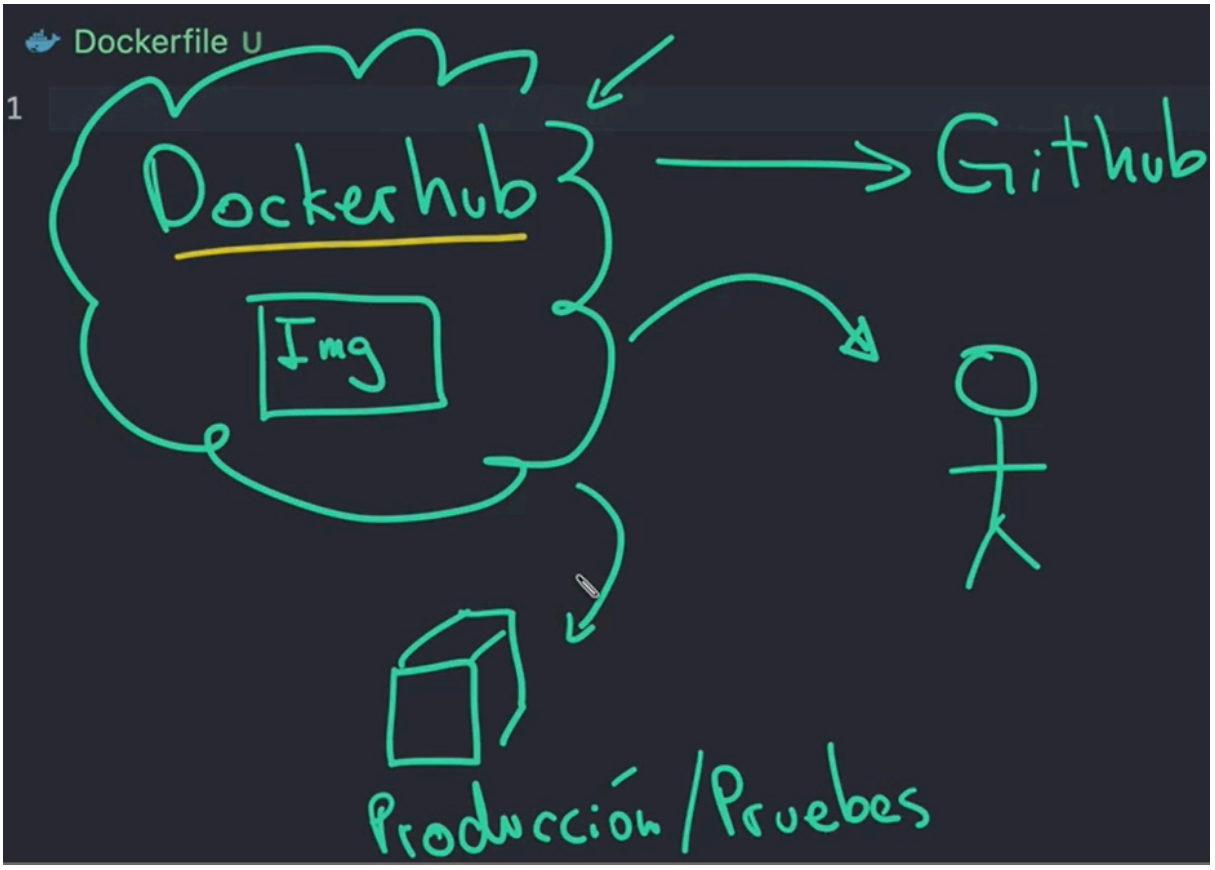
Básicamente, todo lo que necesita nuestra aplicación para poder ejecutarse.

1. Versión pequeña del S.O
2. Ambiente de ejecución (NodeJS, Python, etc.)
3. Archivos de la aplicación
4. Dependencias de terceros
5. Variables de entorno

Y una vez que tenemos nuestra imagen, le podemos decir a Docker que cree un contenedor en base a la imagen que acabamos de crear.

Entonces, un contenedor es sencillamente un proceso (un proceso especial porque cuenta con su propio sistema de archivos provisto por la imagen). Y esa imagen  no es más que nuestra aplicación.

Esa imagen nosotros la podemos subir a un sistema de registros, como Docker Hub.




Docker Hub, para que nos demos una idea... es como GitHub, y Docker, como Git.

Nosotros podemos subir nuestras imágenes a Docker Hub y éstas pueden ser descargadas por cualquier desarrollador e incluso en diferentes entornos y van a ser exactamente las mismas.

## Terminal de Linux

Veremos algunos comandos y atajos para movernos en la terminal de Linux.

Cabe aclarar que todo Linux es case sensitive. Por lo que no es lo mismo #pwd que #Pwd.

#ls → me lista los directorios y archivos del directorio donde me encuentro parado

y ¿cómo sé el directorio en el cuál me encuentro? #pwd

#ls -1 → me lista lo mismo, pero uno debajo del otro.

#ls -l → también me lista lo mismo, pero me ofrece mas detalles.

Otra forma de listar, por ejemplo, si quiero ver lo que tiene un directorio en particular, supongamos, el directorio var, sería #ls var

#whoami → me dice quién soy, qué tipo de usuario soy.

#cd → me permite cambiar de directorio.

#cd ~ → para volver al directorio /root

#echo $0 → me dice el tipo de shell que estoy utilizando.

#echo hola mundo → me imprime por consola la cadena “hola mundo”.

#clear → me limpia la consola.

#history → me muestra el historial de comandos ejecutados.

¿Qué atajos existen?

flecha arriba/flecha abajo → me permite moverme por el historial de comandos.

CTRL+C → para cancelar cualquier comando o ejecución.

CTRL+R → para buscar algún comando que puede ser difícil de encontrar en el historial, sobre todo cuando ejecutamos muchos comandos y no sabemos exactamente cuánto nos tenemos que mover hacia arriba. Es una búsqueda avanzada, básicamente. También te permite moverte en el historial pero específicamente sobre aquellos comandos que contengan ese caracter o esa cadena que vos estás buscando. Es muy útil.

#!5 → me va a ejecutar el comando de la posición 5 en el historial

#mkdir → para crear directorios.

#touch → para crear archivos, y también te permite modificar su fecha de modificación.

#rm holamundo.txt → elimina el archivo holamundo.txt

#rm a*.txt → elimina TODOS los archivos que comiencen con una “a” y terminen con “.txt”, el resto puede contener cualquier combinación de caracteres.

#rm -r midirectorio/ → para borrar directorios. Es un borrado recursivo (-r).

#nano → si ya instalaste previamente nano, pues, es un editor de texto. Allí podremos editar archivos.

#cat → visualiza el contenido de un archivo. También nos va a permitir concatenar 2 o más archivos en 1.

#more → te permite visualizar de una manera más cómoda archivos muy grandes, o que contienen muchísimas líneas. Simplemente con las flechitas arriba/abajo te vas a poder mover tranquilamente por el archivo.

En el caso de que no te deje moverte hacia arriba, te podés descargar “less”.

#apt install less, #less → y listo, ahora deberías poder usar este comando para visualizar el archivo.

Para salir, tanto de more, como de less, usamos la tecla Q.

#head -n 5 → para ver las primeras 5 líneas de un archivo.

#tail -n 2 → para ver las últimas 2 líneas de un archivo.

#tail -f → para seguir en tiempo real las últimas líneas de un archivo. (-f de follow).

#echo “hola mundo” > saludo.txt → redirecciona la salida del comando echo “hola mundo” al archivo saludo.txt (que si no existe, lo crea) y si ya existe, utiliza el mismo. Un solo > reemplaza el contenido (si es que el archivo de destino tenía algún contenido). Dos >> agrega al final del archivo la salida (en este caso) del comando echo “hola mundo”.

Por defecto, los errores se envían a dev/stderr. Para cambiar esto, podríamos hacer lo siguiente:

#ls -al noexistearchivo.txt 2> salida.txt

y ¿Cómo puedo manejar el éxito/fracaso?

#ls -al noexistearchivo.txt > salida.txt 2>&1 → si ocurrió un fracaso o error, lo envía a salida.txt

pero si lo hacemos con un archivo que sí existe, por ejemplo:

#ls -al archivo.txt >salida.txt 2>&1 → también me va a enviar los éxitos al archivo salida.txt

#grep Hola saludo.txt → nos va arrojar el resultado con la cadena “Hola” en otro color (si es que la encuentra).

Como dijimos, Linux es case sensitive, y por lo tanto, grep también. ¿Cómo le indico que ignore las mayúsculas y minúsculas? #grep -i hola saludo.txt

También le puedo pasar más de un archivo: #grep -i hola saludo.txt archivo.txt

y ¿Cómo hago si son muchos archivos los que le tengo que pasar? Bueno, si todos esos archivos tienen en común la extensión, por ejemplo, .txt puedo hacer lo siguiente:

#grep -i hola *.txt → buscará la cadena “hola” ignorando el case, en todos los archivos con extensión .txt

#grep -ir hola . → buscará la cadena “hola” ignorando el case (-i) en el directorio actual (-r). En Linux se pueden combinar los argumentos en uno solo (-ir).

#find → para buscar archivos y directorios.

#find /etc -type f -iname “a*” → me buscará todos los archivos en el directorio /etc que comiencen con la letra a, ignorando si es mayúscula o minúscula.

Si quiero buscar directorios en lugar de archivos, -type d.

Si quiero buscar con el case sensitive, simplemente no le paso el argumento -i.

#mkdir holamundo ; cd holamundo ; echo “directorio creado” → me mostrará la cadena “directorio creado”. ¿Qué sucedió? encadenamos comandos con ;

Podemos hacer encadenamientos condicionales.  ¿Cómo?

#mkdir holamundo && cd holamundo && echo “directorio creado” → se ejecutará el siguiente comando siempre y cuando su antecesor haya sido exitoso.

#mdkir holamundo || cd holamundo || echo “directorio creado” → se ejecutará el siguiente comando siempre y cuando su antecesor haya fallado.

#ls -1 /etc | less → enviará la salida del primer comando al segundo, es decir a less en este caso, y este...bueno, ya sabemos lo que hace.

Con el comando ;\ lo que hacemos es encadenar comandos (por el ;) pero con la diferencia de que el \ nos va a permitir escribir el siguiente comando en una nueva línea. Básicamente es un salto de línea.

#env → contiene las variables de entorno.

#echo $PATH → nos devuelve el valor de la variable de entorno PATH.

#printenv PATH → nos devuelve lo mismo que el anterior.

#printenv → me va a imprimir todas las variables de entorno, al igual que #env

#echo “hola mundo!” > /bin/holamundo → por defecto, los archivos en linux no tienen permiso de ejecución.

#chmod +x /bin/holamundo → con  esto le doy permisos de ejecución.

#export HOLAMUNDO=”soy una variable de entorno” → para crear variables de entorno, en este caso HOLAMUNDO.

El problema que tiene crear variables de entorno de esa forma es que no van a persistir de sesión en sesión. Ni bien cerremos la sesión, dejará de existir.

¿Cómo debo hacer para que se guarden?

#echo HOLAMUNDO=”soy una variable de entorno” >> .bashrc → IMPORTANTE: no olvidar que deben ser dos >> para que se agregue al final, o de lo contrario habrán reemplazado todo el contenido del archivo .bashrc

#nano .bashrc → le agregamos comillas al valor de  nuestra variable.

#source .bashrc → y listo, ahora podríamos poder imprimirla haciendo #echo $HOLAMUNDO o #printenv HOLAMUNDO

#ps → para ver el listado de procesos.

#sleep 500 & → para mandar una espera de 500 segundos a segundo plano.

#jobs → nos muestra los procesos que están corriendo en segundo plano.

#fg %1 → para regresar un proceso de segundo plano. En particular, el [1].

#kill 44 → matamos el proceso con id 44.

#kill -9 44 → hacemos lo mismo, pero el -9 hace un forzado de la eliminación del proceso con id 44. (Siempre es recomendable usar #kill 44, o el id que corresponda).

## Gestión de paquetes en Linux (particularmente en distribución Ubuntu)

Lo hacemos con la herramienta ***apt*** (que es una versión más nueva de ***apt-get***)

Le podemos pasar los siguientes argumentos:

1. Install → para instalar un paquete.
2. remove → para remover un paquete.
3. list → para listar todos los paquetes disponibles que nosotros podemos instalar.

4. autoremove → para eliminar todos los paquetes que no están siendo utilizados por otros paquetes (paquetes innecesarios, dicho de otra forma).
5. update → para sincronizar la base de datos que contiene todos los paquetes disponibles para actualizar.

¿Qué significa esto último de update?

Si nosotros hacemos apt install nano y nos salta un error diciendo que no se encuentra un paquete “nano” disponible, es probablemente, porque no tengamos actualizado nuestra lista de paquetes disponibles.

apt update viene a solucionar esto, sincronizando nuestra base de datos de paquetes disponibles con la base de datos en internet.

## Gestión de usuarios en Linux

#useradd → para agregar usuarios.

#usermod → para modificar usuarios.

#userdel → para eliminar usuarios.

los comandos que ejecutemos en la terminal con el símbolo “#” son en modo admin, o root.

los comandos que ejecutemos en la terminal con el símbolo “$” son en modo user.

#useradd -m felipe → agregamos un usuario con directorio llamado felipe (por defecto le va a dar la shell ‘sh’).

#usermod -s /bin/bash felipe → le cambio la shell a bash.

#su felipe → cambio de usuario a felipe.

¿Cómo podemos ingresar en un contenedor de Docker con el nuevo usuario felipe que creamos?

#docker start -i id_contenedor → para iniciar el contenedor en caso de que esté parado.

#docker exec -it -u felipe id_contenedor bash → -it para hacerlo interactivo y -u para indicarle el usuario, en este caso felipe; id_contenedor para indicarle el contenedor y bash para indicarle con qué shell quiero iniciar sesión.

#id → nos va a mostrar el identificador de usuario, el grupo principal al cual pertenece, y también los grupos secundarios a los cuales pertenece.

## Gestión de grupos en Linux.

Tenemos las mismas 3 primeras acciones que vimos en usuarios.

#groupadd → para agregar grupos.

#groupmod → para modificar grupos.

#groupdel → para eliminar grupos.

Por defecto, cuando creamos un usuario se le asigna a éste un grupo del mismo nombre. Por lo tanto, como creamos el usuario felipe, su grupo principal también será felipe.

¿Cómo lo agregamos a otro grupo?

#usermod -G devs felipe → agregamos al usuario felipe al grupo devs.

#groups felipe → deberíamos ver algo como felipe : felipe devs

y si hacemos #id felipe → deberíamos ver algo como,


Si se da el caso de que no aparece como grupo secundario el grupo principal (suena confuso, volvé a leerlo), que es el comportamiento esperado, lo que podemos hacer es lo siguiente:
#usermod -a -G devs felipe

## Permisos en Linux.

Para ello creamos un archivo ejecutable, #echo “echo hola mundo” > archivo.sh

Por defecto, como habíamos mencionado anteriormente, los archivos no tienen permiso de ejecución (x, de execute). ¿Cómo cambiamos esto?

Para los usuarios:

#chmod u+x archivo.sh

Para los grupos:

#chmod g+x archivo.sh

Para otros:

#chmod o+x archivo.sh → sin embargo esto es peligroso para la seguridad, así que no es recomendable hacerlo.

Para agregar permisos usamos + y para quitar usamos -.

Lo mismo podemos hacer con los permisos de lectura (r, de read) y de escritura (w, de write). Y también podemos combinar esos permisos en una sola línea, haciendo:

#chmod u+x, g+x, o+x archivo.sh

Por otro lado, hay otra forma de asignar o quitar permisos, más directa.

#chmod u=rw archivo.sh → esto, por un lado, está asignando al usuario el permiso de lectura y de escritura, y por el otro, está quitándole el permiso de ejecución. Los permisos que no le pasemos en esta sintaxis, los quitará.

¿Hay otra forma? Sí. Existe otra forma de asignar o quitar permisos que es con los valores que tiene cada permiso, pero no vamos a entrar en detalle con eso. A modo de introducción, podríamos decir que cada permiso tiene un valor asociado.

r = 3; w = 5; x = 7.

## Construcción de imágenes.

Lo primero que vamos a hacer es descargar el proyecto que está adjunto en el curso, es una app usando Vite+React. No es necesario conocer nada acerca del código, simplemente lo descargamos y ya.

También debemos asegurarnos de descargar NodeJs. Es muy simple, lo descargamos, lo instalamos y listo. Asegurarse de que estamos añadiendo la variable de entorno de la ruta de NodeJs a la variable del sistema PATH.

Probablemente una vez que lo descarguemos, si teníamos alguna terminal abierta, debamos cerrarla para que te detecte la instalación de NodeJs, de lo contrario puede que no se actualice automáticamente y no te lo reconozca.

¿Qué vamos a hacer con este proyecto que descargamos?

Lo primero es descomprimirlo en una carpeta que sea accesible y luego vamos a abrir esta carpeta en VSCode. Una vez dentro, vamos a abrir la terminal y vamos a ejecutar el siguiente comando.

$npm install → para instalar las dependencias del proyecto.

$npm dev o npm run dev (una de las dos) → para ejecutar el proyecto y poder verlo en el navegador

Ahora creamos el archivo **Dockerfile**

FROM → para indicar la imagen base.

WORKDIR → para cambiar el directorio de trabajo.

COPY, y ADD → para agregar archivos.

RUN → para ejecutar comandos.

CMD, y ENTRYPOINT → sirven para especificar los comandos que se van a ejecutar al iniciar el contenedor. La diferencia entre ellos es que nosotros, con CMD, podemos reescribirlo desde la línea de comandos y con ENTRYPOINT no.

EXPOSE → para exponer los puertos de nuestra aplicación.

ENV → para definir variables de entorno.

USER → para asignar un usuario con permisos acotados.

Hay más etiquetas de Docker, pero éstas son las más comunes.

Al momento de seleccionar una imagen desde Docker Hub debemos ser muy cuidadosos, ya que si seleccionamos erróneamente una imagen (lo más común es seleccionar imágenes que no especifican versiones) luego podríamos llegar a sufrir problemas o comportamientos no esperados, comportamientos erráticos en nuestra aplicación.

Entonces, recomendación de buena práctica: NO SELECCIONAR LATEST en las imágenes. Y tampoco dejes vacías las versiones, ya que por defecto te traerá la imagen con la etiqueta latest.

Siempre asegurarse de que contengan las versiones específicas, es decir, si vamos a trabajar con una versión de node, podríamos hacer lo siguiente:

**FROM node:20.5.0-alpine3.18** → la versión de alpine es una distribución de Linux muy liviana. Será muy común verla en Docker.

Lo siguiente sería construir la imagen:

**#docker build -t app-react .** → el . es para indicar el directorio actual, que sería en el directorio raíz del proyecto. Asegurarse de estar en ese directorio antes de ejecutar el comando.

**#docker run -it app-react sh** → sh porque alpine no tiene bash.

## Copiando archivos

Ya vimos que para agregar o copiar archivos podemos usar la instrucción de COPY, y hay varias formas de indicar qué archivo/s quiero copiar y en dónde los voy a pegar, es decir, su ruta de destino.

COPY package.json /app → pega el archivo package.json en un directorio llamado app

COPY package.json README.md /app/ → cuando copiamos más de un archivo, tenemos que usar otra barra al final.

COPY package*.json /app/ → copia archivos que cumplan con ese patrón en el directorio app.

COPY . /app/ → copia todos los archivos del directorio actual, donde estoy parado, en el directorio app.

También podríamos usar la instrucción WORKDIR para cambiar el directorio de trabajo y le indicaríamos el destino simplemente con un punto, pero ojo... porque ya no utilizaríamos el punto en el archivo de origen, porque ese archivo que queríamos copiar no se encuentra en el directorio en el que estamos trabajando ahora.

¿Qué pasa con los archivos que tienen espacio en sus nombres?

COPY [”curso docker.txt”, .]

¿Qué diferencia tiene con la instrucción ADD?

Pues que con ADD podemos indicarle una URL donde se encuentran los archivos, por ejemplo:

ADD https://www.miweb.com/documentos/clase7.txt /app

También puede descomprimir archivos.

ADD archivos.zip . → te copia los archivos (y te los descomprime) en el directorio app. (asumiendo que hicimos el WORKDIR /app/)

Pero salvo que necesitemos específicamente las funcionalidades que ofrece la instrucción ADD, conviene siempre utilizar COPY, ya que es más preciso y más simple de usar.

## Excluyendo archivos

Existen 2 razones por las que puedes querer ignorar archivos.

1. Toma menos tiempo construir imágenes.
2. Las imágenes creadas pesan menos.

y ¿Qué queremos ignorar de nuestro proyecto de prueba? el archivo node_modules

Para ello creamos un archivo nuevo en la raíz de nuestro proyecto y le vamos a llamar "***.dockerignore***" y en él vamos a escribir node_modules/ para indicar que queremos ignorar ese directorio.

y ¿Ahora cómo ejecuto mi aplicación?

Con la instrucción de RUN npm install nosotros vamos a instalar las dependencias del node_modules.

Con la instrucción de npm run dev nosotros vamos a ejecutar nuestra aplicación en modo de desarrollo, y tendríamos que ejecutar: **docker run app-react npm run dev** → app-react, recordémos, es el nombre que le dimos a la imagen. Sin embargo, escribir el comando npm run dev cada vez que queremos ejecutar nuestra aplicación es tedioso, quiero que lo haga docker. ¿Cómo lo hago? con el comando CMD npm run dev, que lo vamos a incluir en nuestro Dockerfile.

Pero hay dos formas de escribir ese comando.

1. CMD npm run dev → esto crea una nueva shell para ejecutar nuestra aplicación.
2. CMD ["npm", "run", "dev"] → esto se ejecuta dentro del mismo contenedor, y por lo tanto, va a consumir menos recursos.

¿Cuándo se usa RUN y cuándo se usa CMD?

- RUN se utiliza cuando se están construyendo las imágenes.
- CMD se utiliza cuando se ejecutan los contenedores.

También se puede utilizar ENTRYPOINT ( ENTRYPOINT [”npm”, “run”, “dev”]), que hace lo mismo que CMD, pero con la diferencia de que si nosotros queremos reemplazar esa instrucción al momento de ejecutar nuestro contenedor, explícitamente debemos indicarle —entrypoint, y esto con CMD no es necesario.

Ejemplo:

docker run app-react echo hola → Con CMD no es necesario indicarle nada, simplemente lo reemplazo y ya.

docker run app-react —entrypoint echo hola → Con ENTRYPOINT debo indicarle explícitamente que quiero reemplazarlo.

## Puertos

Para exponer el puerto con el que estamos trabajando, simplemente hacemos:
EXPOSE 5173
¿Lo expone realmente? No. En realidad le estamos diciendo a Docker que, eventualmente, ese puerto será expuesto.

## Usuarios

Por defecto las imágenes se crean con usuario root. Esto es un riesgo para la seguridad. ¿Cómo lo podemos mitigar? Pues creando un nuevo usuario que solamente se va a encargar de nuestra aplicación, y nada más. No va a tener contraseña. No vamos a querer iniciar sesión con él.

Para ello vamos a tener que modificar nuestro archivo Dockerfile.

Luego de que le indicamos la imágen en la cual nosotros nos vamos a basar para crear la nuestra, debemos hacer:

RUN addgroup react && adduser -S -G react react

USER react

¿Qué debemos tener en cuenta para lograr esto? Tenemos que cambiar el propietario de nuestro directorio app/, y ese propietario tiene que ser react. ¿Por qué? pues no podríamos aplicar cambios en nuestra aplicación, porque root sigue seguiría siendo su propietario, entonces... los comandos que no sean RUN, CMD o ENTRYPOINT van a fallar, porque docker va a ejecutarlos con el usuario root.

Por lo tanto, en cada COPY, yo debo agregar ***—chown=react*** para poder cambiar el propietario de los archivos de mi aplicación que yo estoy copiando y sobre el directorio en el cual los estoy copiando.

## Eliminando imágenes

Con el comando docker images → vamos a ver todas las imágenes que fueron creadas, y tal vez...veamos algunas que tienen la etiqueta <none>. ¿Cómo las podemos eliminar?

Si nosotros ejecutamos docker image prune (que es para eliminar todas las imágenes inutilizadas) nos va a decir que no se ha borrado nada, y esto es porque estas imágenes están vinculadas a contenedores creados actualmente (no están en ejecución, están creados). Entonces, ¿de qué manera las podemos eliminar?

Pues tendríamos que ejecutar docker container prune → elimina todos los contenedores que no se están utilizando.

Volvemos a ejecutar docker image prune → y nos va a eliminar todas esas imágenes.

En el caso de que queramos borrar una imagen en particular, o más de una por ejemplo, podríamos hacer:

docker image rm app-react ubuntu

## Etiquetas

Hay 2 formas de asignarle una etiqueta a una imagen:

1. Al momento de crearla: docker build -t app-react:1.0.0
2. Luego de crearla: docker image tag app-react:1.0.0 app-react:1.0.1 (le corregimos un bug por ejemplo).

Ahora bien, la segunda opción, lo único que va a hacer es crearnos una imagen, con el mismo ID que ya tenía...pero con la etiqueta nueva. Si nosotros queremos una nueva imagen, deberíamos hacer el cambio que pretendíamos hacerle a la aplicación, por ejemplo, corregir un bug, y crear una nueva imagen con la etiqueta actualizada, y ahí sí vamos a estar creando una nueva imagen con otro ID.

# Publicando imágenes

Para ello vamos a tener que crearnos una cuenta en Docker Hub.
https://hub.docker.com

Una vez dentro, nos dirigimos al panel de administración en donde vamos a tener que crear un nuevo repositorio en el cual nosotros vamos a ir almacenando las imágenes creadas con Docker.

Para crear un repositorio es bastante intuitivo Docker Hub, pero básicamente te va a pedir un nombre al repositorio, una descripción que es opcional, la visibilidad que es pública o privada (en el plan gratuito de Docker, solo tenemos capacidad para utilizar 1 solo repositorio privado)

Ahora para subir nuestra imagen al repositorio que acabamos de crear, debemos seguir las prácticas vistas en la clase de Etiquetas.

docker image tag app-react:1.0.0 gcarzolio/app-react:1.0.0

¿Es la única manera de compartir nuestras imágenes? No, veamos una alternativa con docker image save.

docker image save -o app-react.tar gianfranco259/app-react:1.0.0

y luego se lo compartiríamos a otro desarrollador que tendría que abrir esa imagen en Docker de la siguiente manera:

docker image load -i app-react.tar → la opción -i nos permite cargar una imagen desde un archivo comprimido.

# Contenedores

Para ejecutar contenedores hemos estado utilizando el comando de docker run, que es la combinación de otros 3 comandos.

docker run comprende:

1. docker pull → que descarga la imagen siempre y cuando no se encuentre local
2. docker create → nos permite crear una imagen en base a un Dockerfile
3. docker start → inicia el contenedor

Para crear un contenedor con un nombre específico, y no el automático que le pone Docker, debemos hacer:

docker create —name micontenedor gianfranco259/app-react:1.0.0

docker start micontenedor → para iniciar el contenedor.

docker stop micontenedor → para detener el contenedor.

docker run -d —name micontenedor app-react:1.0.0 → estamos haciendo las 3 instrucciones juntas, además de darle un nombre específico (cabe aclarar que no pueden existir 2 contenedores con el mismo nombre) y la opción -d nos sirve para que Docker nos devuelva el control de la terminal inmediatamente después de crear y ejecutar el contenedor.

docker logs → para acceder a los logs.

si hacemos docker logs —help → vemos todas las opciones que tiene.

docker container rm mico → para eliminar uno o más contenedores que están detenidos.

En el caso de que el contenedor, o los contenedores, que deseamos eliminar se encuentran corriendo, tenemos dos opciones:

1. Detener el contenedor y eliminarlo con la instrucción que ya vimos.
2. Ejecutar docker container rm -f mico

docker exec -it mico sh → para entrar en un contenedor que está corriendo.

docker exec mico ls → es para listar todo el contenido del directorio de trabajo del contenedor sin acceder a él directamente

docker ps -a → nos lista todos los contenedores (sin importar si están corriendo o no)

docker container ls -a → hace lo mismo

docker ps -aq → hace lo mismo, y con la opción -q nos muestra solamente los id’s de esos contenedores.

docker container ls -aq → hace lo mismo que el anterior

docker container -rm -f $(docker container ls -aq) → nos elimina todos los contenedores (sin importar si están corriendo o detenidos)

La misma lógica de borrado se puede aplicar con las imágenes.

# Volúmenes

¿Qué son? un enlace entre la máquina host y el contenedor.

No necesariamente tiene que ser nuestra máquina, puede ser un servicio u otra máquina en la red, etc.

Aquí es donde vamos a guardar los datos que nos importa almacenar.

docker volume → si lo ejecuto así puedo ver todas los comandos que puedo ejecutar con él.

docker volume create datos → creo un volumen.

docker run -d -p 3000:3000 -v datos:/app/datos app-react:1.0.0

¿Qué hice?

Estoy creando y ejecutando un contenedor, basado en la imagen que habíamos creado anteriormente, en modo dettach, mapeando el puerto 3000 y creando un volumen llamado datos en el directorio /app

Sin embargo, por defecto, docker me va a crear este volumen con el usuario root como propietario. Y ya vimos por qué esto no es una buena práctica.

Tenemos que modificar el Dockerfile. Luego de la instrucción de WORKDIR /app/ vamos a escribir RUN mkdir datos

Sin embargo... como modificamos el Dockerfile, hay que crear una nueva imagen.

docker build -t app-react:1.0.1 . → creamos la nueva imagen

Detengo cualquier contenedor que esté corriendo y creo uno nuevo basado en esta nueva imagen que creamos.

docker run -d -p 3000:3000 —name mico -v datos:/app/datos app-react:1.0.1

Si nosotros no creamos el volumen antes de crear y ejecutar este contenedor como vimos recién, Docker lo va a hacer por nosotros y si no creamos el directorio datos dentro de /app, también lo va a hacer Docker por nosotros, pero como observamos, lo hace con el usuario root por defecto. Es por eso que modificamos el Dockerfile.

## Compartiendo código

Cuando nosotros nos encontramos trabajando con Docker vamos a tener 2 posibles instancias.

1. Producción → siempre vamos a generar una nueva imagen si hacemos cambios en el código fuente de nuestro proyecto.

2. Desarrollo → usaremos los volúmenes para vincular nuestro proyecto en entorno de desarrollo con nuestro contenedor y ver los cambios en tiempo real.

¿Cómo lo realizamos? usando los volúmenes, como dijimos, pero anónimos.

docker run -d -p 80:5173 —name mico -v ./src:/app/src app-react:1.0.1

Nuestro volumen no tiene un nombre, es anónimo, y le estamos diciendo que genere un enlace entre la carpeta /src original de mi proyecto con la carpeta /src que está dentro de /app y que sería la copia que nosotros indicamos en el Dockerfile, para poder ver los cambios en tiempo real que nosotros le aplicamos a nuestro proyecto.

Hay que ser específico con los archivos y directorios que queremos vincular. No vamos a vincular TODO el proyecto.

y ¿Para qué es todo esto? para acelerar el desarrollo cuando estamos trabajando con Docker.

## Copiando archivos desde Docker al SO anfitrión y viceversa

Habrá veces en las que quieras obtener un archivo del contenedor para analizarlo con otra herramienta por ejemplo, que no está dentro de Docker, entonces lo quieres copiar y pegar en tu SO anfitrión. ¿Cómo hacemos eso?

Asumiendo que tenemos un contenedor corriendo y que tiene un archivo usuarios.txt, ejecutamos:

docker cp <ID_contenedor>:/app/usuarios.txt . → cp es para copiar, luego utilizamos el id del contenedor + la ruta de donde se encuentra el archivo que queremos copiar y finalmente el . es para indicar que vamos a pegarlo en  el directorio actual de mi SO anfitrión, supongamos... directorio workspace

y ¿al revés?

docker cp texto_anfitrion.txt <ID_contenedor>:/app/ → primero debemos asegurarnos de estar en el directorio donde se encuentra el archivo (texto_anfitrion.txt) que queremos copiar de nuestra máquina o, al menos, conocer bien su ruta. Luego le indicamos el id del contenedor + la ruta en la cual vamos a pegar el archivo de origen.

## Trabajando con docker.compose.yml

Este archivo de configuración nos va a servir configurar todos esos comandos que vimos anteriormente en un solo archivo, el docker.compose.yml

Luego, con docker compose up vamos a poder crear e iniciar los contenedores que definimos en el archivo de configuración.

¿Opciones?¿Comandos? muchísimos.

Ejecutas docker compose y te va a indicar todo lo que podés hacer para buscar más información, sobre ciertos comandos, qué opciones aceptan, etc.

Los más comunes son docker compose up, que ya vimos lo que hace, y docker compose down que hace todo lo contrario, detiene y elimina los contenedores.

¿Cómo se configura el archivo docker.compose.yml? escribirlo acá es muy tedioso, pero tenemos uno ya configurado en el proyecto de ejemplo que vimos en clase.

## Redes en Docker

Ejecutamos docker network y vemos todas las opciones que éste nos ofrece.

La que más nos importa ahora es docker network ls → para listar todas las redes que se encuentran en nuestra máquina con Docker.

Se crean 3 redes por defecto.

1. bridge con el driver de bridge. Es una red que se crea para los contenedores que se encuentran dentro de la misma red. Esto les permite a esos contenedores comunicarse entre sí sin ningún problema.
2. host con el driver de host. Cuando utilizamos este driver dejamos de utilizar la implementación de redes de Docker y en su lugar adoptamos la implementación de nuestro SO anfitrión.
3. none con el driver de null. La máquina que se encuentre dentro de esta red no se puede comunicar con nadie.

Existe otro tipo de red que se llama Overlay, y esto permite que 2 o más máquinas que estén ejecutando Docker puedas ponerlas en red y de esta manera podrán comunicarse contenedores que se encuentran en máquinas diferentes. Este tipo de red se utiliza cuando empieces a desplegar tus aplicaciones o contenedores en Kubernetes.

## Desplegando nuestras aplicaciones

Herramientas para realizar despliegues en la nube:

1. Docker Swarm → No se está utilizando. Nos permite a nosotros gestionar múltiples máquinas al mismo tiempo y de esta manera nosotros vamos a poder desplegar nuestras aplicaciones en múltiples servidores.
2. Kubernetes o K8S (es otro nombre que le dan a Kubernetes) → Mucho más completa que Docker Swarm. Al igual que ésta última nos va a permitir a nosotros desplegar nuestros contenedores en múltiples servidores.
3. Despliegue simple → Esto quiere decir que vamos a contratar un servidor, vamos a ingresar en él, vamos a descargar el código de nuestro proyecto y vamos a ejecutar docker compose up (o en el caso de que sea una sola imagen y un solo contenedor tranquilamente con docker run es suficiente).
4. Despliegue de aplicación → Depende de la plataforma que hayamos decidido contratar. Básicamente le vamos a indicar una imagen para que el servicio la descargue y se va a encargar de desplegarla automáticamente sin tener que hacerlo nosotros. Esta opción solamente te permite desplegar una imagen o un contenedor por servidor.

La realidad, es que a nosotros para empezar nos conviene utilizar la opción 3: Despliegue simple.

¿Qué opciones de despliegue tenemos?

1. AWS
2. Azure
3. Google Cloud
4. DigitalOcean → Si bien esta tiene un panel de administración mucho más simple y sencillo, no implica que sea menos poderosa que las otras 3. Esta opción, para comenzar, está más que bien.

## Creación de llaves SSH

Encriptan las comunicaciones entre nuestra máquina y el servidor. También la vamos a utilizar para comunicarnos con GitHub.

¿Qué significan “llave pública” y “llave privada”?

Pues que la llave pública es la que nosotros vamos a compartir, incluido el servidor. Básicamente quien tenga esta llave podrá encriptar toda la información que envíe. Por otro lado, solamente quien tenga la llave privada

podrá desencriptar esa información. Ésta última es la que NO vamos a compartir.

Para nosotros poder crear este par de llaves pública y privada, vamos a ejecutar el siguiente comando:

ssh-keygen -t ed25519 -C “gianfranco.carzolio@gmail.com” → básicamente con el -t ed25519 le estamos diciendo con qué algoritmo vamos a encriptar las llaves. La opción -C es para un comentario, que en este caso es nuestro correo.

Luego tenemos que ingresar para ver la clave pública que es la que vamos a compartir (tiene extensión .pub). Dónde se guardan, es distinto en MacOS y Linux que en Windows. Generalmente el archivo es id_ed25519.pub → porque nosotros usamos ese algoritmo como ejemplo. Eso lo copiamos, todo, y luego lo vamos a ingresar en GitHub.

¿Qué tenemos que hacer en GitHub? Crear un nuevo repositorio (público o privado, en este caso privado). Luego vas a la opción de SSH and GPG keys o similar, vas a crear una nueva ssh key, y allí vas a pegar el valor que copiaste de la llave pública, de nombre utiliza el nombre de tu computadora. Una vez que ya la tienes copiada, lo que vas a hacer es subir a ese repositorio todo el código de tu proyecto (solo los que importan para el proyecto).

git init (dentro del directorio del proyecto)

git add .gitignore backend/ docker-compose.yml/ frontend/

git commit -m “commit inicial”

git branc -M main

git remote add origin git@github.com:gcarzolio/mirepodocker (esto es para indicar el origen de donde nosotros vamos a copiar y subir código), puede ser distinto el nombre de usuario, ajusta eso al momento de ejecutar la instrucción.

y ahora sí ejecutamos git push -u origin main

¿Qué tenemos que hacer en DigitalOcean? Similar. Vamos a crear un nuevo Droplet desde el Marketplace (hay otro opción, pero no vamos a usar esa). Luego te va a pedir una región, que esta región es donde se encuentran los usuarios que van a utilizar tu aplicación. En el caso de que no se encuentre la región de tus usuarios, utiliza la región más cercana.

Desde allí es bastante intuitivo lo que hay que completar, depende sí, de nuestras necesidades particulares, en este caso muchas configuraciones las vamos a dejar por defecto ya que se trata de una prueba. Lo que sí, es importante mencionar, es que nos va a permitir elegir de qué manera nos queremos conectar a nuestra máquina (ssh key o password). Lo más seguro es crear una nueva ssh key, y allí vas a pegar el valor copiado de la llave pública y le vas a dar el nombre de tu computadora.

Luego, como dije tiene otras configuraciones pero eso va a depender de nuestras necesidades. En nuestro caso, vamos a dejar todo por defecto, y por último creamos el Droplet.

De vuelta en el panel de administración, vamos a entrar en el Droplet que acabamos de crear (tal vez tengamos que esperar unos minutos que se termine de configurar antes de modificar cualquier cosa o entrar). Le vamos a asignar una IP estática. También es muy intuitivo la manera de asignarla. La copiamos porque la vamos a usar ahora.

Ya de regreso en nuestra terminal, en la rama main, vamos a ejecutar:

ssh root@<IP_ASIGNADA>

Ahora debemos crear un par de llaves pública y privada en nuestro servidor.

ssh-keygen -t ed25519 -C “gianfranco.carzolio@gmail.com”

cat ~/.ssh/id_ed25519.pub → porque recordemos que el Droplet es basado en Linux. Copiamos el valor.

Debemos entrar a GitHub nuevamente y crear una Deploy key, le damos un nombre y debemos pegar el valor de la llave pública que creamos en el servidor.

git clone git@github.com:gcarzolio/..... → clonamos el proyecto.

docker compose up → Aquí vamos a tener errores al intentar ejecutarlo.

¿Por qué? Porque nuestro archivo docker-compose.yml cambió.

¿Qué cambio? Primero cambió el nombre del directorio, ahora debe apuntar al repositorio creado en GitHub. Segundo, nuestro contenedor de api-tests se basa en la imagen de nuestra app, y eso es un problema porque Docker intentará crear todas las imágenes al mismo tiempo. Tercero, nuestra variable de entorno VITE_API_URL, también cambió, ya no es localhost sino que ahora es esa IP que habíamos asignado a nuestra máquina en DigitalOcean.

¿Qué podemos hacer? Pues, como no queremos borrar nuestros tests por un lado, pero no queremos subir estos a producción por el otro, lo que podemos hacer es crear un nuevo archivo docker-compose.prod.yml (esta es una convención que puede utilizarse o no) para que, al momento de levantar nuestro proyecto, docker apunte a este nuevo archivo para producción.

Realizamos las modificaciones correspondientes a este nuevo archivo, y luego nos aseguramos de guardarlo.

Ahora debemos hacer,

git add docker-compose.prod.yml

git commit -m “agregamos archivo compose de prod”

git push

Excelente, ahora sí vamos a la terminal del servidor.

git pull → para traernos la última versión del código.

docker compose -f up —build docker-compose.prod.yml → —build para asegurarnos de que construya las imágenes, -f para indicar de qué archivo queremos que levante el proyecto.

Un cambio interesante, que podríamos aplicar a nuestro proyecto sería que tengamos variables de entorno para ambientes de desarrollo y otras para producción.

Eso lo podríamos hacer de la siguiente manera:

1. Comentamos la línea del archivo .env o directamente eliminamos el archivo.
2. Modificamos el archivo docker-compose.yml (que es el que usaríamos en ambiente de desarrollo) y luego del servicio de app, agregamos la línea de environment, VITE_API_URL con el valor de localhost.
3. Modificamos el archivo docker-compose.prod.yml (que es el que usamos en producción) y luego del servicio de app, agregamos la línea de environment, VITE_API_URL con el valor de la IP asignada a nuestra máquina.

agregamos los cambios a git,

git add docker-compose.prod.yml docker-compose.yml frontend/.env

git commit -m “variables de entorno modificadas”

git push

De regreso en el servidor,

git pull

docker compose -f up docker-compose.prod.yml —build