# SGE TEMA 3 DOCKER

**Docker**: es un software que nos permite desplegar aplicaciones en contenedores. Su objetivo no es virtual izar, aísla cada contenedor del otro.

**Contenedor**: es un entorno que nos permite ejecutar varias aplicaciones en el mismo puerto, por ejemplo dos apis en el puerto 9000.
No es lo mismo que las maquinas virtuales, el problema es que consume muchos mas recursos.

-Los contenedores son autosuficientes: se pueden ejecutar ellos mismos.

-Hay algunos contenedores que dependen de otros, por ejemplo el de oracle el cual necesita otro contenedor.

-Son portábles por lo tanto te permite desplegar aplicaciones en el So que quieras. Habiendolo empaquetado una vez lo puedes desplegar donde quieras.

-Al desinstalar e instalar programas no ensuciamos el sistema operativo principal.

**Entorno:** todo lo que rodea a la aplicación que estamos desarrollando. Incluyendo frameworks librerias etc.…

**HIPERVISOR**: sistema operativo dedicado a funcionalidades concretas como la ejecucion de una app. Pero no contiene tantas características generalistas como ubuntu o windows.

**IMÁGENES**: son contenedores pre-configurados mediante los que solo con pequeñas configuraciones tendremos todo lo que necesitamos.

**MEAN:** MONGOOSE EXPRESS ANGULAR NODE
**MEN**: MONGOOSE EXPRESS NODE

**Entorno de desarrollo:** escenario sobre el que los programadores trabajan
**Entorno de producción:** escenario que rodea a la app cuando esta operativa de cara al publico.
Puede ser distinto al de desarrollo.
**Súper ventaja docker:** aunque el entorno de desarrollo y producción sea distinto el contenedor siempre tendrá las mismas versiones por lo tanto no habrá errores de versiones de programas.
**DIFERENCIA ENTRE IMAGEN Y CONTENEDOR:**
**Imagen** es un conjunto de ficheros y librerías osea una plantilla
**contenedor**: imagen puesta en ejecución

## COMANDOS DOCKER:

-sudo docker images: ver imagenes docker instaladas
-sudo docker ps: ver los contenedores
-sudo docker ps -a: igual pero con los contenedores que no estan iniciados
-sudo docker rm borra contenedores
-sudo docker rmi borra imágenes

---

12/12/2018 CREAR CONTENEDORES (dockerizar)
https://picodotdev.github.io/blog-bitix/2014/11/como-crear-una-imagen-para-docker-usando-un-dockerfile/

**FICHERO DOCKER / RECETAS:**
sirven para mediante unos ingredientes y una imagen de base se le añaden x configuraciones  y se crea. Ejemplo:
1º se coge la imagen de fusión, que es un so basado en ubuntu para dockerizar.
2ºse establece la variable de la raíz del sistema
3ºactualiza el sistema
4ºcon run nos regenera las conexiones ssh
5ºadd copia ficheros de configuración de nuestro ordenador hacia el docker
6ºexpone (abre) el puerto 22

```
FROM
phusion/baseimg
e:0.9.15

                MAINTAINER picodotdev <pico.dev@gmail.com>



                ENV HOME /root



                RUN apt-get update -q



                RUN /etc/my_init.d/00_regen_ssh_host_keys.sh



                RUN echo
                'root:$6$l/PahbyY$jFhqIAuvHeK/GwjfT71p4OBBkHQpnTe2FErcUWZ8GIN1y
                kdI7CgL05Jkk7MYW6l.0pijAlfoifkQnLpaldEJY0' | chpasswd -e



                ADD bashrc /root/.bashrc

                ADD timezone /etc/timezone
```

```
EXPOSE 22


CMD ["/sbin/my_init"]
```

SOBRE LA IMAGEN DE BASE QUE PUEDE QUE LA USEMOS PARA MAS COSAS LA DEJAMOS LIMPIA Y CREAMOS OTRA SOBRE LA QUE CONFIGUREMOS, de forma que todo lo que se ha hecho en la anterior se hace aqui aparte de lo que escribamos nuevo:

1º la imagen de base ahora es la que hemos creado.
MAINTAINER: indica el creador de la imagen
2ºse instala mysql
3º copiamos los ficheros necesarios
4º se dan permisos de ejecucion
5º se limpian directorios
6ºvolume: como las imágenes de docker son de **solo lectura**, cada vez que se intente escribir en la ruta del docker se escribirá en la ruta del ordenador.
7ºse limpian y vacían listas
8º se abre el puerto 22 y 3306
9ºmy_init arranca el servicio.
**CMD**: COMANDO QUE SE EJECUTA NADA MÁS ARRANCAR LA MÁQUINA.

```
FROM picodotdev/base:1.0

                                MAINTAINER picodotdev <pico.dev@gmail.com>


                                ENV HOME /root


                                RUN apt-get install -y mysql-server mysql-client


                                ADD my.cnf /etc/mysql/my.cnf

                                RUN mkdir /etc/service/mysql

                                ADD mysql /etc/service/mysql/run
```

```
RUN chmod +x /etc/service/mysql/run



RUN rm -R /var/lib/mysql && \
       mkdir /var/lib/mysql && \
   mkdir /mnt/keys



VOLUME ["/var/lib/mysql", "/mnt/keys"]



RUN apt-get clean && \
       rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/*



EXPOSE 22 3306



CMD ["/sbin/my_init"]
```

## PARA COCINAR LA IMAGEN: ejemplo creando uno de apache

1° creamos carpeta
2°creamos un index con hola mundo
3°Creamos un archivo llamado Dockerfile
4°

usamos esta de plantilla:

```
FROM debian
LABEL AUTHOR="JUAN ANTONIO ORTIZ GUERRA"

RUN apt-get update && apt-get install -y apache2 && apt-get clean && rm
-rf /var/lib/apt/lists/*
```

```
ENV APACHE_RUN_USER www-data
ENV APACHE_RUN_GROUP www-data
ENV APACHE_LOG_DIR /var/log/apache2

EXPOSE 80
ADD ["index.html","/var/www/html/"]

ENTRYPOINT ["/usr/sbin/apache2ctl", "-D", "FOREGROUND"]
```

add-> se guarda el index html en la ruta del contenedor
entrypoint-> para que se inicie apache

5°Generamos la imagen->
Creamos la imagen y con -t le damos el nombre, con el . indica que se hara en el directorio actual

```
sudo docker build -t luismienclase/apache2:1.0 .
```

EJECUTAR LA IMAGEN-> sudo docker run -p 80:80 --name nombreContenedor nombreimagen:tag
ejemplo->
luismienclase/apache2:1.0

SABER CONTENEDORES:
sudo docker container ls -a

sudo docker images-> saber las imagenes

BORRAR CONTENEDOR-> sudo docker container stop nombre
sudo docker container rm nombre

**DOCKER CON SPRING**

**https://spring.io/guides/gs/spring-boot-docker/**

1° CLONAMOS EL REPOSITORIO
2° cd initial
3° añadimos que la clase main sea un rest controller y diga hellow world!!!!
4° empaquetar y ejecutar la app->

```
./mvnw package && java -jar target/gs-spring-boot-docker-0.1.0.jar
```

5°dentro de la carpeta initial creamos un Dockerfile

alpine-> imagen muy ligera de linux para aplicaciones java con 8mb de espacio solo.

volume-> para que nuestro contenedor guarda archivos temporales en temp

arg->
para esperar un argumento, osea el nombre del fichero jar. ->
nombreproyecto-version.jar

copy-> copia desde el ordenador el jarfile obtenido y que lo meta en el contenedor llamado app.jar

entrypoint->> invoca el app.jar con opciones de seguridad

```
6° FROM openjdk:8-jdk-alpine
VOLUME /tmp
ARG JAR_FILE
COPY ${JAR_FILE} app.jar
ENTRYPOINT
["java","-Djava.security.egd=file:/dev/./urandom","-jar","/app.jar"]
```

7° en las properties de maven ponemos el plugin de docker

```
<properties>
   <docker.image.prefix>springio</docker.image.prefix>
</properties>
<build>
    <plugins>
        <plugin>
            <groupId>com.spotify</groupId>
            <artifactId>dockerfile-maven-plugin</artifactId>
            <version>1.4.9</version>
            <configuration>

<repository>${docker.image.prefix}/${project.artifactId}</repository>
            </configuration>
        </plugin>
    </plugins>
</build>
```

8°FORMA MEJORADA:

```
FROM openjdk:8-jdk-alpine
VOLUME /tmp
ARG DEPENDENCY=target/dependency
COPY ${DEPENDENCY}/BOOT-INF/lib /app/lib
```

```
COPY ${DEPENDENCY}/META-INF /app/META-INF
COPY ${DEPENDENCY}/BOOT-INF/classes /app
ENTRYPOINT ["java","-cp","app:app/lib/*","hello.Application"]
```

**EN HELLO APLICATION PONEMOS EL NOMBRE DE LA CLASE QUE EJECUTA LA APP**

***copia todo el codigo fuente del jar por separado para ejecutarlo mas rapido.***

9° en el pom:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-dependency-plugin</artifactId>
    <executions>
        <execution>
            <id>unpack</id>
            <phase>package</phase>
            <goals>
                <goal>unpack</goal>
            </goals>
            <configuration>
                <artifactItems>
                    <artifactItem>
                        <groupId>${project.groupId}</groupId>
                        <artifactId>${project.artifactId}</artifactId>
                        <version>${project.version}</version>
                    </artifactItem>
                </artifactItems>
            </configuration>
        </execution>
    </executions>
</plugin>
```

10° construir imagen

```
./mvnw install dockerfile:build
```

19/12/2018
sudo chmod 777 mvnw-> en la ruta del comando para darle permisos

mvn -> maven
mvnw-> script que envuelve a mvn del shell.
mvn package-> genera un jar en la carpeta target del proyecto.
mvn dockerfile:build-> para que se construya la imagen del tiron.

cuando hagamos mvn dockerfile:build > misproblemas.txt
para que nos saque los errores en un txt.

---

**COMO DOCKERIZAR UNA APLICACIÓN NODE**

https://nodejs.org/en/docs/guides/nodejs-docker-webapp/

Pasos:
1° se usa la ultima version de node lts.
2° ENV-> CREAMOS VARIABLES DE ENTORNO
2° workdir-> cada vez que hagamos un comando lo ejecutaremos en esa ruta del contenedor para no escribirla todo el rato. APARTE SE MUEVE CON UN CD DENTRO AUTOMATICAMENTE PARA EJECUTAR TODOS LOS COMANDOS DENTRO.
3° copiamos el packagejson desde el pc a la imagen
4°ejecutamos npm install que se ejecuta en /urs/src/app gracias al WORKDIR
5° installa gracias al packagejson
6° se copia todo el proyecto hacia toda la ruta del contenedor definida por workdir.
7° cmd-> usa el comando npm start
8° en nuestro proyecto en el config tenemos que poner uri_ process.env.MONGODB_URI para que use esa variable de ruta de mongo.
9° crear en el mismo lugar un .dockerignore para que no copie el node_modules.

```
FROM node:8

ENV PORT 9000

ENV MONGODB_URI url mlab

# Create app directory

WORKDIR /usr/src/app


# Install app dependencies

# A wildcard is used to ensure both package.json AND package-lock.json are copied

# where available (npm@5+)

COPY package*.json ./


RUN npm install

# If you are building your code for production

# RUN npm install --only=production
```

```
# Bundle app source
COPY . .

EXPOSE ${PORT}
CMD [ "npm", "run", "dev" ]
```

## Dockerignore contenido:

```
node_modules
npm-debug.log
```

09/01/2019

Al hacer un contenedor se copia el codigo ejecutable al contenedor pero no el codigo fuente y si estuviese minificado los archivos minificados.

Ejemplo-> jquery.min.js

Minificado: codigo sin espacios con nombres de variables mas cortos y que pesa menos y protege a que no copien el codigo.

Cuando exponemos un puerto definimos a que servicio se accede de la imagen, **los otros se quedan sin abrir***

---

10/01/2019

markdown-> escritura de documentos en un formato **(extension .md)**

**MARKDAOWN CHEAtsheet-> TUTOTIAL.**

#texto -> h1

##texto-> h2

*opcion1-> lista

*opcion2…

```java textoParaQueSalgaComoCodigoFuente

SE PONE EL LENGUAJE PARA QUE RESALTE LAS PALABRAS CLAVES.

**VIDEO 37 volúmenes en docker**

WildFly-> servidor aplicaciones de java

¿Como acceder al contenido del contenedor?

Volumenes: carpetas compartidas entre contenedor y la maquina.

Un contenedor puede tener otro contenedor dentro.

*Ahora en vez de copiar del pc al contenedor MAPEAMOS UN VOLUMEN:***

**MAPEAR VOLUMEN:**

decir que una carpeta en el pc por ejemplo /home/luismi/app, para que cuando trabaje sobre el proyecto se cambie en el contenedor

comando-> `sudo docker run -p 80:80 --name nombreContenedor nombreimagen:tag -v rutaPc:rutaImagen`

**SHARDING-**> particionar la BD mongo en distintos servidores.

**DOCKER COMPOSE:**

nos permite crear a partir de dos o mas imagenes contenedores relacionados entre si en un solo fichero llamado **DOCKER COMPOSE** en formato **YAML.**

***La linea de comando usada de **link** es para que los contenedores se comuniquen.***

---

**11/01/2019 DOCKER COMPOSE**

Ejemplo docker compose con web y redis:

Como pasar variables de entorno al construir una imagen pasando la variable NODE_ENV

PORT

MONGODB_URI

42,43 videos

---
```

**DEFINIR VARIABLES DE ENTORNO DE DOCKER FILE:**

```
ENV nombreVariable=valorVariable
```

ej->

ENV NODE_ENV=development

**Pasar variables a el dockerfile->**

## -DOCKER COMPOSE

1º Crear en un directorio docker-compose.yml
version 2-> la version de docker compose usada
services:-> los servicios que se crean a la vez
    myapp:-> servicio de codeigniter
        image: 'bitnami/codeigniter:latest'
        ports: '8000:8000'
        volumes: '.:/app'-> el directorio actual con el de la app
        depends_on: mariadb-> depende del contenedor mariadb
    mariadb:
        image: 'bitnami/mariadb:latest`
        enviroment:-> variables entorno
    - ALLOW_EMPTY:PASSWORD=yes

2º docker-compose up -d -->crear e iniciar todos los contenedores definidos

se le pone como prefijo de nombre el de la carpeta donde esta y como sufijo un numero*****
**se conecta solo la api con la bd porque las imagenes vienen configuradas**

## OTRO EJEMPLO CON WORDPRESS Y MARIADB

wordpress:
    image: wordpress
    links:

```
    -  mariadb:mysql
    environment:
    - WORDPRESS_DB_PASSWORD=contrasea
    ports:
    - “80:80”
    volumes:
    - ./code:/code
    - ./html:/var/www/html
mariadb:
    image: mariadb
    environment:
    - MYSQL_ROOT_PASSWORD=constrasea
    - MYSQL_DATABASE=wordpress
    volumes:
    - ./database:/var/lib/mysql
```

---

16/01/2019

**CORRECCION EJERCICIO DOCKER 2:**

no hace falta el copy . .

---

Volume en docker file-> es lo mismo pero solo se pone la ruta del contenedor que se compartira con la ruta actual del proyecto.

---

17/01/2019

**EXAMEN DOCKER:**

-Pregunta corta de teoria
-Ejercicio de completar huecos
-Un fichero desde 0
Se usa documentacion oficial docker + chuleta de docker compoose

**EJEMPLO DOCKER COMPOOSE:**

**1º CREAMOS LA IMAGEN DE NODE**

```
############################################################
# Dockerfile para configurar aplicación en node.js - Express
############################################################

# Establece la imagen base
FROM node

# Información de Metadata
LABEL "cl.apgca.appNode"="GCA DESARROLLOS TECNOLOGICOS"
LABEL maintainer="mortega@apgca.cl"
LABEL version="1.0"


# Crear directorio de trabajo
RUN mkdir -p /opt/app

# Se estable el directorio de trabajo
WORKDIR /opt/app

# Instala los paquetes existentes en el package.json
COPY package.json .
RUN npm install --quiet

# Instalación de Nodemon en forma Global
# Al realizarse cambios reiniciar el servidor
RUN npm install nodemon -g --quiet

# Copia la Aplicación
COPY . .

# Expone la aplicación en el puerto 8000
EXPOSE 8000

# Inicia la aplicación al iniciar al contenedor
CMD nodemon -L --watch . app.js
```

**BUILD: .  ->** Usa el dockerfile que este en la misma ruta y lo transforma a contenedor.
**en mongo no se expone el puerto, pero como depende la web de DB y estan en la misma red de contenedores si tendran conexion, PERO NOSOTROS DESDE LOCAL NO PODEMOS**

***DEPENDS_ON PUEDE APARECER COMO LINK, PERO CON DEPENDS_ON PUEDES EJECUTAR LAS DOS COSAS A LA VEZ. CON LIKE TE DEJA LEVANTARLOS SOLOS SI QUIERES.**
*

```
#############################################################
# docker-compose.yml  - aplicación en node.js - Express
#############################################################

#version: '2'

services:
  web:
    build: .
    depends_on:
      - db
    ports:
      - "8000:8000"
    volumes:
      - .:/opt/app
      - /opt/app/node_modules
      # Permite sincronizar la carpeta de trabajo durante el desarrollo
      # de la aplicación
  db:
    image: mongo
    expose:
      - "27017"
    volumes:
      - mongodata:/data/db

volumes:
  mongodata:
```

**VOLUMES-> MONGODATA**:se declara el volumen, te lo pide porque aqui no le dimos la ruta en local.

**COMANDOS PARA CREARLA:**

1º docker-compose build

construimos los contedores

2º arrancamos todos los contenedores:

docker-compose up -d