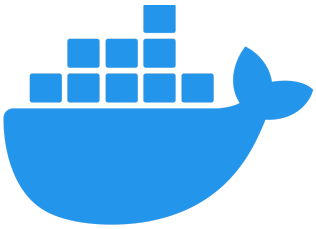
Introducción a docker



https://docs.docker.com/engine/
https://docs.docker.com/get-started/
https://hub.docker.com/_/hello-world
https://stackoverflow.com/questions/16047306/how-is-docker-different-from-a-virtual-machine
https://www.simplilearn.com/tutorials/docker-tutorial/docker-vs-virtual-machine
https://hub.docker.com/_/phpmyadmin
https://hub.docker.com/_/mysql
https://dev.to/azure/desarrollo-de-aplicaciones-node-js-y-express-js-con-docker-4agm
https://www.digitalocean.com/community/tutorials/como-crear-una-aplicacion-node-js-con-docker-es
https://www.digitalocean.com/community/tutorials/how-to-install-and-use-docker-on-ubuntu-20-04
https://www.digitalocean.com/community/tags/docker
https://nodejs.org/en/docs/guides/nodejs-docker-webapp/
https://www.jolugama.com/blog/2018/11/09/app-mean-angular-docker/

https://docs.google.com/presentation/d/16PQt3PjQk2nO8spfjHapSbDHEqhFRRUNzdNOX95acZg/edit#slide=id.g132fffd68c4_0_45
https://docs.google.com/presentation/d/1n21_I4QejKMJcEhDzBKOTevNN8znGKmxE6T07z5Ko08/edit#slide=id.g441308ae56_0_0

https://santex.udemy.com/course/docker-kubernetes-the-practical-guide/
https://santex.udemy.com/course/learn-docker/
https://santex.udemy.com/course/docker-and-kubernetes-the-complete-guide/

Conceptos a aprender

- Virtualización
- ¿Por qué usar Docker?
- ¿Qué son imágenes y contenedores?
- Instalar y configurar en Ubuntu
- Crear un contenedor Hola Mundo
- Crear un contenedor de MySQL y conectarlo al host usando mapeo de puertos
- Persistir un contenedor
- Crear un grupo de contenedores con docker-compose (MySQL, Node, Angular)
- Dockerizar una aplicación simple Hola Mundo

Introducción

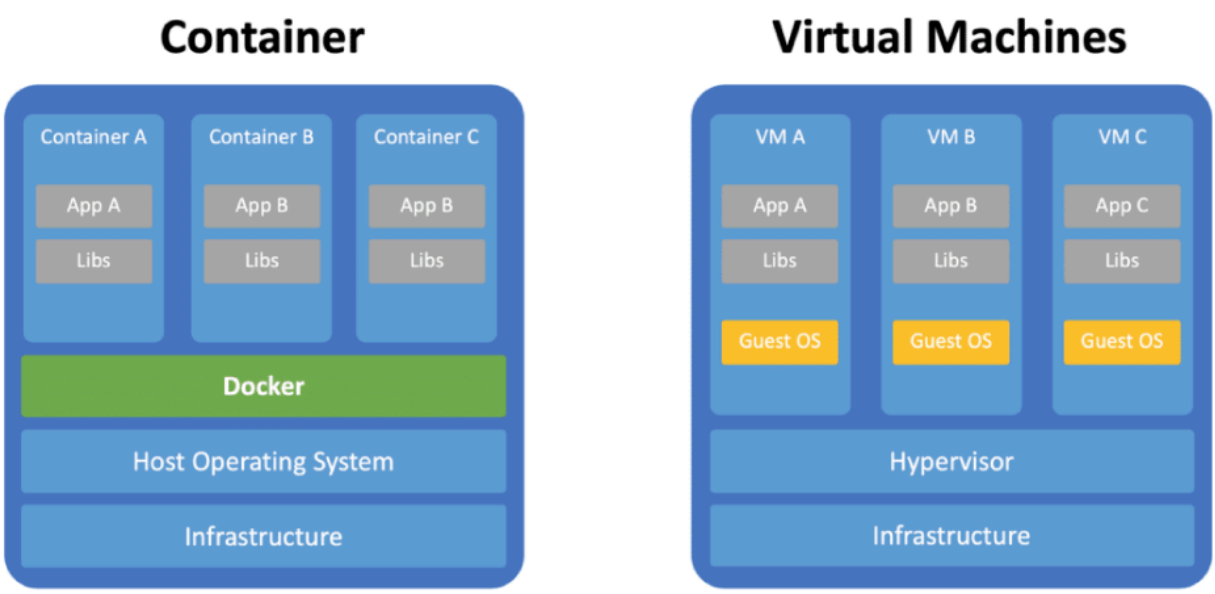
Docker es una herramienta que utiliza características de aislamiento de recursos del Sistema Operativo para administrar contenedores de software, que permiten tener un funcionamiento similar a máquinas virtuales pero sin los costos de inicio y mantenimiento que éstas tienen.

¿Por qué usar Docker?

- **Independencia del Host:** Gracias a Docker nuestra aplicación funcionará siempre igual la cantidad de instancias que queramos, independientemente del Sistema Operativo del host en el cual corramos dicha aplicación, por ejemplo, una aplicación BackEnd en NodeJS correrá igual en nuestra máquina local, en una instancia de AWS o en una Virtual Machine en algún servidor.
- **Isolation / Aislamiento**: Nos permite tener una imagen distinta de la solución para cada versión con sus librerías y dependencias. Y ésta a su vez se mantiene aislada del sistema operativo del host (nuestra máquina o el servidor)
- **Seguridad:** Docker está construido para que sea seguro, ningún container tiene acceso a otro container salvo que se configure de tal forma, incluso se pueden agregar capas de seguridad en el proceso de construcción de cada imagen.
- **Control de ambientes**: Se pueden crear diferentes versiones para distintos ambientes con sus respectivas configuraciones, incluso tener bases de datos diferentes para cada ambiente.
- **Integración continua y Desarrollo rápido**: Docker está preparado para ser integrado con la mayoría de las herramientas de CI/CD lo cual hace que el delivery de nuevas versiones sea muy rápido. Todas las tareas tediosas de despliegue y entrega quedan automatizadas.
- **Escalabilidad y Flexibilidad:** Al estar construido sobre la tecnología de Containers, podemos correr múltiples instancias de la misma imagen de forma paralela (escalabilidad horizontal), esto nos permitirá tener una mejor performance en nuestra solución.

Arquitectura de Docker

Para diferenciar y representar las diferencias entre la arquitectura de Containers vs Máquinas Virtuales, aquí tenemos un gráfico:



Source: https://miro.medium.com/max/1024/0*zBEamnh9NKe7Rmdk.png

Como se puede ver Docker funciona un nivel más arriba que una máquina virtual, eso nos permitirá (ya que Docker se encarga de orquestar los containers), tener mayor libertad para crear nuevos containers sea de forma deliberada o porque alguno falla y se vuelve a recrear, también podemos apreciar que al funcionar un nivel más arriba, al momento de crear una nueva instancia el costo (en tiempo), es mucho menor ya que el sistema operativo ya se encuentra funcionando.

¿Qué son imágenes y contenedores?

Imágenes

Son archivos usados para ejecutar un contenedor. Contienen instrucciones precisas para permitir la ejecución completa de una aplicación y dependen del kernel del sistema operativo anfitrión.

- Archivos compuestos por múltiples capas
- Usados para instanciar contenedores Docker
- Cada capa representa instrucciones precisas en el Dockerfile de la imagen
- Todas las capas, excepto la última, son de solo lectura.
- Cada capa es una colección de cambios con respecto a la capa que le precede.

Contenedores

Son la unidad de software estándar para empaquetar el código de una aplicación junto a todas sus dependencias para correrlo de manera rápida y confiable en cualquier ambiente. En otras palabras los contenedores son las INSTANCIAS de las imágenes, podemos crear cuantos contenedores queramos de una misma imágen.

- Todos los cambios en el contenedor están en la capa superior de escritura
- La capa de escritura es eliminada si el contenedor es eliminado
- Los cambios en el contenedor no afectan a la imagen de la que se basa.
- Cada contenedor funciona como una “mini” máquina virtual.

Instalar y configurar en Docker

Desinstalar versiones antiguas

Antes que nada, debemos asegurarnos de que no hay versiones antiguas de docker instaladas, para quitar todas, hacemos uso de apt remove:

```
sudo apt remove docker docker-engine docker.io containerd runc
```

Luego de correr este comando, podemos estar seguros de que tenemos todo lo necesario para hacer una instalación ‘limpia’. Si la consola nos reporta que ninguno de los paquetes estaba instalado, no hay problemas.

Configurar repositorio

Primero debemos actualizar el índice de paquetes e instalar algunos paquetes que nos permitirán usar un repositorio por HTTPS:

```
sudo apt update
sudo apt install -y ca-certificate curl gpupg lsb-release
```

Una vez actualizado el índice y los paquetes necesarios instalados, pasamos a agregar la llave GPG de Docker:

```
sudo mkdir -p /etc/apt/keyrings
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg
--dearmor -o /etc/apt/keyrings/docker.gpg
```

Una vez descargada la llave, nos aseguramos de que tenga los permisos necesarios para que pueda ser leída y permita actualizar el índice de paquetes:

```
sudo chmod a+r /etc/apt/keyrings/docker.gpg
```

Ya preparada la llave, pasamos a usarla para configurar el repositorio:

```
echo \
  "deb [arch=$(dpkg --print-architecture)
signed-by=/etc/apt/keyrings/docker.gpg]
https://download.docker.com/linux/ubuntu \
  $(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list
> /dev/null
```

Instalar Docker

Actualizamos el índice de paquetes e instalamos todos los paquetes necesarios

```
sudo apt update
sudo apt install -y docker-ce docker-ce-cli containerd.io
docker-compose-plugin
```

Una vez que todos los paquetes estén instalados, validamos la instalación corriendo la imagen hello-world

```
sudo docker run hello-world
```

Este comando descarga una imagen de prueba y la corre en un contenedor. Si todo sale bien, nos muestra un mensaje de éxito y se cierra.

Configurar Docker para que no necesite sudo

Como el demonio de Docker corre bajo el usuario root, es necesario usar sudo para poder ejecutar los comandos de docker. Para evitar que esto sea necesario, necesitamos crear un grupo docker y agregar nuestro usuario personal a ese grupo.

Primero creamos el grupo “docker”, nótese que es necesario usar “sudo”:

```
sudo groupadd docker
```

Lo siguiente es agregar nuestro usuario al grupo “docker”:

```
sudo usermod -aG docker $USER
```

Para aplicar los cambios podemos cerrar la consola y abrir una nuevo, o ejecutar el siguiente comando para activar el cambio al grupo docker:

```
newgrp docker
```

Finalmente, podemos verificar que podemos usar el comando docker sin sudo, volviendo a ejecutar la imagen hello-world:

```
docker run hello-world
```

Configurar Docker para que se inicie al bootear Ubuntu

Docker tiene un par de servicios que podemos activar para que se ejecuten automáticamente al iniciar la computadora. Para esto, hacemos uso del comando systemctl:

```
sudo systemctl enable docker
sudo systemctl enable containerd
```

Crear un contenedor Hola Mundo

En este apartado crearemos una imagen de docker a partir de un proyecto ReactJS.

1. Debemos crear una aplicación ReactJS básica, utilizamos el siguiente comando:

```
npx create-react-app web-app-with-docker
```

2. Una vez terminada la creación, ingresamos al nuevo directorio y ejecutamos la aplicación para corroborar que funciona como es esperado:

```
cd .\web-app-with-docker\
npm start
```

3. Si navegamos en un browser a la siguiente ruta: http://localhost:3000/ debemos ver algo como esto:

4. Si todo es correcto, ya estamos listos para crear nuestra imagen de docker. Para ello abrimos cualquier editor de texto en la carpeta creada. Creamos un archivo “Dockerfile” (no tiene extensión, solo se llama Dockerfile, esto es por convención para más info https://docs.docker.com/engine/reference/builder/).
5. En ese archivo copiamos el siguiente código:

```
FROM node:12-alpine3.12 AS build
WORKDIR /app

COPY package.json ./
```

```
RUN npm install
COPY . .

RUN npm run build

FROM nginx:1.19.0-alpine AS prod-stage
COPY --from=build /app/build /usr/share/nginx/html
EXPOSE 80

CMD ["nginx", "-g", "daemon off;"]
```

En este archivo tenemos:

- Tomamos la primer capa, la imagen “node:12-alpine3.12” en esta imagen basaremos nuestra nueva imagen, además le colocamos un alias “AS build”
- Nos posicionamos en la carpeta “app”
- Copiamos el package.json del proyecto al destino “./”
- Corremos “npm install” para que se creen todas las dependencias.
- Copiamos todo en el destino
- Ahorra corremos “npm run build” para que se “compile” el proyecto en modo build
- Creamos una nueva capa, tomamos la imagen “nginx:1.19.0-alpine” que es un servidor web donde alojaremos nuestra solución final.
- Copiamos desde la imagen / capa que denominamos build la carpeta “/app/build” en el destino “/usr/share/nginx/html”
- Exponemos el puerto 80
- Ejecutamos el comando “nginx -g daemon off;” en la capa final.

6. Ahora estamos en condiciones de construir nuestra imagen, para ello utilizamos el comando:

```
docker build -t web-app .
```

- El flag “-t” indica que le vamos a asignar el nombre o tag “web-app” y el “.’” significa que deberá encontrar el archivo Dockerfile en el directorio actual.

El resultado debería ser:

```
Mauricio@HPOmen D:\..\..\web-app-with-docker > docker build -t web-app .
[+] Building 185.8s (14/14) FINISHED
 => [internal] load build definition from Dockerfile
 => => transferring dockerfile: 304B
 => [internal] load .dockerignore
 => => transferring context: 52B
 => [internal] load metadata for docker.io/library/nginx:1.19.0-alpine
 => [internal] load metadata for docker.io/library/node:12-alpine3.12
 => CACHED [build 1/6] FROM docker.io/library/node:12-alpine3.12@sha256:8883e589c6a5da49a6fea80afec58dec2213eb2aa
 => [internal] load build context
 => => transferring context: 1.03kB
 => CACHED [prod-stage 1/2] FROM docker.io/library/nginx:1.19.0-alpine@sha256:17ba9c1ca3dbea0b44f7c890862161c4976
 => [build 2/6] WORKDIR /app
 => [build 3/6] COPY package.json ./
 => [build 4/6] RUN npm install
 => [build 5/6] COPY . .
 => [build 6/6] RUN npm run build
 => [prod-stage 2/2] COPY --from=build /app/build /usr/share/nginx/html
 => exporting to image
 => => exporting layers
 => => writing image sha256:9ac1d53fbbb4d48d865a8a83a6bae67a47790250b9a23c491df35166c402b032
 => => naming to docker.io/library/web-app

Use 'docker scan' to run Snyk tests against images to find vulnerabilities and learn how to fix them
Mauricio@HPOmen D:\..\..\web-app-with-docker >
```

7. Si ejecutamos el siguiente comando:

```
docker images
```

Veremos que la imagen que creamos existe y se puede utilizar.

```
Mauricio@HPOmen D:\..\..\web-app-with-docker > docker images
REPOSITORY                                    TAG       IMAGE ID       CREATED          SIZE
web-app                                       latest    9ac1d53fbbb4   36 minutes ago   21.8MB
postgres                                      latest    68f5d950dcd3   5 weeks ago      379MB
form3-interview-test_accounthelper            latest    262e42194056   6 months ago     982MB
```

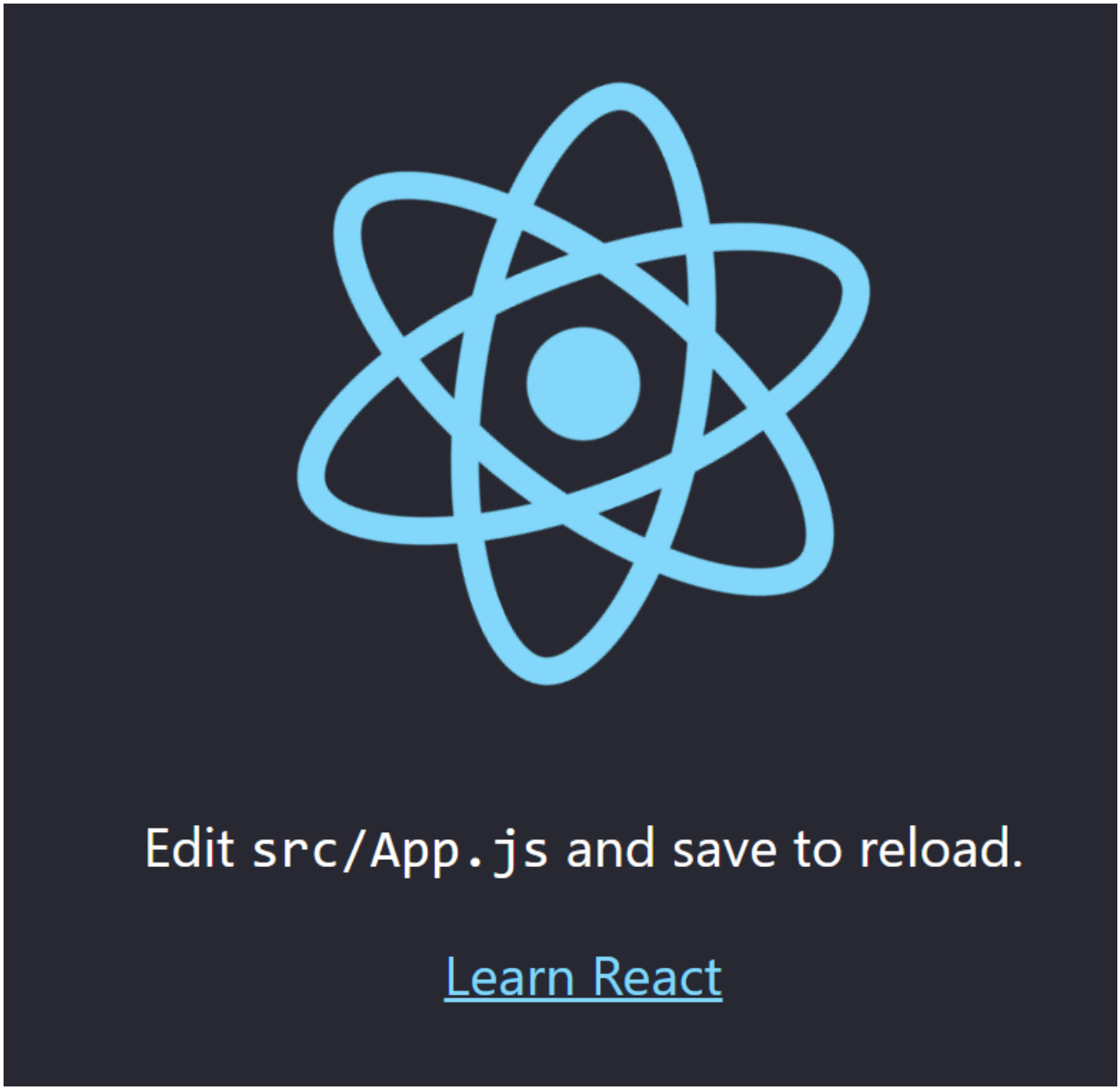
8. Ahora ya podemos crear un nuevo container de nuestra imagen utilizando el siguiente comando:

```
docker run -it -p 5000:80 –name webappdockerized web-app
```

Hemos utilizado los siguientes parámetros:

- “-it”: para que corra en consola y podamos ir viendo los logs.
- “-p”: para poder mapear el puerto 5000 al 80 expuesto internamente.
- “-name”: un nombre simple para después poder identificarlo.

9. Si navegamos a “http://localhost:5000/” deberíamos ver lo siguiente:



Esto quiere decir que hemos creado exitosamente nuestra imagen y luego un container de dicha imagen.

Mapeo de puertos

Ya se ha explicado que Docker por definición es seguro, por ende debemos abrir una “puerta” para acceder al contenido de nuestros contenedores. Esas “puertas” son los puertos. Debemos configurar un puerto en el Host y mapear (igualarlo) a uno del container.

Esto quiere decir que debemos indicar que, por ejemplo, una aplicación web que por defecto expone su contenido en el puerto 80 deberá ser expuesta en el Host en el puerto 90, 80, 8080, 9898 (el que decidamos y esté libre), y de esta forma podremos navegar el contenido de la aplicación web en el browser utilizando la siguiente url: *http://<<ip-del-host>>:90* (si el seleccionado fue puerto 90, si el host es nuestra máquina local podemos usar “localhost”).

Para hacer dicho mapeo debemos hacerlo por línea de comandos cuando inicializamos el container, por ejemplo:

Debemos conocer que puertos expone nuestra imagen utilizando la opción “inspect”:

```
docker image inspect postgres
```

El resultado será un JSON con la siguiente forma:


```
[
    {
        "Id":
"sha256:68f5d950dcd3ecf7da0ad0aec5b2b531cf8f4141e56afa7e3c3c68c8308e68be",
        "RepoTags": [
            "postgres:latest"
        ],
        "RepoDigests": [

"postgres@sha256:766e8867182b474f02e48c7b1a556d12ddfa246138ddc748d70c891bf2
873d82"
        ],
        "Parent": "",
        "Comment": "",
        "Created": "2022-11-15T06:38:54.351326401Z",
        "Container":
"47b0bc59cd285c32a718ab63a3f69e59360118b0920e423d30a1fcbaf0f1f7ef",
        "ContainerConfig": {
            "Hostname": "47b0bc59cd28",
            "Domainname": "",
            "User": "",
            "AttachStdin": false,
            "AttachStdout": false,
            "AttachStderr": false,
            "ExposedPorts": {
                "5432/tcp": {}
            },
            …
```

```
        }
    }
]
```

La propiedad “ExposedPorts” nos indica que en esta imagen podemos utilizar el puerto 5432 para acceder a esta base de datos PostgreSQL.

Ahora cuando decidimos iniciar un container con dicha imagen lo hacemos utilizando el siguiente comando:

```
docker run -p 5432:5432 -d postgres
```

De esta forma quedará mapeado el puerto 5432 (-p <puerto-host>:<puerto-container>) de nuestro Host al puerto 5432 del container.

Persistencia de datos

Por definición, no podemos persistir datos en los containers, no al menos si esperamos que luego de ser reiniciados o regenerados se mantengan.

Por eso Docker ofrece algunos métodos de persistencia, entre ellos “Bind Mounts”, “VOLUMES”

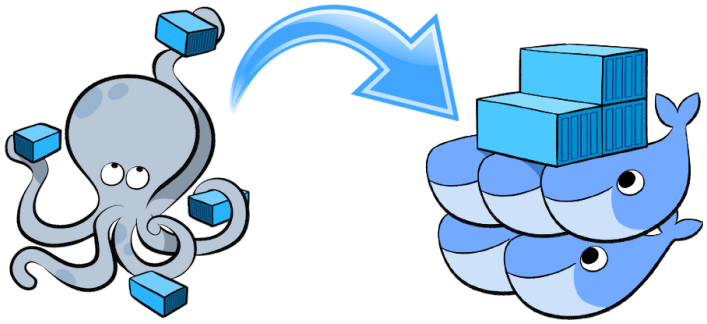
- Bind Mounts: es un método utilizado por Docker para compartir datos entre contenedores y la máquina Host, esto se logra montando archivos o directorios de la máquina host en el contenedor Docker. Tiene la particularidad de no requerir la existencia de dicho archivo o directorio ya que si no existe se puede crear a demanda.
- Volumes: tiene la particularidad de ser gestionada 100% por Docker mediante comandos CLI, esto permite que dicha herramienta sea agnóstica del host. Es también compartida entre containers y se puede almacenar en la nube, o host remoto.

Docker Compose

Es una herramienta que permite construir soluciones con más de un container (por ejemplo, un frontend en ReactJS, con un backend en NodeJS y una base de datos PostgreSQL).
Docker Compose se instala en conjunto con Docker Desktop, en Linux podemos instalarlo como un plugin (https://docs.docker.com/compose/install/).

Una solución en Docker Compose se construye en un archivo YAML en el cual se definen las imágenes con sus versiones, los puertos expuestos, los VOLUMES compartidos así como los

recursos de RED (por ejemplo la aplicación FrontEnd puede ver al BackEnd, pero no a la Base de Datos).



Docker Compose Ejemplo

```yaml
version: '2'
services:
  postgres:
    image: postgres
    ports:
      - '5432:5432'
    environment:
      - POSTGRES_DB=postgres
      - POSTGRES_USER=postgres
      - POSTGRES_PASSWORD=postgres
```

En este archivo YAML definimos:

- Servicios: es decir los componentes que va a tener nuestra solución (“postgres” es el nombre, podría bien llamarse “baseDeDatos”, “dataBase”)
- Image: la imagen que utilizaremos para dicho servicio con su o sus puertos.
- Environment: un listado de variables de entorno que podríamos querer pasarle a nuestro servicio, en este caso es información del usuario root de la base de datos.

Para levantar o ejecutar este docker compose, debemos ejecutar el siguiente comando en nuestra consola (parados en el directorio donde se encuentra dicho archivo YAML):

```
Docker compose up
```

El resultado: