Introducción a Docker

Creación de app fullstack de ejemplo

¿Qué es docker?

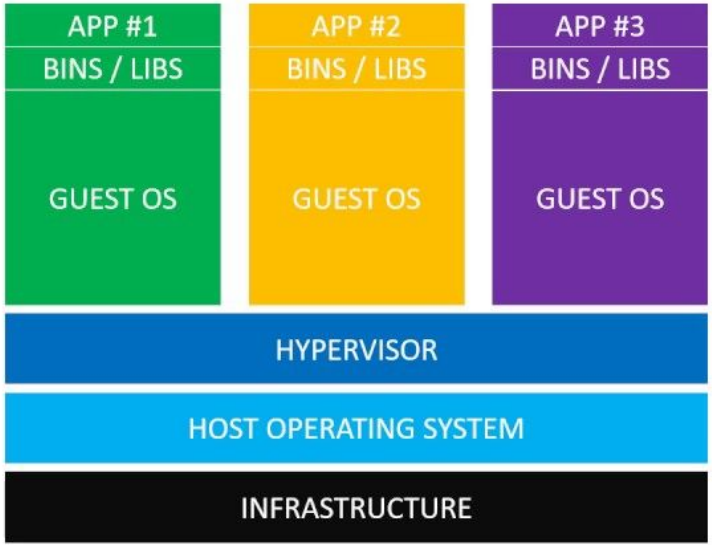
Docker es un proyecto de código abierto que automatiza el despliegue de aplicaciones dentro de contenedores de software, proporcionando una capa adicional de abstracción y automatización de virtualización de aplicaciones en múltiples sistemas operativos.

¿Qué es docker?

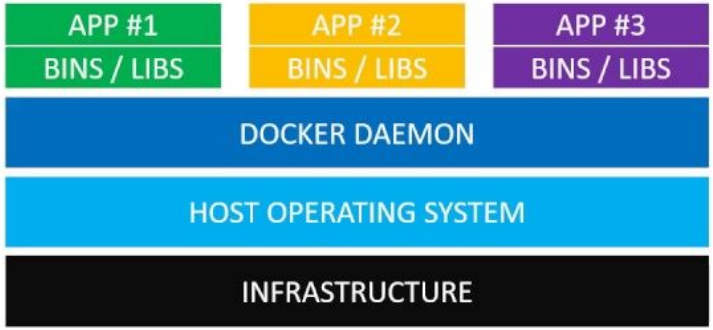
Docker es un “emulador” de entornos aislados para poder ejecutar programas sin que afecte a mi sistema operativo (SO) y pudiendose llevar, crecer y replicar en otros SS.OO. o entornos.

Parecido a VirtualBox o VMWare, pero mucho más ligero y a nivel de sistema operativo. Básicamente no vas a tener más de un sistema operativo completo corriendo en tu máquina.

¿Qué es docker?



Virtual Machines



Docker Containers

¿Qué es docker?

Docker consta de imágenes y contenedores.

Una **imagen** es una parte mínima y suficiente del sistema operativo para ejecutar programas.

Un **contenedor** es un entorno aislado con la copia de una imagen la cual se puede configurar.

Instalación de Docker en ubuntu

1. Instalamos docker y docker-compose:

```
#   sudo snap install docker docker-compose
```

2. Añadimos al usuario de ubuntu al grupo docker

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

3. En https://hub.docker.com/ podemos buscar imagenes de docker ya preparadas

4. Podemos comprobar qué imágenes tenemos con:

```
#   docker images
```

Comandos de Docker: Básicos

Listado de imágenes de docker instaladas:

```
# docker images
```

Listado de contenedores de docker creados:

```
# docker container ls
```
o su alias
```
# docker ps
```

Para bajar una de imágen de docker del docker hub:

```
# docker pull IMAGEN
```

Comandos básicos de Docker

```
#   docker build -t NOMBRE-CONTENEDOR .
#   docker run IMAGEN
#   docker inspect CONTENEDOR
#   docker logs CONTENEDOR
#   docker exec -it CONTENEDOR /bin/bash
#   docker start / stop / restart CONTENEDOR
#   docker rm CONTENEDOR
#   docker rmi IMAGEN
```

Pasos para crear y lanzar un contenedor Docker

1. Crear el directorio de trabajo
2. Entrar en el directorio de trabajo
3. Crear el fichero Dockerfile
4. Crear el contenedor: docker build
5. Lanzar el contenedor creado: docker run
6. Comprobar que está lanzado: docker ps
7. Comprobar logs del contenedor: docker logs

Ejemplo: Creando un contenedor nginx

Levantando el contenedor nginx básico:

```
#    mkdir docker_example
#    cd docker_example
#    docker pull nginx
#    docker images
#    docker ps
#    docker run -d --name "web" -p 80:80 nginx;
#    docker ps
#    docker logs [CONTENEDOR-ID]
#    docker inspect [CONTENEDOR-ID]
```

Montando una carpeta compartida entre un directorio local y un directorio dentro del contenedor Docker

```
#    mkdir www
#    cd www
#    echo “<h1>MI CONTENEDOR NGINX CON DOCKER</h1>” > index.html
#    cd ..
#    docker rm [CONTENEDOR-ID]
#    docker ps
#    docker run -d --name "web" -p 80:80 -v $(pwd)/www:/usr/share/nginx/html nginx;
#    docker ps
#    docker ps -a
#    docker rm [CONTENEDOR-ID]
#    docker ps -a
```

Análisis del los parametros del comando “docker run”

```
#    docker run -d --name "web" -p 80:80 -v $(pwd)/www:/usr/share/nginx/html nginx
```

COMANDO: **run** Sirve para lanzar un contenedor de la imagen nginx, llamado "web" PARAMETROS:

-d o **--detach**, ejecuta el contenedor en segundo plano (background).
--name para darle un nombre a nuestro contenedor y sea mas sencillo referenciarlo e identificarlo.
-v le indicamos que monte un volumen que enlaza desde la ruta local *$(pwd)/www* a la ruta dentro del contenedor */usr/share/nginx/html* nginx que es la que utiliza este programa para servir

Analisis del los parametros del comando “docker run”

-p conectamos el puerto 80 de nuestra máquina con el puerto 80 del contenedor, que como tiene el servidor nginx corriendo en dicho puerto, estaremos dando acceso al puerto del contenedor desde nuestro puerto, siempre y cuando este esté libre, si no deberemos ponerlo en un puerto libre de nuestro equipo.

Podemos comprobarlo:

- Comprobamos http://localhost en un navegador
- Modificamos index.html
- Comprobamos http://localhost en un navegador que los cambios se ven

EJERCICIOS

1 . Clonar un componente y hacer que se muestre la demo en un contenedor de nginx

2. Arrancar tres contenedores de nginx en los puertos 80, 8080 y 8081

Haciendo más cosas con Docker

¿ Y si necesito crear un servidor express en node para exponer un api ?

Pues lo ideal es separar el servidor node-express del servidor de mongodb.

De esta manera si necesito escalar o cambiar uno de los dos, el otro no tiene porqué verse afectado. Seguiremos los siguientes pasos:

1. Vamos a crear un contenedor con node, express y mongodb.
2. Vamos a crear un contenedor con un servidor de mongodb
3. Vamos a conectar los dos contenedores

Servidor con node, express y mongodb

```
# mkdir api (dentro de docker_example)
# cd api
# npm init
# npm install --save express mongodb
```

Servidor con node, express y mongodb

Vamos a crear un fichero index.js con un servidor express mínimo:

```
const express = require('express');
const app = express();
const PORT = 3000;
app.get('/', function(req, res) {
    res.json({"hello": "express with mongo"});
});
app.listen(PORT, function(){
    console.log('Your node js server is running on PORT:',PORT);
});
```

Ejecutamos:

```
# node index.js
```

Probamos http://localhost:3000

Comandos de Dockerfile

FROM nos permite especificar desde qué imagen base de Docker Hub (https://hub.docker.com/) queremos construir.

RUN nos permite ejecutar un comando.

WORKDIR establece un directorio como el directorio de trabajo para las instrucciones COPY, RUN y CMD.

COPY y ADD permite copiar archivos o un directorio completo desde una fuente fuera del contenedor a un destino dentro del contenedor.

EXPOSE expone el puerto en el que el contenedor escuchará.

CMD establece el comando predeterminado para ejecutar nuestro contenedor.

Nuestro Dockerfile para el api node-express

Creamos el fichero *Dockerfile* con la configuración de nuestra imagen:

```
FROM node:8
WORKDIR /usr/src/app
COPY package*.json ./
RUN npm install
COPY . .
EXPOSE 8080
CMD [ "node", "index.js" ]
```

Fichero .dockerignore

Creamos el fichero .*dockerignore* para evitar copiar a la imagen los ficheros que no queramos copiar en ella.

```
node_modules
*.log
docker-compose.yml
Dockerfile
data
```

Construyendo y corriendo el contenedor node

Construimos la imagen según le indica Dockerfile:

```
# docker build -t manufosela/api .
```

```
# docker images
```

Lanzamos el contenedor

```
# docker run -p 3000:3000 -d manufosela/api
```

```
# docker ps
```

Pregunta

¿Si cambio algo en el index.js se refleja en http://localhost:3000?

¿Por qué?

No se refleja puesto que en el Dockerfile hemos indicado con COPY . . que copiemos toda la carpeta, a excepción de lo indicado en el fichero .dockerignore, en la carpeta de trabajo de la imagen. Una vez copiado ya no es posible alterar su contenido. Esto es lo que querremos hacer cuando subamos a producción nuestra aplicación, pero no cuando estamos desarrollando.

Solución

Modificamos index.js.

Tendremos que borrar el contenedor y volver a construirlo y volver a ejecutar el contenedor para que el servidor node-express lea los cambios.

```
# docker rm [CONTENEDOR_ID]
```

```
# docker build -t manufosela/api .
```

```
# docker run -p 3000:3000 -d manufosela/api
```

Ahora sí se ven reflejados los cambios en http://localhost:3000

Docker de mongoDB

Vamos a utilizar la imagen ya hecha de mongodb de docker hub:

```
#    docker pull mongo
#    docker images
#    docker run -it -d mongo
#    docker ps
```

Con esto tenemos corriendo un servidor mongodb que expone el puerto 27017, que es el puerto por defecto de mongo, de manera que tenemos una base de datos mongo corriendo en nuestro equipo, pero sin haber tenido que instalar ni configurar nada y sin que haya "ensuciado" nuestro sistema operativo instalando dependencias o librerías.

Probando el contenedor de mongoDB

Para poder probarlo, primero debemos averiguar la IP del servidor mongodb, para ello ejecutamos:

```
# docker ps
```

Nos fijamos, al final, en el nombre aleatorio que le ha dado al contenedor de mongo que serán dos nombres separados por un guión bajo. Usamos dicho nombre para ejecutar lo siguiente:

```
# docker run -it --link=[NOMBRE_CONT]:mongo mongo /bin/bash
```

Con esto entramos en la consola del contenedor que está corriendo el servidor de mongo. Ejecutamos:

```
# env
```

Probando el contenedor de mongoDB

Que nos muestra todas las variables de entorno. Nos fijamos en la línea que nos muestra MONGO_PORT_27017_TCP_ADDR para obtener la ip del contenedor:

```
[...]
HOME=/root
SHLVL=1
MONGO_PORT_27017_TCP_ADDR=172.17.0.3
MONGO_ENV_JSYAML_VERSION=3.10.0
MONGO_MAJOR=4.0
[...]
```

Cerramos la consola con exit

Probando el contenedor de mongoDB

El comando para ejecutar mongo es mongo para lo que deberemos tener instalado el cliente de mongodb en nuestro equipo llamado mongodb-clients. Para saber si lo tenemos instalado podemos ejecutar:

```
# sudo dpkg --get-selections | grep mongo
```

Si no lo está lo podemos instalar con:

```
# sudo apt install mongodb-clients
```

Probando el contenedor de mongoDB

Si lo ejecutamos simplemente mongo y no tenemos un servidor mongodb local corriendo, obtendremos este error:

couldn't connect to server 127.0.0.1:27017, connection attempt failed: SocketException: Error connecting to 127.0.0.1:27017 :: caused by :: Connection refused

Para poder conectar con el servidor mongodb de nuestro contenedor deberemos indicar la ip y el puerto al que conectarnos:

```
# mongo [IP]:27017
```

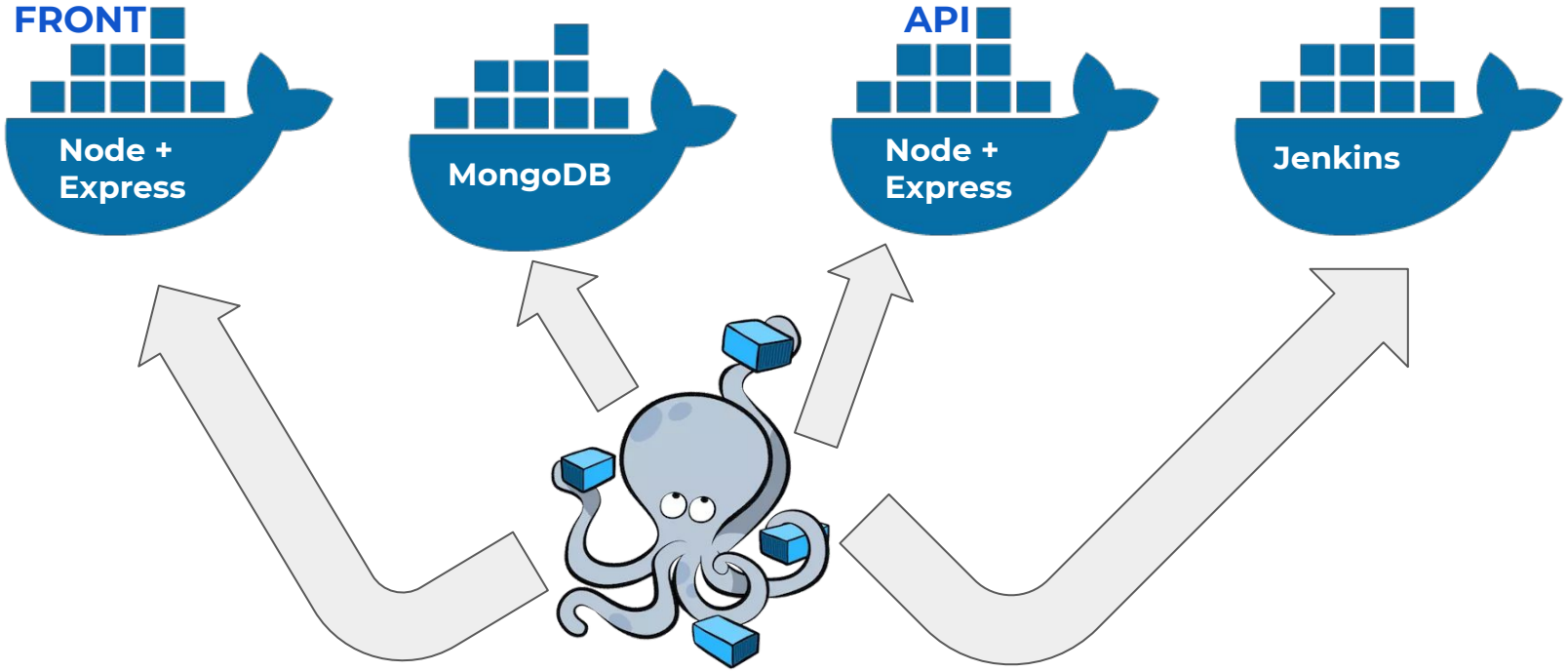
Probamos mongodb

Una vez conectados con la línea de comandos de mongo que se caracteriza por tener el prompt **>** Para interactuar con la base de datos podemos ejecutar algunos comandos de mongo:

```
#    show dbs
#    use local
#    show collections
#    db.startup_log.find({})
#    db.startup_log.find({}).pretty()
#    use mydbdb.mycollection.insert({elemento:"uno"})
#    show collections
#    db.mycollection.find()
```

CTRL+C para salir de la consola

Docker y Docker-compose para ayudarnos



Conectar dockers con Docker-compose

Para conectar el contenedor de node, express, mongodb con el contenedor de MongoDB vamos a valernos de docker-compose

Docker-compose es un orquestador de contenedores para que se relacionen entre ellos.

Se configura mediante un archivo .yml llamado *docker-compose.yml*

En dicho fichero se indica qué contenedores se enlazan con quien, de manera que de una sola llamada podemos arrancar y relacionar varios contenedores.

Creamos el fichero docker-compose.yml

```
version: "2"
services:
  app:
    container_name: app
    restart: always
    build: .
    ports:
      - "3000:3000"
    links:
      - mongo
  mongo:
    container_name: mongo
    image: mongo
    volumes:
      - ./data:/data/db
    ports:
      - "27017:27017"
```

Paramos todos los contenedores de mongo y api

Antes de continuar y para poder utilizar docker-compose debemos parar los contenedores que hemos lanzado manualmente. Para ello ejecutamos:

```
docker ps
```

Y anotamos los CONTAINER_ID del contenedor de mongo y node-express, el de nginx podemos dejarlo corriendo. Una vez tengamos los dos ids ejecutamos:

```
docker stop [CONTAINER_ID_NODE] [CONTAINER_ID_MONGO]
```

Con esto los paramos, pero los contenedores siguen lanzados, por lo que ahora deberemos borrarlos:

```
docker rm [CONTAINER_ID_NODE] [CONTAINER_ID_MONGO]
```

Probando docker-compose

Ejecutamos:

```
#    docker-compose build
#    docker-compose up -d
#    docker-compose ps
#    docker ps
```

Podemos probar que tenemos servidor de node y de mongo corriendo, entrando en http://localhost:3000 y ejecutando desde la linea de comandos **mongo**, esta vez sin IP ni puerto, ya que docker-compose conecta automáticamente localhost con la IP del contenedor mongo.

Probando docker-compose

Con docker-compose podemos parar todos los contenedores de una vez.

```
#    docker-compose down
```

Y podemos comprqalos con:

```
#    docker-compose ps
#    docker ps
```

Viendo que no hay ningún contenedor corriendo.

Usando mongo en nuestro API

Modificamos el fichero index.js añadiendo lo que viene en negrita:

```
const express = require('express');
const app = express();


const mongodb = require('mongodb');
const config = {
  DB: 'mongodb://mongo:27017'
};

const PORT = 3000;
app.get('/', function(req, res) {
    res.json({"hello": "express with mongo"});
});


var dbo;
const client = mongodb.MongoClient;
```

Usando mongo en nuestro API

```
client.connect(config.DB, function(err, db) {
    if(err) {
        console.log('database is not connected')
    }
    else {
        console.log('connected!!');
        dbo = db.db("midb");
    }
});
app.get('/misdatos', function(req, res){
    let data = dbo.collection("micoleccion").find({}).toArray((err, result) => {
        if (err) throw err;
        res.json(result);
      });
});
app.listen(PORT, function(){
    console.log('Your node js server is running on PORT:',PORT);
});
```

Usando mongo en nuestro API

Ahora para que los cambios tengan efecto debemos parar todo, volver a construir los contenedores y volver a lanzarlos:

```
#    docker-compose down
#    docker-compose build
#    docker-compose up -d
#    docker-compose ps
```

Para poder probar el api debemos cargar con datos la base de datos llamada mibd y la colección de esa base de datos micoleccion

Probamos los entry-point

Mediante la consola de mongo creamos la base de datos:

```
#    mongo
     > use midb
     > db.micoleccion.insert({“titulo”: “primero de prueba”})
     > exit
```

Probamos los entry-point

Tambien podemos importar un fichero con los datos

```
mongoimport --db midb --collection micoleccion --file datos.json
```

El fichero datos.json contendrá información de esta manera:

{ "_id" : 1, "titulo": "primero de prueba"} { "_id" : 2, "titulo": "segundo de prueba"}

Probamos en http://localhost:3000/misdatos y veremos como nos muestra el json de la información que se recupera.

Usar Nginx para servir los estaticos y el API

Ya que tenemos la imagen de nginx instalada y el contenedor corriendo, en vez de tener que usar el puerto 3000 para llamar a nuestro API, podemos facilitarlo con nginx que haga de proxy y redirija una url del puerto 80 al puerto 3000.

Seguimos dentro de nuestro directorio docker_example/api y creamos el directorio config y entramos en él:

```
# mkdir config
# cd config
```

Creando la configuración de nginx

Creamos un archivo llamado **api.conf** con el siguiente contenido:

```
server {
        listen          80;
        server_name     localhost;
        location / {
                root /usr/share/nginx/html;
                index index.html index.htm;
        }
        location /api/ {
            resolver app;
            proxy_pass http://app:3000/;
            proxy_set_header  X-Real-IP  $remote_addr;
            proxy_set_header  X-Forwarded-For $proxy_add_x_forwarded_for;
            proxy_set_header  Host $http_host;
            proxy_redirect    off;
        }
    }
```

Creando la configuración de nginx

Este fichero de configuración reemplazará al que trae por defecto nginx.

Lo único que hemos hecho ha sido añadir el apartado "location /api/" para indicar que cuando se llame a http://localhost/api queremos que haga un "proxy_pass" a "localhost:3000", enmascarando las llamadas a nuestro API.

Nginx en la configuracion de docker-compose

Ahora hay que modificar el fichero docker-compose para que también lance el contenedor de nginx.

Añadimos lo siguiente al final del **docker-compose.yml**

```
web:
   container_name: web
   image: nginx
   volumes:
     - ./config/api.conf:/etc/nginx/conf.d/default.conf
     - ../www:/usr/share/nginx/html
   ports:
     - "80:80"
   links:
     - app
```

Probamos la nueva configuración

```
# docker-compose build
# docker-compose up -d
# docker-compose ps
```

- Podemos entrar en http://localhost y ver como sirve la pagina index.html que creamos en la carpeta /www
- Podemos entrar en http://localhost/api y ver como sirve el mensaje que devuelve el servidor node-express cuando accedemos a http://localhost:3000
- Podemos entrar en http://localhost/api/misdatos y ver como sirve el json de la base de datos que sirve también cuando accedemos a http://localhost:3000/misdatos

Resumiendo

Con **docker** podemos crear entornos de ejecución aislados y configurarlos a nuestro antojo sin romper nada y asegurando compatibilidad. Pudiendo copiar/subir dichos contenedores en la nube o en otros equipos

Con **docker-compose** podemos orquestar todos los contenedores creados, comunicarlos e iniciarlo y pararlos fácilmente.

BONUS EXTRA

¿Qué es Continuous Delivery o Continuous Deployment?

1. Continuous Integration
2. Control de versiones
3. Code Review
4. Configuration Managment
5. Monitorización
6. Orquestación
7. Dashboards