$ 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)

$ docker run ubuntu (corre un ubuntu pero lo deja apagado)  
$ docker ps -a (lista todos los contenedores)  
$ docker -it ubuntu (lo corre y entro al shell de ubuntu)  
-i: interactivo  
-t: abre la consola

<h1>cat /etc/lsb-release (veo la versión de Linux)</h1>

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

$ docker ps -a (veo todos los contenedores)  
$ docker --name <nombre> -d ubuntu -f <comando>  
$ docker --name alwaysup -d ubuntu tail -f /dev/null (mantiene el contenedor activo) docker run -d --name proxy nginx  
$ docker exec -it alwaysup bash (entro al contenedor)  
$ docker inspect --format ‘{{.State.Pid}}’ alwaysup (veo el main process del ubuntu)

docker **stop** <container\_id **or** container\_name>

$ 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)

docker logs -f proxy

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)

*shows dbs (listo las BBDD)  
use platzi ( creo la BBDD platzi)  
db.users.insert({“nombre”:“guido”}) (inserto un nuevo dato)  
db.users.find() (veo el dato que cargué)  
$ docker run -d --name db -v <path de mi maquina>:<path dentro del contenedor(/data/db mongo)> (corro un contenedor de mongo y creo un bind mount)*

docker run -d --name db -v /c:/Users/akasha/Desktop/Docker/data-db:/data/db mongo

Comandos:

$ docker volume ls (listo los volumes)  
$ docker volume create dbdata (creo un volume)  
$ docker run -d --name db --mount src=dbdata,dst=/data/db mongo (corro la BBDD y monto el volume)  
$ docker inspect db (veo la información detallada del contenedor)  
$ mongo (me conecto a la BBDD)

*shows dbs (listo las BBDD)  
use platzi ( creo la BBDD platzi)  
db.users.insert({“nombre”:“guido”}) (inserto un nuevo dato)  
db.users.find() (veo el dato que cargué)*

docker run -d --name db --mount src=dbdata,dst=//data/db 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

Acerca de el comando tail

Por default, éste comando imprime las últimas 10 líneas de un archivo en el sistema operativo GNU/Linux y luego termina.  
al pasarle -f, indicamos que se “vigilen” los cambios en el archivo que le pasemos.

Acerca de /dev/null

Se le conoce como una “caja negra” que no contiene ni hace nada.  
para entender mejor el uso de éste fichero podemos analizar los siguientes casos:  
Si ejecutamos el comando

cat /dev/**null**

No se producirá salida o mensaje

Por lo cual es normalmente utilizado para recibir archivos basura (descartables), o para vaciar archivos mediante redirección:

ls > /dev/**null**

Manda la lista de archivos de ls a el archivo /dev/null, con lo cual dicha lista desaparece.

Si ejecutamos la siguiente orden:

cat /dev/**null** > Archivo.Awk

Estaríamos borrando el contenido de Archivo.Awk, es decir, dejándolo vacío.

Como nota final, cabe aclarar que no es posible copiar o mover archivos hacia /dev/null

Práctica realizada:

Para crear un archivo en mi máquina:

touch prueba.txt

Crear un contenedor llamado copytest, que corra ubuntu e iniciar un proceso que se mantenga activo:

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

Iniciar una conexión de terminal con el contenedor que está corriendo ubuntu

docker exec -**it** copytest **bash**

Creamos un directorio llamado testing en el contenedor de ubuntu

mkdir testing

Comando docker para copiar un fichero o un directorio desde la máquina anfitrión hacia el contenedor, copytest es el nombre del contenedor hacia el cual queremos copiar el archivo

docker cp prueba.txt copytest:/testing/test.txt

Comando para ejecutar la acción inversa, es decir, copiar fichero o directorio desde el contenedor hacia la máquina anfitrión.  
donde localtesting será el nombre de un nuevo directorio que se creará en la máquina anfitrión dentro del directorio en el que nos encontremos al momento de haber ejecutado el siguiente comando:

docker cp copytest:/testing localtesting

Nota: utilizando docker cp, no es necesario que el contenedor esté corriendo

Comandos:

$ docker image ls (veo las imágenes que tengo localmente)  
$ docker pull ubuntu:20.04 (bajo la imagen de ubuntu con una versión específica)

Comandos:

$ 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:pruebaDockerFile . (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)

Comandos:

$ docker history ubuntu:platzi (veo la info de como se construyó cada capa)  
$ dive ubuntu:platzi (veo la info de la imagen con el programa dive)

$ 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)

$ 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)

docker run --rm -p 3000:3000 -v /c:/Users/akasha/Desktop/Docker/docker/index.js:/usr/src/index.js test-app

Comandos:  
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=mongodb://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”)

Comandos:  
$ docker-compose up -d (crea todo lo declarado en el archivo 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)

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)

Comandos:  
$ docker ps -a (veo todos los contenedores de mi máquina)  
$ docker container prune (borra todos los contenedores inactivos)  
$ docker rm -f $(docker ps -aq) (borra todos los contenedores que estén corriendo o apagados)  
$ docker network ls (lista todas las redes)  
$ docker volume ls (lista todos los volumes)  
$ docker image ls (lista todas las imágenes)  
$ docker system prune (borra todo lo que no se esté usando)  
$ docker run -d --name app --memory 1g platziapp (limito el uso de memoria)  
$ docker stats (veo cuantos recursos consume docker en mi sistema)  
$ docker inspect app (puedo ver si el proceso muere por falta de recursos