

Curso de Docker

Docker es una plataforma de contenedor de software diseñada para desarrollar, enviar y ejecutar aplicaciones aprovechando la tecnología de los contenedores. Docker se presenta en dos versiones: edición empresarial y edición de comunidad.

Un contenedor de Docker es un conocido contenedor ejecutable, independiente, ligero que integra todo lo necesario para ejecutar una aplicación, incluidas bibliotecas, herramientas del sistema, código y tiempo de ejecución. A diferencia de una máquina virtual que proporciona virtualización de hardware, un contenedor proporciona virtualización ligera a nivel de sistema operativo. Los contenedores comparten el núcleo del sistema host con otros contenedores. Un contenedor, que se ejecuta en el sistema operativo host, es una unidad de software estándar que empaqueta código y todas sus dependencias, para que las aplicaciones se puedan ejecutar de forma rápida y fiable de un entorno a otro.

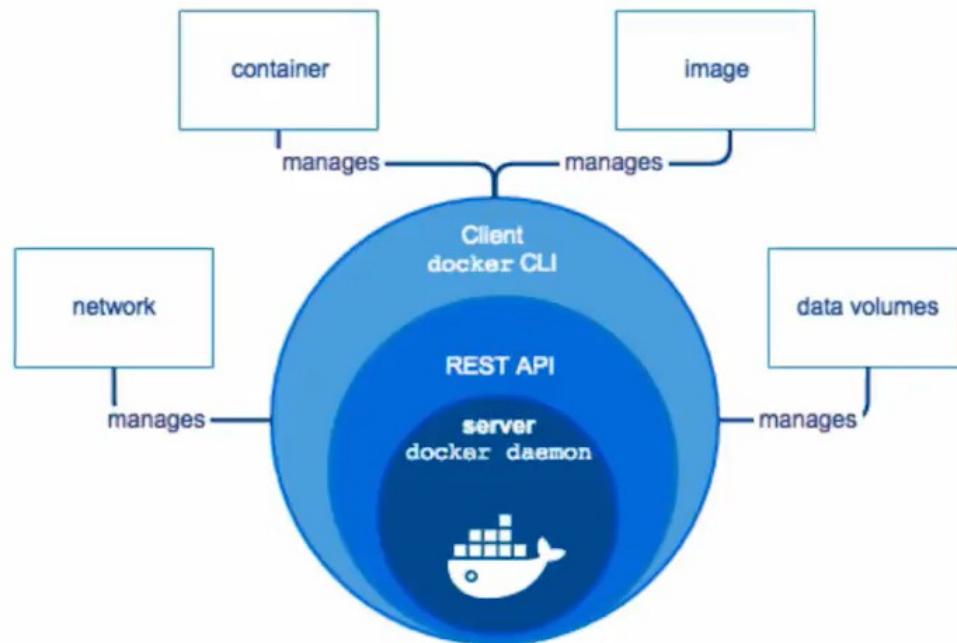
Docker te permite ***construir, distribuir y ejecutar*** cualquier aplicación en cualquier lado.

Distribuir software



Tu código tiene que transformarse en un artefacto, o varios, que puedan ser transportados a donde tengan que ser ejecutados.

Arquitectura de Docker



Una limitación de Docker es que no te permite tener dos contenedores con el mismo nombre en tu misma instalación.

```
docker run hello-world # Corre el contenedor más básico de docker.

docker ps # Ver los contenedores que estamos corriendo en el momento.

docker ps -a # Ver todos los contenedores que se han corrido en la máquina.

docker run --name <name> <image> # colocar un nombre custom al contenedor

docker rename <actual-name> <new-name> # Renombrar un contenedor

docker rm <container-id> ó <container-name> # Borrar un contenedor

docker container prune # Borrar todos los contenedores detenidos

docker stop <container-name> # Detiene el contenedor
```

```
# Eliminar TODOS los contenedores
$ docker rm -f $(docker ps -aq)
```

Más comandos:

- `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
- `cat /etc/lsb-release` (veo la versión de Linux)

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**.

Un main process determina la vida del contenedor, un contenedor corre siempre y cuando su proceso principal este corriendo.

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.

Ejemplos manejados en la clase:

1. **`docker run --name alwaysup -d ubuntu tail -f /dev/null`** La opción `-d` lo hace correr en segundo plano.
2. Te puedes conectar al contenedor y hacer cosas dentro del él con el siguiente comando (sub proceso) **`docker exec -it alwaysup bash`**
3. Se puede matar un Main process desde afuera del contenedor, esto se logra conociendo el id del proceso: **`docker inspect --format '{{.State.Pid}}' alwaysup`**
4. Kill <Id del proceso>
5. Otra forma de detener el contenedor es mediante su nombre: **`docker stop alwaysup`**

Más ejemplos, => 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)

Ejemplo con mongoddb:

1. **docker run -d --name db mongo**
2. **docker exec -it db bash**
3. **mongosh** (Una vez dentro, ya tenemos instalado mongo en el contenedor por lo que procedemos a entrar a mongoddb)
4. **use platzi** (Creamos una db llamada platzi)
5. **db.users.insertOne({"nombre":"Hector"})** (Insertamos un registro)
6. **db.users.find()** (Nos muestra los registros)

```
platzi> db.users.insertOne({"nombre":"Hector"})
{
  acknowledged: true,
  insertedId: ObjectId("64dbfcfa4955b208f1784172")
}
platzi> db.users.find()
[
  { _id: ObjectId("64dbfc9f4955b208f1784171"), nombre: 'Hector' },
  { _id: ObjectId("64dbfcfa4955b208f1784172"), nombre: 'Hector' }
]
platzi> |
```

Si se mata el proceso del contenedor se mata la data que habíamos puesto.
En esta clase, queremos tener una carpeta en nuestro S.O tal que lo que pongamos en nuestro contenedor aparezca también en la carpeta. Y esto lo hacemos con los siguientes comandos.

1. **docker run -d --name db -v /mnt/c/Users/"INSPIRON 7460"/OneDrive/Documentos/platzi/docker/dockerdata/mongodata:/data/db mongo** (Sin embargo, este da un error el cual cierra el proceso principal matando al contenedor. Por lo que en los comentarios de platzi se opto por:)
 1. **docker volume create --name mongodata**
 2. **docker run -d --name db --v mongodata:/data/db mongo**
2. **docker exec -it db bash**
3. **mongosh**
4. **use platzi**
5. **db.users.insertOne({"nombre":"Hector"})**
6. **db.users.find()**
7. **exit**
8. **exit**
9. **docker rm -f db**

Volúmenes

```
toreohm@DESKTOP-06TNPIJ:/mnt/c/Users/INSPIRON_7460/OneDrive/Documentos/platzi/docker/dockerdata/mongodata$ docker volume ls
DRIVER      VOLUME NAME
local       0c130e201bc7c2daa1062cb35f1ee5cd8548975f3c62fdd9def523c5ac53ce5b
local       0da4b3671d9ff7968a9e090abf95eee74ffa70a8633af22a52ea93ca50be509
local       1fc2b25ac5b23a6ec22e93d0a847328e5a536200018d2e48e2f4d5ccc8175f80
local       2bd64edbc40c61353f003440de8c2039de8b6ff026ab5e5e51eb99f4d00acd77
local       2ee5e0b985d90286b239566469762a434d2323c8f3e1858d8b34269f5951f53c
local       74891fa29b406059ae4d2a2adba244c94e725706235561afbc44fc38e302cbab
local       a1b674c2162f3014342be932fe88b8763cc29a4f26759e41e46f3ddc9ad9f6bd
local       ca6cf70e18f53ae39fd768f7a95aee00a22ed3479f5946a6aef101df30f1b4fa
local       mongodata
toreohm@DESKTOP-06TNPIJ:/mnt/c/Users/INSPIRON_7460/OneDrive/Documentos/platzi/docker/dockerdata/mongodata$ |
```

Comandos

- **docker volume create dbdata**
- **docker volume ls**
- **docker run -d --name db --mount src=dbdata,dst=/data/db mongo**
(corro la base de datos y monto el volumen)
- **docker exec -it db bash**
- **mongosh**
- **use platzi**
- **db.user.insertOne({"nombre":"hector", "edad":38})**
- **db.user.find()**
- **exit, exit**
- **docker rm -f db**
- **docker run -d --name db --mount src=dbdata,dst=/data/db mongo**
- **docker exec -it db bash**
- **mongosh**
- **use platzi**
- **db.user.find()**

Los datos que guardé en el contenedor anterior persisten, todo esta data está en un volumen administrado por docker.

```
platzi> db.user.find()
[
  {
    _id: ObjectId("64dc1a1ae2e799edea9b1d79"),
    nombre: 'hector',
    edad: 38
  }
]
platzi>
```

Insertar y extraer archivos de un contenedor

Vamos a crear un contenedor, el cuál desde nuestra maquina queremos copiar un archivo hacia el contenedor.

Comandos:

1. **touch prueba.txt**
2. **docker run -d --name copytest ubuntu tail -f /dev/null**
3. **docker ps**
4. **docker exec -it copytest bash**
5. **mkdir testing ; ls**
6. **exit**
7. **docker cp prueba.txt copytest:/testing/nuevoNombre.txt** (Le cambiamos el nombre del archivo, aunque también puede ser el mismo.)
8. **docker exec -it copytest bash**
9. **ls testing/** (Y vemos que el archivo ha sido transferido)

```
root@4850d28a4f57:/# ls testing/  
nuevoNombre.txt  
root@4850d28a4f57:/# |
```

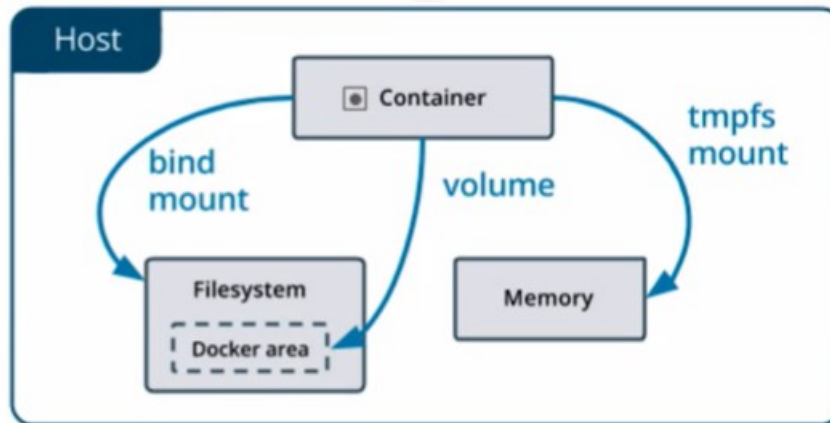
Y ahora vamos extraer el mismo archivo de docker a nuestra maquina con otro nombre.

1. **docker cp copytest:/testing localtesting**
2. **ls -lah**
3. **ls -lah localtesting/**

```
toreohm@DESKTOP-06TNPIJ:/mnt/c/Users/INSPIRON 7460/OneDrive/Documentos/platzi/docker/dockerdata$ docker cp copytest:/testing localtesting  
Successfully copied 2.05kB to /mnt/c/Users/INSPIRON 7460/OneDrive/Documentos/platzi/docker/dockerdata/localtesting  
toreohm@DESKTOP-06TNPIJ:/mnt/c/Users/INSPIRON 7460/OneDrive/Documentos/platzi/docker/dockerdata$ ls -lah  
total 0  
drwxrwxrwx 1 torehm torehm 512 Aug 15 19:03  
drwxrwxrwx 1 torehm torehm 512 Aug 15 19:00  
drwxrwxrwx 1 torehm torehm 512 Aug 15 18:56 localtesting  
drwxrwxrwx 1 torehm torehm 512 Aug 15 17:20 mondata  
-rwxrwxrwx 1 torehm torehm 0 Aug 15 18:51 prueba.txt  
toreohm@DESKTOP-06TNPIJ:/mnt/c/Users/INSPIRON 7460/OneDrive/Documentos/platzi/docker/dockerdata$ ls -lah localtesting/  
total 0  
drwxrwxrwx 1 torehm torehm 512 Aug 15 18:56  
drwxrwxrwx 1 torehm torehm 512 Aug 15 19:03  
-rwxrwxrwx 1 torehm torehm 0 Aug 15 18:51 nuevoNombre.txt  
toreohm@DESKTOP-06TNPIJ:/mnt/c/Users/INSPIRON 7460/OneDrive/Documentos/platzi/docker/dockerdata$ |
```

No hace falta que el contenedor este corriendo (activo) para poder usar **docker cp** en él. Con que exista y esté frenado con eso alcanza.

Datos en Docker



Conceptos fundamentales de Docker: imágenes

Una imagen en Docker es un archivo o file que se encuentra compuesto de diversas capas y que se utiliza con el objetivo de ejecutar un código dentro de un contenedor de Docker. Estas imágenes contienen todo el sistema de ficheros inicial en los que se va a basar el container para su funcionamiento.

Una analogía sería que el container es el objeto/instancia y la imagen la clase/plantilla.

De manera que estas imágenes se encargan de actuar como un script o conjunto de instrucciones útiles para construir un contenedor en Docker, así como una plantilla. De la misma forma, una imagen en esta plataforma funciona como un punto de partida cuando el usuario utiliza Docker.

¿De dónde descarga docker las imagenes?

<https://hub.docker.com/>

```
toreohm@DESKTOP-06TNPIJ:/mnt/c/Users/INSPIRON 7460/OneDrive/Documentos/platzi/docker$ docker image ls
REPOSITORY    TAG       IMAGE ID       CREATED        SIZE
nginx          latest    89da1fb6dcb9   2 weeks ago   187MB
mongo          latest    fb5fba25b25a   4 weeks ago   654MB
ubuntu         latest    5a81c4b8502e   7 weeks ago   77.8MB
hello-world    latest    9c7a54a9a43c   3 months ago  13.3kB
toreohm@DESKTOP-06TNPIJ:/mnt/c/Users/INSPIRON 7460/OneDrive/Documentos/platzi/docker$ |
```

Comandos de la clase

- **docker image ls**
- **docker pull ubuntu:22.04** (Me quiero traer una imagen sin correr un contenedor)

- **docker image ls**

```
toreohm@DESKTOP-06TNPIJ:/mnt/c/Users/INSPIRON 7460/OneDrive/Documentos/platzi/docker$ docker image ls
REPOSITORY          TAG             IMAGE ID        CREATED         SIZE
ubuntu               22.04          01f29b872827   12 days ago    77.8MB
nginx                latest         89da1fb6dcb9   2 weeks ago    187MB
mongo                latest         fb5fba25b25a   4 weeks ago    654MB
ubuntu               latest         5a81c4b8502e   7 weeks ago    77.8MB
hello-world          latest         9c7a54a9a43c   3 months ago   13.3kB
toreohm@DESKTOP-06TNPIJ:/mnt/c/Users/INSPIRON 7460/OneDrive/Documentos/platzi/docker$
```

Construyendo una imagen propia

Este proceso de crear imagenes va estar basado en un archivo llamado Dockerfile. ¿Y para que nos sirve una imagen? Para crear contenedores. Y de una imagen podemos crear infinitos contenedores, no hay limite.

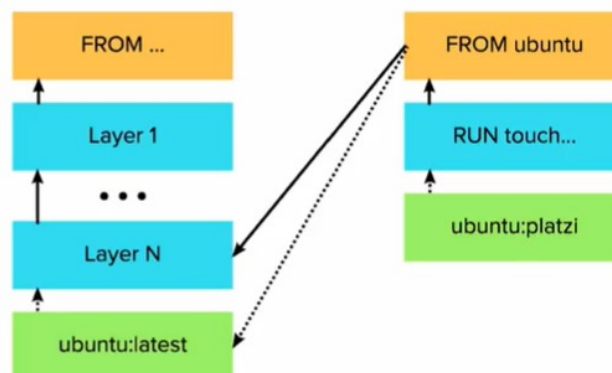
Comandos:

- **mkdir imagenes**
- **cd imagenes**
- **touch Dockerfile**
- **code .** (El comando no me funciono en wsl, se abre manualmente en windows)
- Checar el archivo Dockerfile para ver su contenido

```
Dockerfile X
Dockerfile > ...
1 FROM ubuntu:latest
2
3 RUN touch /usr/src/hola-platzi.txt
```

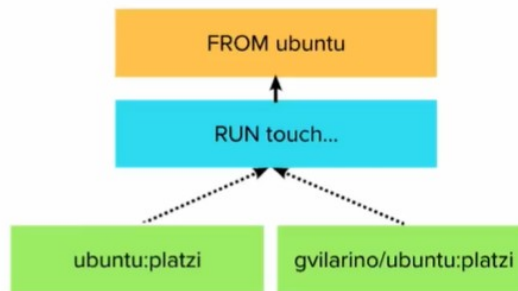
- **docker build -t ubuntu:platzi .**

Las capas de nuestra imagen



- **docker run -it ubuntu:platzi**
- **docker login** (credenciales de hub.docker.com)
- **docker tag ubuntu:platzi <usuario de dockerhub>/ubuntu:platzi** (dueño de la imagen, software, versión)
- **docker image ls** (chechar los cambios)

Retaggear una imagen



- Una vez que estés logeado con tus credenciales de dockerhub, vas a poder publicar a tu repositorio: **docker push <usuario>/ubuntu:platzi**

El sistema de capas

Una imagen de docker es un conjunto de capas; una va debajo de la otra y están ordenadas. Las capas son inmutables. Se puede saber como está hecha una imagen a través de su dockerfile. Podemos entrar a dockerhub y checar la imagen de ubuntu.

Supported tags and respective Dockerfile links

- 20.04 , focal-20230801 , focal
- 22.04 , jammy-20230804 , jammy , latest
- 23.04 , lunar-20230731 , lunar , rolling
- 23.10 , mantic-20230807.1 , mantic , devel

Comandos:

- **docker history ubuntu**

```
toreohm@DESKTOP-06TNPIJ:/mnt/c/Users/INSPIRON 7460/OneDrive/Documentos/platzi/docker/imagenes$ docker history ubuntu
IMAGE          CREATED          CREATED BY          SIZE      COMMENT
5a81c4b8502e   7 weeks ago     /bin/sh -c #(nop)  CMD ["/bin/bash"]   0B
<missing>      7 weeks ago     /bin/sh -c #(nop)  ADD file:140fb5108b4a2861b... 77.8MB
<missing>      7 weeks ago     /bin/sh -c #(nop)  LABEL org.opencontainers... 0B
<missing>      7 weeks ago     /bin/sh -c #(nop)  LABEL org.opencontainers... 0B
<missing>      7 weeks ago     /bin/sh -c #(nop)  ARG LAUNCHPAD_BUILD_ARCH 0B
<missing>      7 weeks ago     /bin/sh -c #(nop)  ARG RELEASE          0B
toreohm@DESKTOP-06TNPIJ:/mnt/c/Users/INSPIRON 7460/OneDrive/Documentos/platzi/docker/imagenes$
```

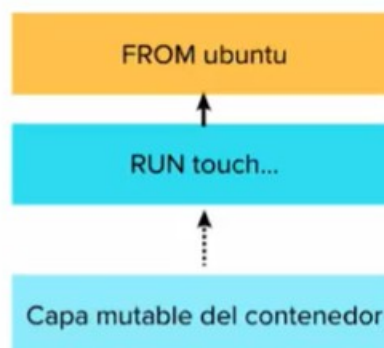
En la columna "CREATED BY" se lee de abajo hacia arriba, y cada una representa una capa de la imagen.

Hay una herramienta que nos ayuda a analizar la composición de las imágenes: **wagoodman/dive**

Cada vez que un contenedor se ejecuta, docker le "ofrece" una capa mutable para que el contenedor lo pueda mutar según sea necesario. (Imagen abajo).

Sin embargo, **las capas de la imagen no pueden ser cambiadas ni por el contenedor ni por nosotros una vez que la hayamos creado.**

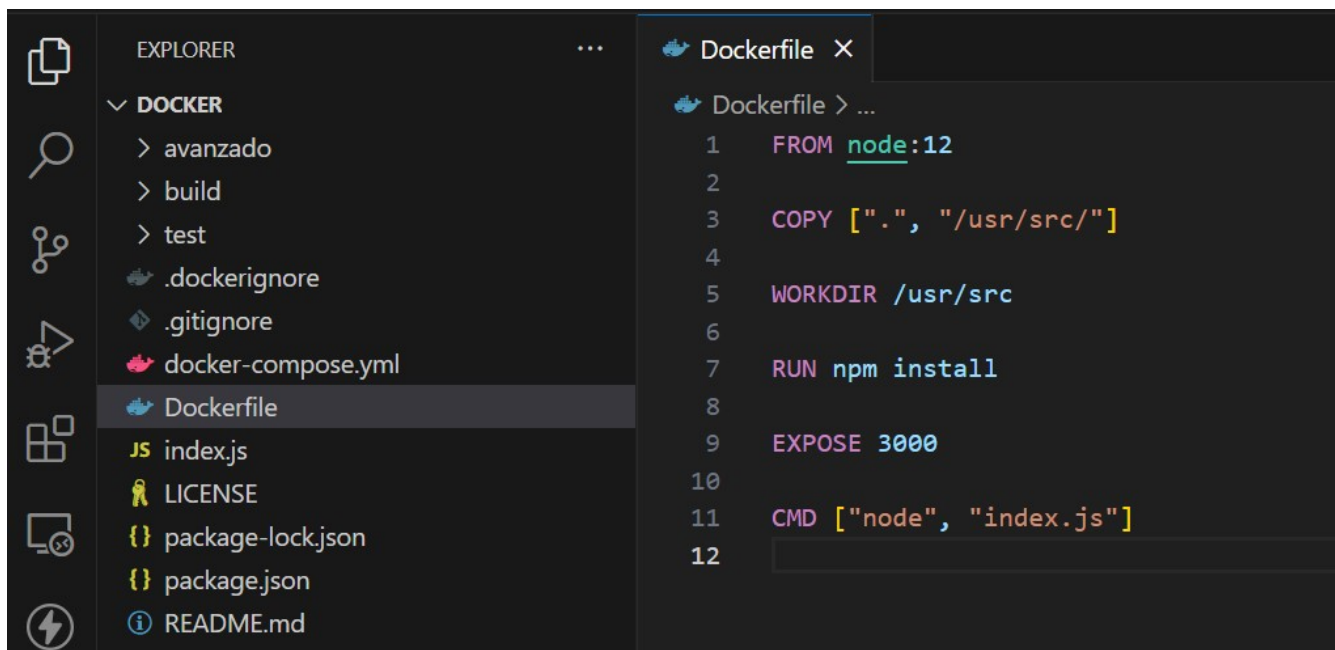
Capas en un contenedor



Usando Docker para desarrollar aplicaciones

Comandos:

- **git clone** <https://github.com/platzi/docker>
- En la imagen de abajo se muestra el **Dockerfile** del proyecto clonado.



- **sudo docker build -t platziapp .**

Vemos que ya se ha creado la imagen.

```
toreohm@DESKTOP-06TNPIJ:/mnt/c/Users/INSPIRON 7460/OneDrive/Documentos/platzi/docker/docker$ docker image ls
REPOSITORY          TAG             IMAGE ID        CREATED         SIZE
platziapp            latest          0e463eb50100   About a minute ago  931MB
ubuntu               platzi         fd0632c6f71d   30 minutes ago   77.8MB
ubuntu              22.04         01f29b872827   12 days ago     77.8MB
nginx               latest         89da1fb6dcb9   2 weeks ago     187MB
mongo               latest         fb5fba25b25a   4 weeks ago     654MB
ubuntu              latest         5a81c4b8502e   7 weeks ago     77.8MB
hello-world         latest         9c7a54a9a43c   3 months ago    13.3kB
toreohm@DESKTOP-06TNPIJ:/mnt/c/Users/INSPIRON 7460/OneDrive/Documentos/platzi/docker/docker$ |
```

Una vez que se tenga la imagen construida, a partir de ésta se puede correr un contenedor.

- **docker run --rm -d -p 3000:3000 platziapp** (creo el contenedor y cuando se detenga se borra, lo publica el puerto 3000 del contenedor y en el puerto 3000 de mi maquina)
- **docker ps -a** (veo los contenedores activos)

Aprovechando el caché de capas para estructurar correctamente tus imágenes

- Modificamos el Dockerfile

```
Dockerfile M X
Dockerfile > ...
1 FROM node:12
2
3 COPY ["package.json", "package-lock.json", "/usr/src/"]
4
5 WORKDIR /usr/src
6
7 RUN npm install
8
9 COPY [".", "/usr/src/"]
10
11 EXPOSE 3000
12
13 CMD ["node", "index.js"]
```

- **sudo docker build -t platziapp .** (Y aquí el problema es que reinstala todo, y nosotros queremos cambiar/agregar código sin reinstalar todo).

```
coreohm@DESKTOP-06TNP13: /mnt/c:/Users/INSPIRON 7460/OneDrive/Documentos/platzi/docker/docker$ sudo docker build -t platziapp .
[+] Building 106.8s (10/10) FINISHED                                docker:default
=> [internal] load build definition from Dockerfile                0.1s
=> => transferring dockerfile: 208B                                0.0s
=> [internal] load .dockerignore                                  0.1s
=> => transferring context: 133B                                    0.0s
=> [internal] load metadata for docker.io/library/node:14        1.3s
=> [1/5] FROM docker.io/library/node:14@sha256:a158d3b9b4e3fa813fa6c8c590b8f0a860e015ad4e59bbce5744d2f6fd8461aa 95.7s
=> resolve docker.io/library/node:14@sha256:a158d3b9b4e3fa813fa6c8c590b8f0a860e015ad4e59bbce5744d2f6fd8461aa 0.1s
=> sha256:2ff1d7c41c74a25258bfa6f0b8adb0a727f84518f55f65ca845ebc747976c408 50.45MB / 50.45MB 14.1s
=> sha256:a158d3b9b4e3fa813fa6c8c590b8f0a860e015ad4e59bbce5744d2f6fd8461aa 776B / 776B 0.0s
=> sha256:2cfa3fbb0b6529e4726b4f599ec27ee557ea3dea7019182323b3779959927f 2.21kB / 2.21kB 0.0s
=> sha256:1d12470fa662a2a5cb50378dc8c8e228c1735747db410bbefb8e2d9144b5452 7.51kB / 7.51kB 0.0s
=> sha256:b253aea7e0671bb6008df01de101a38a045ff7bc656e3b0fbfc7c05cca5 7.86MB / 7.86MB 13.5s
=> sha256:3d2201bd995cccf12851a50820de83d34a17011dcbb9ac9f9df3a50c952cbb131 10.00MB / 10.00MB 13.2s
=> sha256:1de76e268b103d05fa8960e0f77951ff54b912b63029c34f5d6adfd09f5f9ee2 51.88MB / 51.88MB 41.1s
=> sha256:d9a8df5894511ce28a05e2925a75e8a4acb0634c39ad734fd6ba8e23d1b1569 101.85MB / 101.85MB 77.3s
=> extracting sha256:2ff1d7c41c74a25258bfa6f0b8adb0a727f84518f55f65ca845ebc747976c408 7.1s
=> sha256:6f51ee005deac0d99898e41b8ce60ebf250ebe1a31a0b03f613aec6bbcb9b83d8 4.19kB / 4.19kB 14.4s
=> sha256:5f32ed3c3f278edda4fc571c880b5277355a29ae8f52b52cdf865f058378a590 35.24MB / 35.24MB 27.2s
=> extracting sha256:b253aea7e0671bb6008df01de101a38a045ff7bc656e3b0fbfc7c05cca5 1.7s
=> extracting sha256:3d2201bd995cccf12851a50820de83d34a17011dcbb9ac9f9df3a50c952cbb131 1.3s
=> sha256:0c8cc2f24a4dc64e602e086fc9446b0a541e8acd9ad72d2e90df3ba22f158b3 2.29MB / 2.29MB 29.0s
=> sha256:0d27a8e861329007574c6766fba946d48e20d2c8e964e873de352603f22c4ceb 450B / 450B 29.4s
=> extracting sha256:1de76e268b103d05fa8960e0f77951ff54b912b63029c34f5d6adfd09f5f9ee2 19.9s
=> extracting sha256:d9a8df5894511ce28a05e2925a75e8a4acb0634c39ad734fd6ba8e23d1b1569 11.4s
=> extracting sha256:6f51ee005deac0d99898e41b8ce60ebf250ebe1a31a0b03f613aec6bbcb9b83d8 0.0s
=> extracting sha256:5f32ed3c3f278edda4fc571c880b5277355a29ae8f52b52cdf865f058378a590 3.1s
=> extracting sha256:0c8cc2f24a4dc64e602e086fc9446b0a541e8acd9ad72d2e90df3ba22f158b3 0.1s
=> extracting sha256:0d27a8e861329007574c6766fba946d48e20d2c8e964e873de352603f22c4ceb 0.0s
=> [internal] load build context                                0.2s
```

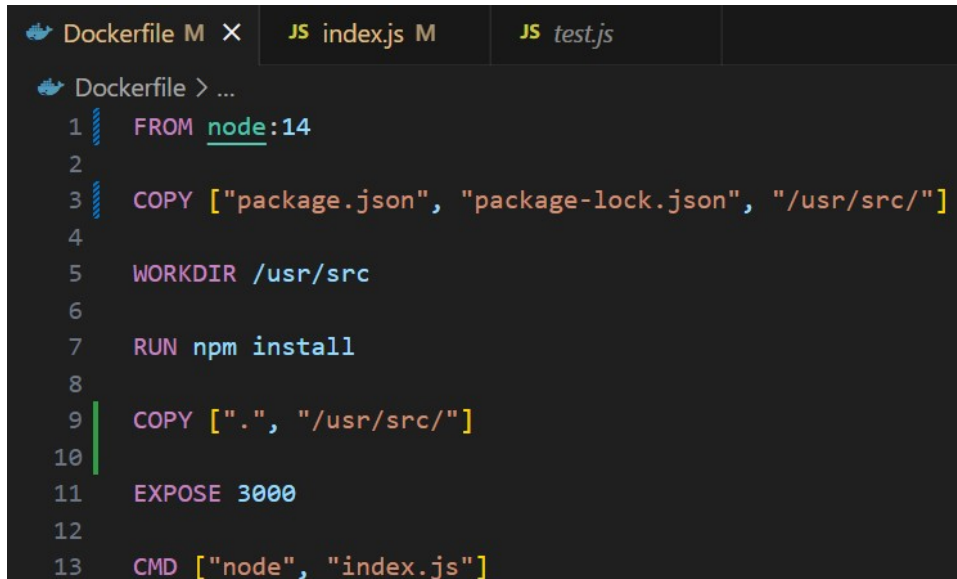
- Para eso volvemos a modificar el archivo Dockerfile

```
Dockerfile M X JS index.js JS test.js
Dockerfile > ...
1 FROM node:14
2
3 COPY ["package.json", "package-lock.json", "/usr/src/"]
4
5 WORKDIR /usr/src
6
7 RUN npm install
8
9 COPY [".", "/usr/src/"]
10
11 EXPOSE 3000
12
13 CMD ["npx", "nodemon", "index.js"]
```

- **docker run --rm -p 3000:3000 "\$(pwd)/index.js":/usr/src/index.js platziapp**

Docker networking: colaboración entre contenedores

Arreglamos nuestro Dockerfile.

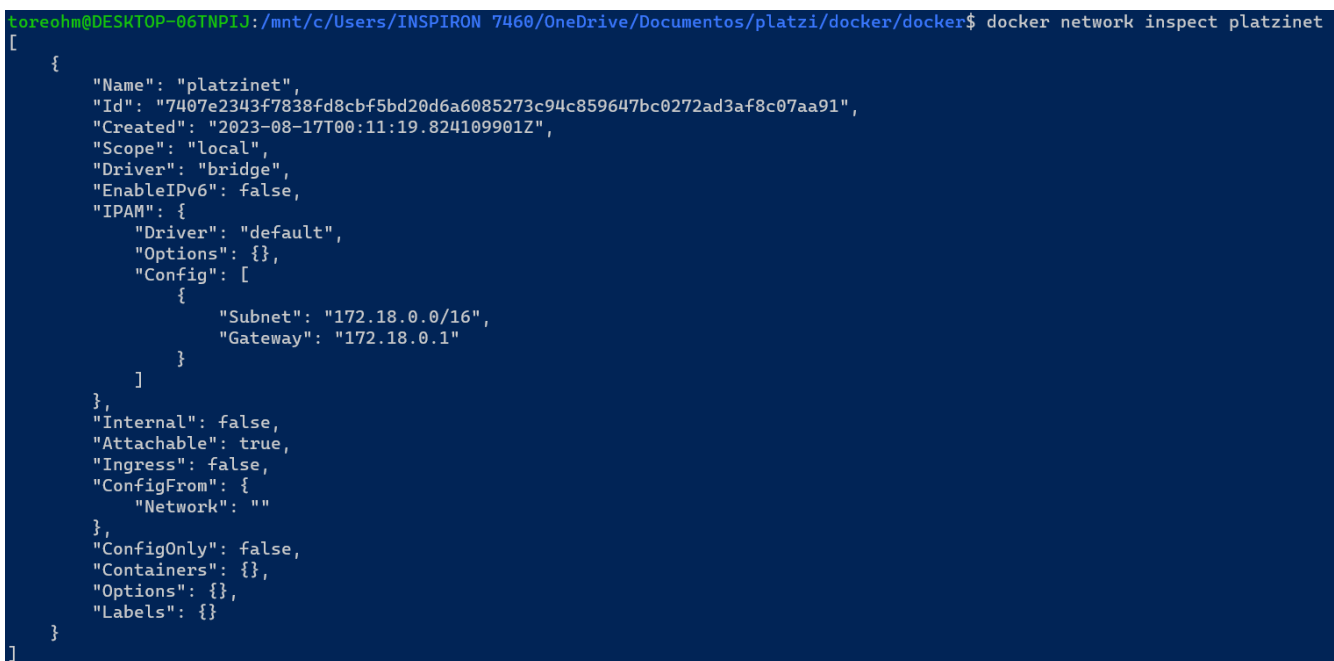


```

1 FROM node:14
2
3 COPY ["package.json", "package-lock.json", "/usr/src/"]
4
5 WORKDIR /usr/src
6
7 RUN npm install
8
9 COPY [".", "/usr/src/"]
10
11 EXPOSE 3000
12
13 CMD ["node", "index.js"]

```

- **docker network ls**
- **docker network create --attachable platzinet**
- **docker network ls**
- **docker network inspect platzinet**



```

toreohm@DESKTOP-06TNPIJ:/mnt/c/Users/INSPIRON 7460/OneDrive/Documentos/platzi/docker/docker$ docker network inspect platzinet
[
  {
    "Name": "platzinet",
    "Id": "7407e2343f7838fd8cbf5bd20d6a6085273c94c859647bc0272ad3af8c07aa91",
    "Created": "2023-08-17T00:11:19.824109901Z",
    "Scope": "local",
    "Driver": "bridge",
    "EnableIPv6": false,
    "IPAM": {
      "Driver": "default",
      "Options": {},
      "Config": [
        {
          "Subnet": "172.18.0.0/16",
          "Gateway": "172.18.0.1"
        }
      ]
    },
    "Internal": false,
    "Attachable": true,
    "Ingress": false,
    "ConfigFrom": {
      "Network": ""
    },
    "ConfigOnly": false,
    "Containers": {},
    "Options": {},
    "Labels": {}
  }
]

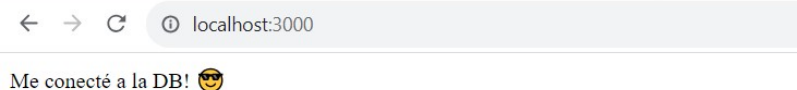
```

Ahora nos falta crear dos contenedores: Uno de la base de datos y otro de la aplicación.

- **docker run -d --name db mongo**
- **docker ps** (cheamos que el container este corriendo)
- Ahora conectamos nuestro contenedor de base de datos a la red creada (**platzinet**). **docker network connect platzinet db**
- **docker network inspect platzinet**

```
toreohm@DESKTOP-06TNPIJ:/mnt/c/Users/INSPIRON 7460/OneDrive/Documentos/platzi/docker/docker$ docker network inspect platzinet
[
  {
    "Name": "platzinet",
    "Id": "7407e2343f7838fd8cbf5bd20d6a6085273c94c859647bc0272ad3af8c07aa91",
    "Created": "2023-08-17T00:11:19.824109901Z",
    "Scope": "local",
    "Driver": "bridge",
    "EnableIPv6": false,
    "IPAM": {
      "Driver": "default",
      "Options": {},
      "Config": [
        {
          "Subnet": "172.18.0.0/16",
          "Gateway": "172.18.0.1"
        }
      ]
    },
    "Internal": false,
    "Attachable": true,
    "Ingress": false,
    "ConfigFrom": {
      "Network": ""
    },
    "ConfigOnly": false,
    "Containers": {
      "71aa76f6229905ed9210d041a3a5de74f47ce40c4588d8483f0377b88715310e": {
        "Name": "db",
        "EndpointID": "c1ccb3ce36e15bae575228d4f1f04f178e11da7c2dee9e039515a5abf035392a",
        "MacAddress": "02:42:ac:12:00:02",
        "IPv4Address": "172.18.0.2/16",
        "IPv6Address": ""
      }
    },
    "Options": {},
    "Labels": {}
  }
]
```

- **docker run -d --name app -p 3000:3000 --env MONGO_URL=mongodb://db:27017/test platziapp** (Notamos que en la url de mongodb basta con poner el nombre del contenedor **db**. La red creada de docker lo reconoce y lo conecta con el contenedor de la app).
- **docker network connect platzinet app**
- Abrimos el navegador: localhost:3000 y ya funciona



Docker Compose: la herramienta todo en uno

```
Dockerfile M  docker-compose.yml X  JS index.js M
docker-compose.yml
1  version: "3.8"
2
3  services:
4    app:
5      image: platziapp
6      environment:
7        MONGO_URL: "mongodb://db:27017/test"
8      depends_on:
9        - db
10     ports:
11       - "3000:3000"
12
13    db:
14      image: mongo
15
```

En este tipo de archivo los espacios/tabulaciones son muy importantes. Un servicio puede tener uno o más contenedores de la misma imagen.

Comandos:

- **docker-compose up**
- **docker-compose up -d** (Para que se ejecute en segundo plano)
- **docker ps**

```
toreohm@DESKTOP-06TNPIJ:/mnt/c/Users/INSPIRON 7460/OneDrive/Documentos/platzi/docker/docker$ docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED        STATUS        PORTS                    NAMES
ca0957627aa7   platziapp "docker-entrypoint.s..." 5 minutes ago Up 32 seconds  0.0.0.0:3000->3000/tcp   docker-app-1
1bb05bf6a2e0   mongo     "docker-entrypoint.s..." 5 minutes ago Up 33 seconds  27017/tcp               docker-db-1
toreohm@DESKTOP-06TNPIJ:/mnt/c/Users/INSPIRON 7460/OneDrive/Documentos/platzi/docker/docker$
```

Subcomandos de Docker Compose

docker-compose conecta todos los contenedores del mismo servicio a una red.

- **docker network ls**

```
toreohm@DESKTOP-06TNPIJ:/mnt/c/Users/INSPIRON 7460/OneDrive/Documentos/platzi/docker/docker$ docker network ls
NETWORK ID     NAME      DRIVER    SCOPE
78ca11f20008   bridge    bridge    local
fbdd0873230b   docker_default    bridge    local
4c9564e89ef6   host      host      local
38e5da50190f   none      null      local
7407e2343f78   platzinet bridge    local
toreohm@DESKTOP-06TNPIJ:/mnt/c/Users/INSPIRON 7460/OneDrive/Documentos/platzi/docker/docker$ |
```

Se llama docker por el directorio en el que estoy y "default" que significa por defecto.

- **docker network inspect docker_default** (Si inspeccionamos la red, vamos a ver que están conectado ambos contenedores, y por eso es que se pueden “ver” entre si a través de su hostname).

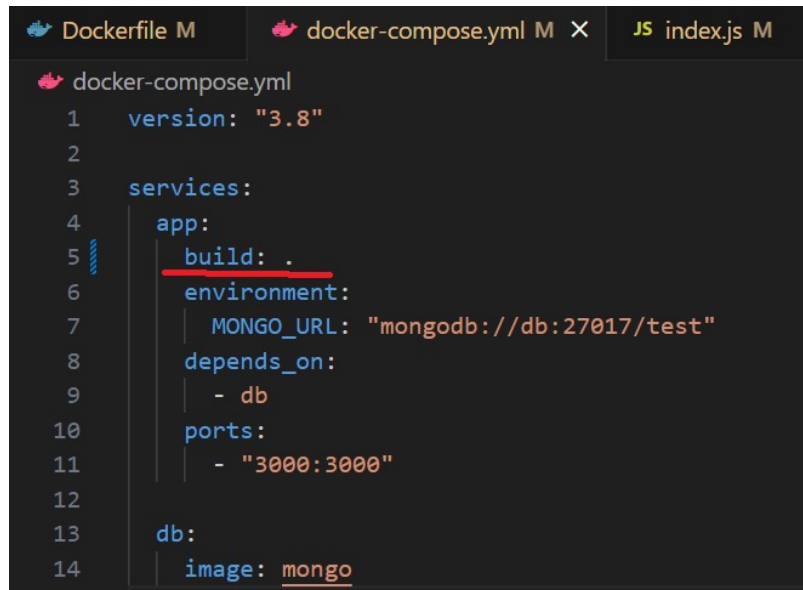
```
toreohm@DESKTOP-06TNPIJ:/mnt/c/Users/INSPIRON 7460/OneDrive/Documentos/platzi/docker/docker$ docker network inspect docker_default
[
  {
    "Name": "docker_default",
    "Id": "fbdd0873230b34b725a9da10504f8542c6744418b1bf2d4cb90be0134ea6a357",
    "Created": "2023-08-17T03:01:09.956593576Z",
    "Scope": "local",
    "Driver": "bridge",
    "EnableIPv6": false,
    "IPAM": {
      "Driver": "default",
      "Options": null,
      "Config": [
        {
          "Subnet": "172.19.0.0/16",
          "Gateway": "172.19.0.1"
        }
      ]
    },
    "Internal": false,
    "Attachable": false,
    "Ingress": false,
    "ConfigFrom": {
      "Network": ""
    },
    "ConfigOnly": false,
    "Containers": {
      "1bb05bf6a2e0d364bb0f55bc7f8ab3d72c80077e26f2cbc0c7177b3b7c1e9d95": {
        "Name": "docker-db-1",
        "EndpointID": "4261a75905c1054c8ab166b026bd4100bedc4b1f530e438133359bcd7d6837f6",
        "MacAddress": "02:42:ac:13:00:02",
        "IPv4Address": "172.19.0.2/16",
        "IPv6Address": ""
      },
      "ca0957627aa79c482b1665ead2de3d6814fe0e45da0916b45fc71c9de01c6e39": {
        "Name": "docker-app-1",
        "EndpointID": "c744ce7c94ace88ffa5162c821546dfcd3f423fa9826a41257fe03a3db755bcd",
        "MacAddress": "02:42:ac:13:00:03",
        "IPv4Address": "172.19.0.3/16",
        "IPv6Address": ""
      }
    },
    "Options": {},
    "Labels": {
      "com.docker.compose.network": "default",
      "com.docker.compose.project": "docker",
      "com.docker.compose.version": "2.20.2"
    }
  }
]
```

- **docker-compose logs** (Podemos ver todos los logs de todos los servicios).
- **docker-compose logs app** (Me muestra los logs de un servicio en específico “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, incluyendo la red)

```
toreohm@DESKTOP-06TNPIJ:/mnt/c/Users/INSPIRON 7460/OneDrive/Documentos/platzi/docker/docker$ docker-compose down
[+] Running 3/3
✓ Container docker-app-1 Removed
✓ Container docker-db-1 Removed
✓ Network docker_default Removed
toreohm@DESKTOP-06TNPIJ:/mnt/c/Users/INSPIRON 7460/OneDrive/Documentos/platzi/docker/docker$
```

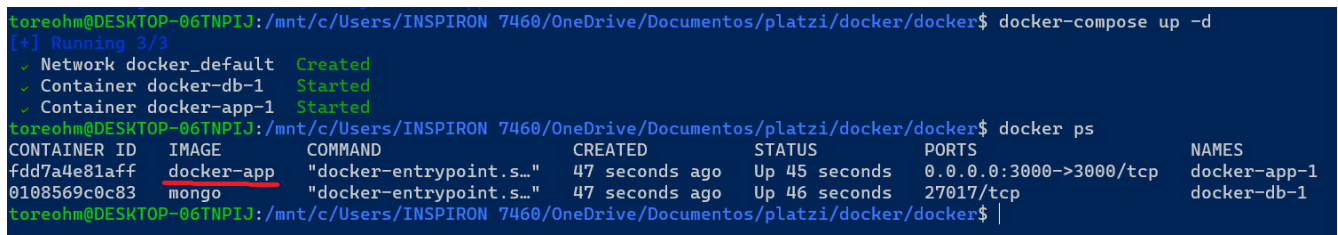

Docker Compose como herramienta de desarrollo

- **sudo docker-compose build** (Para esto tuvimos primero que modificar el archivo docker-compose.yml)



```
docker-compose.yml
1  version: "3.8"
2
3  services:
4    app:
5      build: .
6      environment:
7        MONGO_URL: "mongodb://db:27017/test"
8      depends_on:
9        - db
10     ports:
11       - "3000:3000"
12
13    db:
14      image: mongo
```

- **docker-compose up -d**
- **docker ps** (Y vemos la imagen que docker-compose creo)



```
toreohm@DESKTOP-06TNPIJ:/mnt/c/Users/INSPIRON 7460/OneDrive/Documentos/platzi/docker/docker$ docker-compose up -d
[+] Running 3/3
 ✓ Network docker_default Created
 ✓ Container docker-db-1 Started
 ✓ Container docker-app-1 Started
toreohm@DESKTOP-06TNPIJ:/mnt/c/Users/INSPIRON 7460/OneDrive/Documentos/platzi/docker/docker$ docker ps
CONTAINER ID   IMAGE      COMMAND                  CREATED        STATUS        PORTS                               NAMES
fdd7a4e81aff   docker-app "docker-entrypoint.s..." 47 seconds ago Up 45 seconds 0.0.0.0:3000->3000/tcp             docker-app-1
0108569c0c83   mongo     "docker-entrypoint.s..." 47 seconds ago Up 46 seconds 27017/tcp                          docker-db-1
toreohm@DESKTOP-06TNPIJ:/mnt/c/Users/INSPIRON 7460/OneDrive/Documentos/platzi/docker/docker$
```

Si haces un cambio en el código y quieres que se refleje dentro del contenedor app (no necesariamente en todos los servicios sino al contenedor que le corresponde), puedes hacer eso:

- **sudo docker-compose build app**
- **sudo docker-compose up -d** (Se da cuenta de que hay un cambio en la imagen asociado a este servicio debido al comando anterior. Y va a regenerar el contenedor de app (borrar el anterior y crear uno con la imagen nueva)).

Volvemos a modificar el archivo docker-compose.yml

```
Dockerfile M  docker-compose.yml M  JS index.js M
docker-compose.yml
1  version: "3.8"
2
3  services:
4    app:
5      build: .
6      environment:
7        MONGO_URL: "mongodb://db:27017/test"
8      depends_on:
9        - db
10     ports:
11       - "3000:3000"
12     volumes:
13       - ./usr/src
14   db:
15     image: mongo
```

- Volvemos a ejecutar este comando: **sudo docker-compose up -d** (Y se va a dar cuenta de que hay un cambio).
- Resulta ser que el contenedor se rompe porque estamos montando un directorio que no tiene instalado los módulos de node.

- **docker-compose logs app**

```
toreohm@DESKTOP-06TNPIJ:/mnt/c/Users/INSPIRON_7460/OneDrive/Documentos/platzi/docker/docker$ docker-compose logs app
docker-app-1 | internal/modules/cjs/loader.js:934
docker-app-1 |   throw err;
docker-app-1 |   ^
docker-app-1 |
docker-app-1 | Error: Cannot find module 'express'
docker-app-1 | Require stack:
docker-app-1 |   - /usr/src/index.js
docker-app-1 |     at Function.Module._resolveFilename (internal/modules/cjs/loader.js:931:15)
docker-app-1 |     at Function.Module._load (internal/modules/cjs/loader.js:774:27)
docker-app-1 |     at Module.require (internal/modules/cjs/loader.js:1003:19)
docker-app-1 |     at require (internal/modules/cjs/helpers.js:107:18)
docker-app-1 |     at Object.<anonymous> (/usr/src/index.js:1:17)
docker-app-1 |     at Module._compile (internal/modules/cjs/loader.js:1114:14)
docker-app-1 |     at Object.Module._extensions..js (internal/modules/cjs/loader.js:1143:10)
docker-app-1 |     at Module.load (internal/modules/cjs/loader.js:979:32)
docker-app-1 |     at Function.Module._load (internal/modules/cjs/loader.js:819:12)
docker-app-1 |     at Function.executeUserEntryPoint [as runMain] (internal/modules/run_main.js:75:12) {
docker-app-1 |   code: 'MODULE_NOT_FOUND',
docker-app-1 |   requireStack: [ '/usr/src/index.js' ]
docker-app-1 | }
toreohm@DESKTOP-06TNPIJ:/mnt/c/Users/INSPIRON_7460/OneDrive/Documentos/platzi/docker/docker$ |
```

Lo que queremos es montar los archivos que queremos usar y no las dependencias, en /usr/src

Entonces para resolver esto queremos que si montamos los archivos al contenedor ignore los módulos (node_modules). Y volvemos a modificar el archivo en la parte del volume.

(Ver imagen abajo)

Notemos que esta vez no se pone el punto (.) al inicio.

```
docker-compose.yml
1  version: "3.8"
2
3  services:
4    app:
5      build: .
6      environment:
7        MONGO_URL: "mongodb://db:27017/test"
8      depends_on:
9        - db
10     ports:
11       - "3000:3000"
12     volumes:
13       - ./usr/src
14       - /usr/src/node_modules
15   db:
16     image: mongo
17
```

- Nuevamente ejecutamos el comando: **sudo docker-compose up -d**
- **docker-compose ps** (Y confirmamos que ya esta funcionando).

```
toreohm@DESKTOP-06TNP1J:/mnt/c/Users/INSPIRON_7460/OneDrive/Documentos/platzi/docker/docker$ docker-compose ps
NAME                IMAGE                COMMAND                  SERVICE    CREATED          STATUS              PORTS
docker-app-1        docker-app           "docker-entrypoint.s..." app        42 seconds ago   Up 41 seconds      0.0.0.0:3000->3000/tcp
docker-db-1         mongo               "docker-entrypoint.s..." db         29 minutes ago   Up 29 minutes      27017/tcp
toreohm@DESKTOP-06TNP1J:/mnt/c/Users/INSPIRON_7460/OneDrive/Documentos/platzi/docker/docker$
```

Finalmente hacemos un ultimo cambio para ver los cambios al código en tiempo real, en el archivo docker-compose.

- **sudo docker-compose up -d**
- Hacemos un cambio en el código y vemos que se refleja en los logs **sudo docker-compose logs -f app**
- Y si refrescamos la pagina localhost:3000 veremos los cambios reflejados.

```
docker-compose.yml
1  version: "3.8"
2
3  services:
4    app:
5      build: .
6      environment:
7        MONGO_URL: "mongodb://db:27017/test"
8      depends_on:
9        - db
10     ports:
11       - "3000:3000"
12     volumes:
13       - ./usr/src
14       - /usr/src/node_modules
15     command: npx nodemon --legacy-watch index.js
16   db:
17     image: mongo
18
```

Compose en equipo: override

- **touch docker-compose.override.yml**
- Modificamos nuestro archivo docker-compose.yml para quitarle las "cosas" que le pusimos.
- En el caso de "environment", docker hace un "merge" entre los dos archivos.
- Y el nuevo archivo (izquierda) quedaría así:

```
🔥 docker-compose.override.yml
1  version: "3.8"
2
3  services:
4    app:
5      build: .
6      environment:
7        MI_VARIABLE: "Hola mundo!"
```

```
🔥 docker-compose.yml
1  version: "3.8"
2
3  services:
4    app:
5      image: platziapp
6      environment:
7        MONGO_URL: "mongodb://db:27017/test"
8      depends_on:
9        - db
10     ports:
11       - "3000-3001:3000"
12
13   db:
14     image: mongo
```

- **sudo docker-compose build**
- **sudo docker-compose up -d**
- **docker-compose exec app bash**
- **env**

```
toreohm@DESKTOP-06TNPIJ:/mnt/c/Users/INSPIRON 7460/OneDrive/Documentos/platzi/docker/docker$ docker-compose exec app bash
root@5325d6c45b95:/usr/src# env
HOSTNAME=5325d6c45b95
MI_VARIABLE=Hola mundo!
YARN_VERSION=1.22.19
PWD=/usr/src
HOME=/root
TERM=xterm
SHLVL=1
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
MONGO_URL=mongodb://db:27017/test
NODE_VERSION=14.21.3
_=/usr/bin/env
root@5325d6c45b95:/usr/src#
```

- **docker-compose ps** (Comprobamos que tu aplicación esté levantada).
- **docker ps**
- **git status** (Y nos asegurarnos de no hacerle commit a los dos archivos)
- Para escalar un servicio teniendo dos instancias de mi app: **sudo docker-compose up -d --scale app=2** (Y esto va a levantar dos contenedores de app)

```
toreohm@DESKTOP-06TNPIJ:/mnt/c/Users/INSPIRON 7460/OneDrive/Documentos/platzi/docker/docker$ sudo docker-compose up -d --scale app=2
[+] Running 3/3
  ✓ Container docker-db-1 Started
  ✓ Container docker-app-2 Started
  ✓ Container docker-app-1 Started
toreohm@DESKTOP-06TNPIJ:/mnt/c/Users/INSPIRON 7460/OneDrive/Documentos/platzi/docker/docker$ docker-compose ps
NAME                IMAGE             COMMAND                  SERVICE    CREATED         STATUS         PORTS
docker-app-1        platziapp         "docker-entrypoint.s..." app        About a minute ago Up About a minute 0.0.0.0:3000->3000/tcp
docker-app-2        platziapp         "docker-entrypoint.s..." app        About a minute ago Up About a minute 0.0.0.0:3001->3000/tcp
docker-db-1         mongo             "docker-entrypoint.s..." db         About a minute ago Up About a minute 27017/tcp
```

El primer contenedor escucha en el puerto 3000 de tu maquina y el segundo en el 3001. Pero ambos están escuchando en el puerto 3000 de cada contenedor.

- Cuando terminamos podemos hacer **docker-compose down**

Administrando tu ambiente de Docker

Como repaso: Podemos eliminar los contenedores que ya no se están utilizando con:

- **docker container prune** (Y así podemos salvar algo de espacio en el disco).
- **docker ps -q** (Nos muestra el ID de los contenedores).
- Para detener y borrar todos los contenedores que estén corriendo podemos usar este comando: **docker rm -f \$(docker ps -aq)**
- **docker network ls** (lista las networks/redes creadas).
- **docker volume ls** (lista todos los volúmenes)
- **docker network prune** (Borra las networks que no estemos usando)
- **docker volume prune** (Borra los volúmenes que no estemos usando)
- **docker system prune** (Borra todo lo que no se esté usando)
- **docker image ls** (lista todas las imágenes)
- **docker image ls -q** (te da el ID de todas las imágenes)

¿Cómo podemos manejar/limitar los recursos a los cuales nuestros contenedores en docker acceden?

- **docker run -d --name app --memory 1g platziapp** (creamos un contenedor con la imagen de platziapp y limitamos su memoria a 1GB)
- **docker run -d --name app --memory 8m platziapp** (creamos un contenedor con la imagen de platziapp y limitamos su memoria a 8MB)
- **docker ps** (cheamos que el contenedor este corriendo).

```
toreohm@DESKTOP-06TNPIJ:/mnt/c/Users/INSPIRON 7460/OneDrive/Documentos/platzi/docker/docker$ docker run -d --name app --memory 1g platziapp
0f3dd4e4916430387d1764aadb51417cd212f06e950903a1a02bbd185a256e48
toreohm@DESKTOP-06TNPIJ:/mnt/c/Users/INSPIRON 7460/OneDrive/Documentos/platzi/docker/docker$ docker ps
CONTAINER ID   IMAGE             COMMAND                  CREATED         STATUS         PORTS                    NAMES
0f3dd4e49164   platziapp         "docker-entrypoint.s..." 4 seconds ago   Up 2 seconds   3000/tcp                 app
5bdafeb1818a   mongo:latest      "docker-entrypoint.s..." About an hour ago Up About an hour 0.0.0.0:27017->27017/tcp  some-mongo
toreohm@DESKTOP-06TNPIJ:/mnt/c/Users/INSPIRON 7460/OneDrive/Documentos/platzi/docker/docker$ docker ps
```

- **docker stats** (Podemos ver cuantos recursos estan consumiendo los contenedores que estan corriendo en nuestro sistema).
- **docker inspect app** (puedo ver si el proceso muere por falta de recursos)

Deteniendo contenedores correctamente: SHELL vs. EXEC

Comandos:

- Estando en la carpeta raíz del proyecto docker de platzi: **cd avanzado/loop**
- **ls** (Y vemos que tenemos dos archivos: Dockerfile loop.sh)

```
$ loop.sh X
avanzado > loop > $ loop.sh
1  #!/usr/bin/env bash
2  trap 'exit 0' SIGTERM
3  while true; do ;; done
```

```
Dockerfile X
avanzado > loop > Dockerfile > ...
1  FROM ubuntu:trusty
2  COPY ["loop.sh", "/"]
3  CMD /loop.sh
```

- **sudo docker build -t loop .** (construyo la imagen la opción -t es "Set the target build stage to build").
- **docker run -d --name loop** (Creamos el contenedor con la imagen)
- **docker ps** (Mostramos los contenedores activos)
- **docker stop loop** (le envía la señal SIGTERM al contenedor para que paré el proceso).
- **docker ps -l** (muestra el ps del último proceso. Siempre que tengamos un código de salida mayor a 128; es el resultado de una salida por una excepción o por una señal no manejada correctamente)

```
toreohm@DESKTOP-06TNPIJ:/mnt/c/Users/INSPIRON 7460/OneDrive/Documentos/platzi/docker/docker/avanzado/loop$ docker stop loop
loop
toreohm@DESKTOP-06TNPIJ:/mnt/c/Users/INSPIRON 7460/OneDrive/Documentos/platzi/docker/docker/avanzado/loop$ docker ps -l
CONTAINER ID   IMAGE     COMMAND                  CREATED        STATUS      PORTS   NAMES
a5c1485c4897   loop     "/bin/sh -c /loop.sh"    4 minutes ago Exited (137) 13 seconds ago   loop
```

- En este caso es 137, 9 unidades mayor a 128. Significa que el comando **docker stop <name-contenedor>** tuvo que hacer al final un **kill -9**. Por lo que docker tuvo que forzar el apagado de este contenedor porque no respondía a la señal de SIGTERM.
- Lo probamos de nuevo: **docker rm loop**
- **docker run -d --name loop**
- Y ahora probamos con **docker kill loop** y vemos que mata el proceso inmediatamente, a diferencia de **docker stop**.
- Lo borramos y creamos nuevamente para intentar otra cosa: **docker rm loop, docker run -d --name loop**
- **docker exec loop ps -ef** (veo los procesos del contenedor, docker exec sirve para Execute a command in a running container).
- Es mejor usar el formato exec sobre el shell.

Formato exec:

```
Dockerfile X loop.sh
avanzado > loop > Dockerfile > ...
You, a few seconds ago | 1 author (You)
1 FROM ubuntu:trusty
2 COPY ["loop.sh", "/"]
3 CMD ["/loop.sh"]
```

Formato shell:

```
Dockerfile X
avanzado > loop > Dockerfile > ...
1 FROM ubuntu:trusty
2 COPY ["loop.sh", "/"]
3 CMD /loop.sh
```

Contenedores ejecutables: ENTRYPOINT vs CMD

- Salimos del directorio loop y entramos a la carpeta ping: **cd ../ping/**
- **ls** (y vemos que tenemos un Dockerfile)

```
Dockerfile X
avanzado > ping > Dockerfile > ...
1 FROM ubuntu:trusty
2 CMD ["/bin/ping", "-c", "3", "localhost"]
3
```

- Construimos la imagen de ping: **sudo docker build -t ping .**
- **docker image ls**

```
toreohm@DESKTOP-06TNPIJ:/mnt/c/Users/INSPIRON_7460/OneDrive/Documentos/platzi/docker/docker/avanzado/ping$ docker image ls
REPOSITORY    TAG       IMAGE ID       CREATED        SIZE
loop          latest   32fbdca32d08   4 hours ago    197MB
mongo         latest   01e8e54fc538   26 hours ago   720MB
platziapp     latest   3c3f806fe2cb   28 hours ago   925MB
docker-app    latest   41b184bfecbb   31 hours ago   925MB
ubuntu        platzi   fd0632c6f71d   2 days ago     77.8MB
ubuntu        22.04    01f29b872827   2 weeks ago    77.8MB
nginx         latest   89da1fb6dcb9   3 weeks ago    187MB
ubuntu        latest   5a81c4b8502e   7 weeks ago    77.8MB
hello-world   latest   9c7a54a9a43c   3 months ago   13.3kB
ping          latest   0bb33882b3e3   2 years ago    197MB
```

- Corremos (creamos) el contenedor con la imagen: **docker run --name pinger ping** (Y al crearlo hace ping de localhost dentro del contenedor y no de nuestra maquina).
- **docker ps -a** (Y vemos la imagen de abajo).


```
toreohm@DESKTOP-06TNPIJ:/mnt/c/Users/INSPIRON 7460/OneDrive/Documentos/platzi/docker/docker/avanzado/ping$ docker run --name pinger ping
PING localhost (127.0.0.1) 56(84) bytes of data.
64 bytes from localhost (127.0.0.1): icmp_seq=1 ttl=64 time=0.076 ms
64 bytes from localhost (127.0.0.1): icmp_seq=2 ttl=64 time=0.024 ms
64 bytes from localhost (127.0.0.1): icmp_seq=3 ttl=64 time=0.023 ms

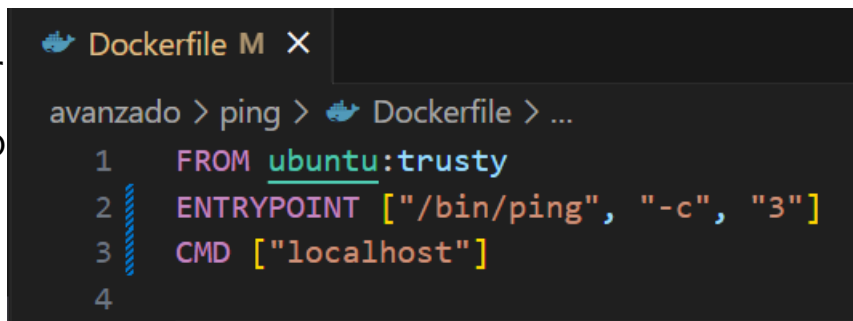
--- localhost ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2105ms
rtt min/avg/max/mdev = 0.023/0.041/0.076/0.024 ms
toreohm@DESKTOP-06TNPIJ:/mnt/c/Users/INSPIRON 7460/OneDrive/Documentos/platzi/docker/docker/avanzado/ping$ docker ps
CONTAINER ID   IMAGE      COMMAND                  CREATED        STATUS        PORTS          NAMES
84b8c16192df   loop      "/bin/sh -c /loop.sh"   3 hours ago   Up 3 hours           looper
5bdafeb1818a   mongo:latest "docker-entrypoint.s..." 6 hours ago   Up 6 hours       0.0.0.0:27017->27017/tcp some-mongo
toreohm@DESKTOP-06TNPIJ:/mnt/c/Users/INSPIRON 7460/OneDrive/Documentos/platzi/docker/docker/avanzado/ping$ docker ps -a
CONTAINER ID   IMAGE      COMMAND                  CREATED        STATUS        PORTS          NAMES
41bbb141d5a    ping      "/bin/ping -c 3 loca..." 45 seconds ago Exited (0) 42 seconds ago
84b8c16192df   loop      "/bin/sh -c /loop.sh"   3 hours ago   Up 3 hours           looper
3c9202ca78c8   platziapp "docker-entrypoint.s..." 5 hours ago   Exited (137) 5 hours ago
5bdafeb1818a   mongo:latest "docker-entrypoint.s..." 6 hours ago   Up 6 hours       0.0.0.0:27017->27017/tcp some-mongo
toreohm@DESKTOP-06TNPIJ:/mnt/c/Users/INSPIRON 7460/OneDrive/Documentos/platzi/docker/docker/avanzado/ping$
```

¿Cómo usar este contenedor como un binario de ping para poder usar en cualquier máquina?

Modificamos el archivo Dockerfile.

ENTRYPOINT de un contenedor es el comando por defecto que se va a correr y va a utilizar la que "diga" CMD como parámetro del ENTRYPOINT. Note que estamos usando el formato **EXEC** en el Dockerfile.

Y en CMD lo único que dejamos es el parámetro del comando ping.



```
Dockerfile M X
avanzado > ping > Dockerfile > ...
1 FROM ubuntu:trusty
2 ENTRYPOINT ["/bin/ping", "-c", "3"]
3 CMD ["localhost"]
4
```

- **docker rm -f pinger**
- **sudo docker build -t ping .** (creamos nuevamente la imagen de ping).
- **docker run --name pinger ping** (Funciona aparentemente igual pero, también puedo hacer que le haga ping a otro target).
- **docker rm -f pinger**
- **docker run --name pinger ping google.com** (Con este comando "pisamos" el parametro de ping que es localhost y lo sustituimos por uno nuevo: google.com)

```
toreohm@DESKTOP-06TNPIJ:/mnt/c/Users/INSPIRON 7460/OneDrive/Documentos/platzi/docker/docker/avanzado/ping$ docker run --name pinger ping google.com
PING google.com (142.251.34.14) 56(84) bytes of data.
64 bytes from 142.251.34.14: icmp_seq=1 ttl=62 time=84.4 ms
64 bytes from 142.251.34.14: icmp_seq=2 ttl=62 time=39.9 ms
64 bytes from 142.251.34.14: icmp_seq=3 ttl=62 time=26.9 ms

--- google.com ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2002ms
rtt min/avg/max/mdev = 26.917/50.428/84.428/24.622 ms
toreohm@DESKTOP-06TNPIJ:/mnt/c/Users/INSPIRON 7460/OneDrive/Documentos/platzi/docker/docker/avanzado/ping$
```

- Y esto se puede confirmar con el comando: **docker ps -l**

```
toreohm@DESKTOP-06TNPIJ:/mnt/c/Users/INSPIRON 7460/OneDrive/Documentos/platzi/docker/docker/avanzado/ping$ docker ps -l
CONTAINER ID   IMAGE      COMMAND                  CREATED        STATUS        PORTS          NAMES
3bf16b92f4de   ping      "/bin/ping -c 3 goog..." 3 minutes ago   Exited (0) 3 minutes ago           pinger
toreohm@DESKTOP-06TNPIJ:/mnt/c/Users/INSPIRON 7460/OneDrive/Documentos/platzi/docker/docker/avanzado/ping$
```


Y el command en la imagen de arriba es la combinación del ENTRYPOINT con el CMD.

Aquí la conclusión es que gracias a la modificación que hicimos en el archivo Dockerfile, podemos usar ping a lo que queramos sin necesidad de estar reconstruyendo (**sudo docker build -t ping .**) la imagen.

El contexto de build

¿Qué es lo que pasa cuando construimos imágenes en docker?

Cuando hacemos el build de una imagen, docker va a montar en un file system temporal todos los archivos disponibles en la ruta que se le pasa como ultimo parámetro a docker build. Muchas veces se le pone punto (.) porque es el directorio en el que te encuentras, ejemplo: (**sudo docker build -t ping .**).

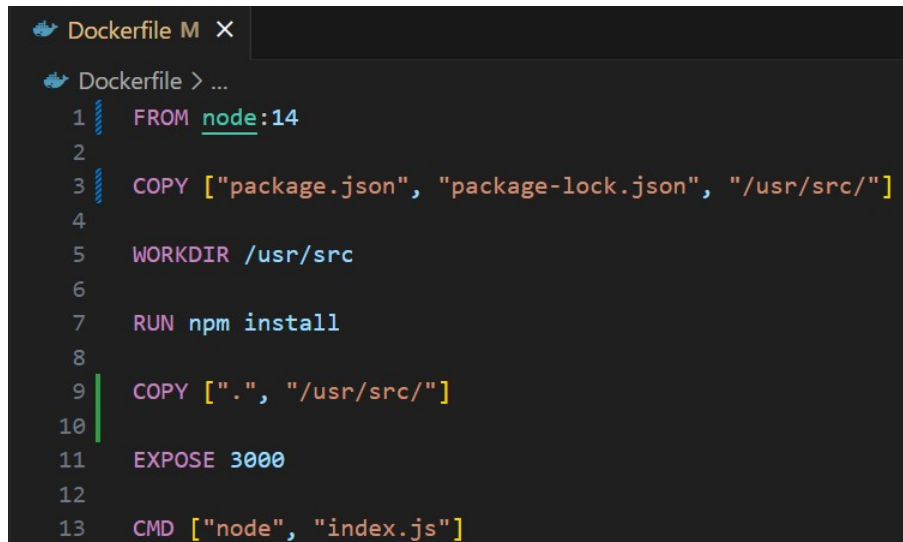
Y de ahí copia todo lo que necesita para el hacer el build.

El problema es que a veces se termina copiando archivos que no se necesitan al momento de hacer el build.

Entonces ¿Cómo podemos optimizar ese contexto de build?

En este momento nuestro Dockerfile es éste (imagen):

Entonces, si al proyecto yo lo hiciera **sudo npm install**, me estaría instalando muchas dependencias reflejadas en el **node_modules** y ocuparía espacio. Y al momento de hacer el build me copiaría todos los archivos, cosa que no queremos.

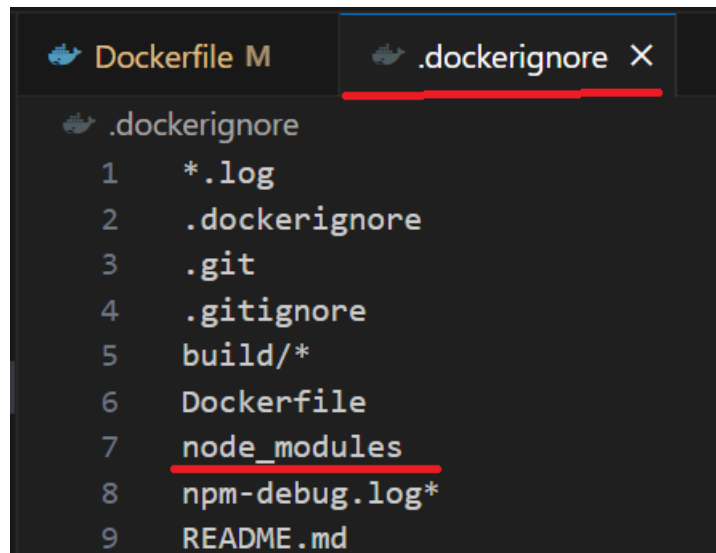


```
Dockerfile M X
Dockerfile > ...
1 FROM node:14
2
3 COPY ["package.json", "package-lock.json", "/usr/src/"]
4
5 WORKDIR /usr/src
6
7 RUN npm install
8
9 COPY [ ".", "/usr/src/" ]
10
11 EXPOSE 3000
12
13 CMD ["node", "index.js"]
```

Entonces la manera de hacer que esto no pase, es la siguiente:

Hay un archivo llamado **.dockerignore** (similar al .gitignore). Y ahí se le puede especificar lo que quieras que se ignore al momento de hacer el build de una imagen, evitando así que el contexto de éste pese más de lo debido.

(Imagen abajo)



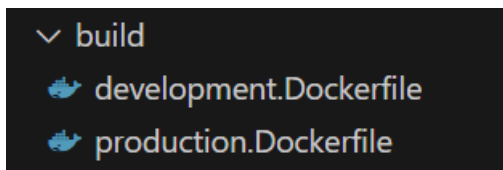
```
Dockerfile M .dockerignore X
.dockerignore
1 *.log
2 .dockerignore
3 .git
4 .gitignore
5 build/*
6 Dockerfile
7 node_modules
8 npm-debug.log*
9 README.md
```

- **docker run -d --rm --name appmx pruebaplatzi** (creamos un contenedor con el nombre de appmx sobre la imagen que creamos (pruebaplatzi) en segundo plano, y con -rm le especificamos que se borre al momento de detenerlo).
- **docker exec -it appmx bash** (Entramos al bash de nuestro contenedor)
- **ls -lah**

Multi-stage build

En ocasiones puede ser que no queramos incluir el código fuente de todo el proyecto sino el solamente el ejecutable. Docker tiene una funcionalidad que nos ayuda a crear nuestras imágenes sin alterar tanto nuestras capas (una capa es cada instrucción en el Dockerfile).

Vemos en nuestro proyecto que hay un directorio que se llama build.



```
build
├── development.Dockerfile
└── production.Dockerfile
```

En la imagen de la página de abajo vemos la estructura del archivo **production.Dockerfile**. Con este archivo lo que nos permite hacer docker en este caso es: que de una fase posterior (la parte de abajo donde dice #productive image), podemos acceder a lo que ocurrió en una fase anterior (la parte de arriba). ¿Cómo? Con el nombre que está en la primera línea: **builder**.

En la parte de arriba del archivo vemos que estamos creando una imagen solo para correr los tests y sólo las dependencias de desarrollo.

```
production.Dockerfile X
build > production.Dockerfile > ...
1 FROM node:12 as builder
2
3 COPY ["package.json", "package-lock.json", "/usr/src/"]
4
5 WORKDIR /usr/src
6
7 RUN npm install --only=production
8
9 COPY [".", "/usr/src/"]
10
11 RUN npm install --only=development
12
13 RUN npm run test
14
15
16 # Productive image
17 FROM node:12
18
19 COPY ["package.json", "package-lock.json", "/usr/src/"]
20
21 WORKDIR /usr/src
22
23 RUN npm install --only=production
24
25 COPY --from=builder ["/usr/src/index.js", "/usr/src/"]
26
27 EXPOSE 3000
28
29 CMD ["node", "index.js"]
```

El resultado de un build multi-imagen es en la imagen final (la ultima fase). Note que en la sección de #productive image (en la ultima fase/stage) copiamos solamente las dependencias de producción y solamente el código que voy a ejecutar, en este caso el archivo index.js. Y de esa manera nos ahorramos el copiar todo lo demás. Y terminamos ejecutando el comando productivo de node (CMD ["node", "index.js"])

- **sudo docker build -t prodapp -f build/production.Dockerfile .**
(Creamos la imagen con el archivo analizado).

- **docker image ls**
- **docker run -d --name prod prodapp**
- **docker ps**
- **docker exec -it prod bash**
- **ls -lah** (Y vemos que esta imagen productiva solo tiene el index.js que es lo que le copiamos y las dependencias productivas. Nótese que no tiene los tests).

```
root@601a34b98bfe:/usr/src# ls -lah
total 124K
drwxr-xr-x  1 root root 4.0K Aug 22 04:12 .
drwxr-xr-x  1 root root 4.0K Apr 18 2022 ..
-rwxrwxrwx  1 root root 579 Aug 17 16:34 index.js
drwxr-xr-x 60 root root 4.0K Aug 22 04:12 node_modules
-rwxrwxrwx  1 root root 97K Aug 22 04:12 package-lock.json
-rwxrwxrwx  1 root root 662 Aug 16 21:46 package.json
root@601a34b98bfe:/usr/src#
```

Nótese también que con este tipo de build si falla el test también el build. Te marca error y no te crea la imagen.

Docker-in-Docker

La posibilidad de utilizar docker desde otros contenedores...

A veces se requieren ejecutar scripts para ejecutar comandos dentro de contenedores, pero los contenedores están aislados y no "saben" que hay un sistema afuera de ellos. Aquí es donde entra el concepto de docker-in-docker. El cliente que tenemos instalados en nuestras maquinas le "habla" a docker por un socket, y esto es un archivo.

Entonces ¿Qué pasa si a un contenedor que tiene el cliente de docker le montamos nuestro docker-socket?

Lo checamos en la terminal (wsl en mi caso).

- **sudo docker run -it --rm -v**
//var/run/docker.sock:/var/run/docker.sock docker:19.03.12
- Y una vez estando dentro del contenedor **docker ps** vemos la imagen desde dentro.

```
/ # docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
6ecc6db2b51d      docker:19.03.12   "docker-entrypoint.s..." About a minute ago  Up About a minute  0.0.0.0:27017->27017/tcp  exciting_dhawan
f0534af624e6      mongo:latest       "docker-entrypoint.s..." 2 hours ago         Up 2 hours          0.0.0.0:27017->27017/tcp  some-mongo
/ #
```

- **docker run -d --name app2 mongo:latest**
- **docker ps**

- Abrimos una nueva terminal y con **docker ps** listamos y vemos los mismos contenedores desde afuera.

Y todo lo que se haga desde el contenedor que creamos, va a impactar a la aplicación de docker (afuera del contenedor). Porque le está "hablando" directamente.

- **`docker run --rm -it -v /var/run/docker.sock:/var/run/docker.sock -v $(which docker):/bin/docker wagoodman/dive:latest mongo:latest`** (Y ahora desde un contenedor ya estamos analizando una imagen con dive sin la necesidad de tener que instalarlo manualmente) (Cabe destacar que esta practica puede ser peligrosa, ya que le das a un contenedor de una imagen que no conoces acceso a tu docker a través del socket. Pudiendo ejecutar así comandos, alterando nuestras imagenes u otras cosas).

Layers				Current Layer Contents		
Cmp	Size	Command	Permission	UID:GID	Size	Filetree
	78 MB	FROM 3460e6984382161	-rwxrwxrwx	0:0	0 B	bin → usr/bin
329 kB	set -eux;	groupadd --gid 999 --system mongod; useradd --	drwxr-xr-x	0:0	0 B	boot
12 MB	set -eux;	apt-get update; apt-get install -y --no-install	drwxr-xr-x	0:0	0 B	dev
3.5 MB	set -ex;	savedAptMark="\$(apt-mark showmanual)"; apt-g	drwxr-xr-x	0:0	111 kB	etc
0 B	mkdir /docker-entrypoint-initdb.d		-rw-r--r--	0:0	0 B	├─ .pwd.lock
1.2 kB	set -ex;	export GNUPGHOME="\$(mktemp -d)"; set -- 'E58302	drwxr-xr-x	0:0	3.0 kB	├─ adduser.conf
116 B	echo "deb [signed-by=/etc/apt/keysring/mongod.gpg] http://\$MON		-rw-r--r--	0:0	100 B	├─ alternatives
627 MB	set -x	&& export DEBIAN_FRONTEND=noninteractive && apt-ge	-rw-r--r--	0:0	100 B	├─ README
14 kB	#(nop) COPY file:8fc8efb4e3db886ece2de8764459b4ab3e639e636ed08b2e		-rwxrwxrwx	0:0	0 B	├─ awk → /usr/bin/mawk
			-rwxrwxrwx	0:0	0 B	├─ awk → /usr/bin/mawk
			-rwxrwxrwx	0:0	0 B	├─ pager → /bin/more
			-rwxrwxrwx	0:0	0 B	├─ rmt → /usr/sbin/rmt-tar
			-rwxrwxrwx	0:0	0 B	├─ which → /usr/bin/which.debianu
			-rwxrwxrwx	0:0	0 B	
			drwxr-xr-x	0:0	8.3 kB	├─ apt
			drwxr-xr-x	0:0	1.4 kB	├─ apt.conf.d
			-rw-r--r--	0:0	92 B	├─ 01-vendor-ubuntu
			-rw-r--r--	0:0	630 B	├─ 01autoremove
			-rw-r--r--	0:0	182 B	├─ 70debconf
			-rw-r--r--	0:0	44 B	├─ docker-autoremove-suggests
			-rw-r--r--	0:0	318 B	├─ docker-clean
			-rw-r--r--	0:0	27 B	├─ docker-disable-periodic-up
			-rw-r--r--	0:0	70 B	├─ docker-gzip-indexes
			-rw-r--r--	0:0	27 B	├─ docker-no-languages
			drwxr-xr-x	0:0	0 B	├─ auth.conf.d
			drwxr-xr-x	0:0	0 B	├─ keyrings
			-rw-r--r--	0:0	0 B	├─ preferences.d
			-rw-r--r--	0:0	2.4 kB	├─ sources.list
			drwxr-xr-x	0:0	0 B	├─ sources.list.d
			drwxr-xr-x	0:0	4.5 kB	├─ trusted.gpg.d
			-rw-r--r--	0:0	2.8 kB	├─ ubuntu-keyring-2012-cdmag
			-rw-r--r--	0:0	1.7 kB	├─ ubuntu-keyring-2018-archiv
			-rw-r--r--	0:0	2.3 kB	├─ bash.bashrc
			-rw-r--r--	0:0	367 B	├─ bindresport.blacklist
			drwxr-xr-x	0:0	17 B	├─ cloud
			-rw-r--r--	0:0	17 B	├─ build.info
			drwxr-xr-x	0:0	201 B	├─ cron.d
			-rw-r--r--	0:0	201 B	├─ e2scrub_all
Layer Details						
Tags: (unavailable)						
Id: 3460e69843821618bf790677784599f0843ff0ec9f253d9ab5c83f1310fca58d						
Digest: sha256:bce45ce613d34bff6a3404a4c2d56a5f72640f804c3d0bd67e2cf0bf97cb950c						
Command:						
#(nop) ADD file:bb1fa1d9d012ae826908afdc8c9fa2feebf221b2ab032e1535259051444						
11a in /						
Image Details						
Image name: mongo:latest						
Total Image size: 720 MB						
Potential wasted space: 5.6 MB						
Image efficiency score: 99 %						
Count	Total Space	Path				
4	2.3 MB	/var/cache/debconf/templates.dat				
2	1.0 MB	/var/cache/debconf/templates.dat-old				
4	766 kB	/var/log/dpkg.log				
4	499 kB	/var/lib/dpkg/status				