

Contenedores Docker

Antonio Espín Herranz

Contenidos

- Despliegue de Microservicios C/C++ con Docker.
- Contenedores de microservicios en C/C++ con **Docker**:
 - Creación de Dockerfiles optimizados para aplicaciones C/C++.
 - Gestión de dependencias y bibliotecas externas dentro de los contenedores.

Docker

Docker

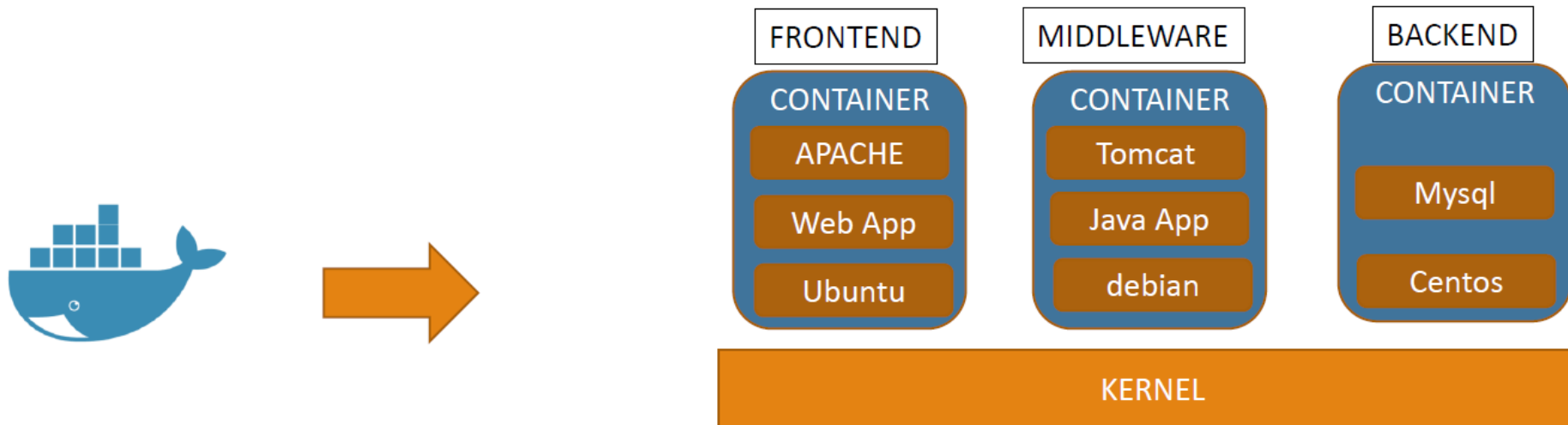
- Es una plataforma de software que permite desarrollar, enviar y ejecutar aplicaciones dentro de contenedores.
- Docker facilita el trabajo de desarrolladores, administradores de sistemas y equipos DevOps al permitir:
 - **Empaquetar aplicaciones** con todas sus dependencias.
 - **Ejecutarlas de forma consistente** en cualquier entorno (desarrollo, pruebas, producción).
 - **Aislar procesos** para evitar conflictos entre aplicaciones.
 - **Escalar fácilmente** en arquitecturas distribuidas como microservicios.

Un contenedor en el Software

- El contenedor empaqueta de forma ligera todo lo necesario para que uno o mas procesos funcionen: código, herramientas del sistema, bibliotecas, dependencias, etc.
- Nos **proporcionan un servicio y todo lo necesario para que ese servicio funcione.**
- Completamente independiente y orientado a microservicios.
 - Están empaquetados de forma individual y son independientes.

Un contenedor en el Software

- La idea es poder llevarnos el contenedor a otro sistema sin tener que hacer ningún tipo de cambio → el contenedor es autosuficiente.
 - **Solo necesita un Runtime de contenedores.**
 - Distintos contenedores proporcionando cada uno distintos servicios.

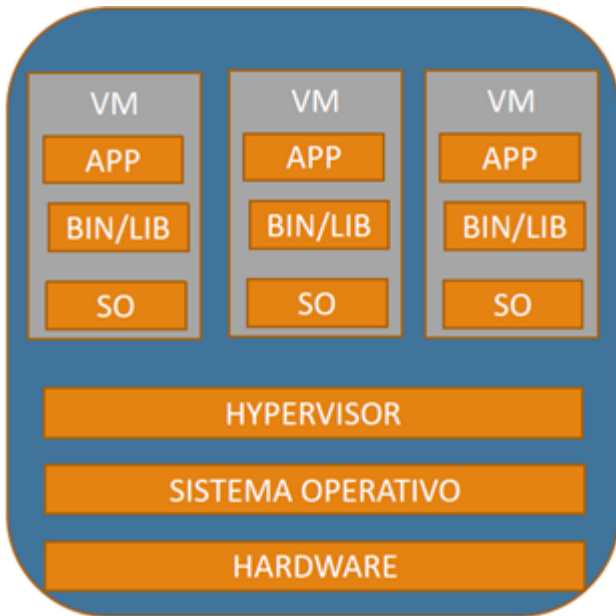


Resumen

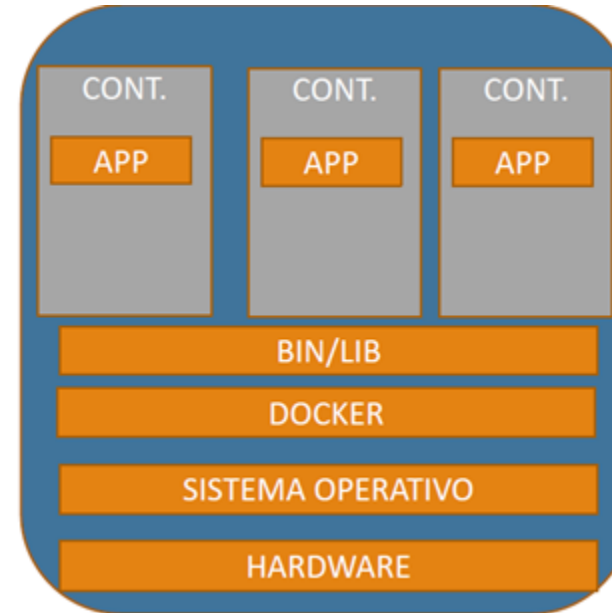
- Un contenedor es:
 - Un **objeto ligero** donde,
 - Se **empaqueta** todo lo necesario
 - Para **ofrecer un servicio**
 - A un tercero.

Docker vs Máquinas virtuales

- Docker **no requiere un S.O. independiente**, en cambio, en una máquina virtual si es necesario.



- Cada M.Virtual es independiente:
 - Tienen un S.O. por cada máquina
 - Librerías y se ejecutan las apps.
- Componentes monolíticos totalmente Independientes unos de otros.



En Docker sería un proceso o Daemon que se ejecuta en el S.O. sería el equivalente al Hypervisor, pero los contenedores Utilizan los recursos de S.O. para poder compartir: lib, bin ...
NO implementan su propio S.O., ni LIBs
LOS CONTENEDORES SON INDEPENDIENTES,
PERO, COMPARTEN RECURSOS!!

Componentes de docker

- **Contenedores**

- Se crea a partir de una imagen. Se puede definir como un proceso que ha sido aislado de todos los demás procesos de la máquina donde se ejecuta (el host de Docker).
- ***Buenas prácticas: 1 sólo proceso en 1 contenedor. 1 → 1***

- **Imágenes**

- Las imágenes contienen el sistema de archivos que utilizarán los contenedores Docker. Para crear un contenedor, es obligatorio utilizar una imagen.
- **Plantilla. 1 imagen → N contenedores**

- **Volúmenes**





- Mecanismo de persistencia para los contenedores. Si un contenedor no tiene un volumen asociado al terminar y eliminar el contenedor los datos se pierden.

- **Redes**

- Docker permite crear diferentes tipos de redes para que los contenedores se puedan comunicar entre ellos. La gestión de redes en Docker se gestiona con **libnetwork**

Docker Hub

- Repositorio de imágenes de Docker (***ojo utilizar imágenes oficiales***).

IMAGE + 1 MORE		IMAGE + 1 MORE	
	postgres 		mysql 
PostgreSQL Docker Official Images		MySQL Docker Official Images	
The PostgreSQL object-relational database system provides reliability and data integrity.		MySQL is a widely used, open-source relational database management system (RDBMS).	
Pulls	1B+	Pulls	1B+
Stars	14530	Stars	15917
Last Updated	8 days	Last Updated	10 days

Opciones para trabajar

- **Dos opciones:**

- 1) Crear una máquina virtual por ejemplo con **Ubuntu** e instalar **Docker Engine** → trabajaremos en modo comando.
- Si estamos en una máquina Linux (instalar directamente).

- 2) **Docker desktop** para **Windows** o **Mac**

- Incluyen docker engine

- **Docker desktop for Linux (no se aconseja)**, está montado sobre una máquina virtual. Es mejor la opción Docker Engine.

- Como no está soportada la virtualización anidada esta opción solo se puede montar en una máquina Linux (real).
- No se puede probar dentro de una máquina virtual y que se estuviera ejecutando dentro de Windows.

- **Ver:**

- <https://www.docker.com/blog/how-to-check-docker-version/#::~:~:text=Users%20can%20leverage%20the%20Docker,Docker%20CLI%20to%20run%20commands>.

Dockerfile

Dockerfile

- Es un fichero de texto que se **utiliza para definir la imagen de un contenedor Docker**.
- Se compone de instrucciones para construir una imagen personalizada.
- Dockerfile nos sirve para:
 - **Desarrollar aplicaciones:** Crear un entorno estandarizado con las versiones exactas de software y dependencias necesarias para ejecutar tu aplicación.
 - **Automatizar la configuración y construcción de entornos**
 - **Microservicios:** Diseñar imágenes ligeras y optimizadas para cada servicio.
 - **Testing y debugging:** Configurar entornos de prueba controlados.

DockerFile comandos I

FROM : Especifica la imagen base que se usará para construir la nueva imagen. Por ejemplo, `FROM ubuntu:20.04` .

WORKDIR : Define el directorio de trabajo dentro del contenedor donde se ejecutarán los comandos posteriores. Ejemplo: `WORKDIR /app` .

COPY : Copia archivos o directorios desde el sistema host al contenedor. Ejemplo: `COPY . /app` .

ADD : Similar a **COPY** , pero también puede descargar archivos de URL y descomprimir archivos. Ejemplo: `ADD app.tar.gz /app` .

RUN : Ejecuta comandos durante la construcción de la imagen, como instalar dependencias. Ejemplo: `RUN apt-get update && apt-get install -y python3` .

CMD : Define el comando por defecto que se ejecutará al iniciar el contenedor. Ejemplo: `CMD ["python", "app.py"]` .

ENTRYPOINT : Similar a **CMD** , pero más fijo y permite agregar argumentos. Ejemplo: `ENTRYPOINT ["python"]` .

DockerFile comandos II

ENV : Configura variables de entorno dentro del contenedor. Ejemplo: **ENV**
PORT=8080 .

EXPOSE : Indica los puertos en los que el contenedor está configurado para escuchar. Ejemplo: **EXPOSE 8080** .

VOLUME : Define puntos de montaje para datos persistentes. Ejemplo: **VOLUME**
/data .

ARG : Permite pasar variables al construir la imagen. Ejemplo: **ARG VERSION=1.0** .

LABEL : Agrega metadatos a la imagen, como autor o versión. Ejemplo: **LABEL**
maintainer="tuemail@example.com" .

USER : Cambia al usuario que ejecutará los comandos dentro del contenedor. Ejemplo: **USER appuser** .

ONBUILD : Especifica instrucciones que se ejecutarán al usar la imagen como base en otro Dockerfile.

Ejemplo

```
# Imagen base
FROM python:3.9-slim

# Establecer el directorio de trabajo dentro del contenedor
WORKDIR /app

# Copiar el archivo de dependencias (requirements.txt) al contenedor
COPY requirements.txt .

# Instalar las dependencias requeridas
RUN pip install --no-cache-dir -r requirements.txt

# Copiar los archivos de la aplicación al contenedor
COPY . .

# Especificar el comando para ejecutar la aplicación
CMD ["python", "app.py"]
```

FROM python:3.9-slim: Usamos una imagen base ligera de Python.

WORKDIR /app: Configura el directorio de trabajo dentro del contenedor.

COPY requirements.txt .: Copia el archivo `requirements.txt` para instalar las dependencias.

RUN pip install: Instala las librerías definidas en `requirements.txt`.

COPY . .: Copia todos los archivos del directorio actual al contenedor.

CMD: Define el comando por defecto para ejecutar la aplicación (`app.py` en este caso).

El fichero requirement.txt contendrá los nombres de las librerías de Python habría que instalar en el contenedor:

- pandas
- Flask
- Etc.

Dockerfile (ENTRYPOINT vs CMD)

- **ENTRYPOINT**

- Propósito: **Define el programa principal que siempre se ejecutará en el contenedor.** Es el “punto de entrada” fijo.
- Formato: Puede escribirse en dos formas:
 - **Lista** ejecutable: `ENTRYPOINT ["executable", "param1", "param2"]` (forma recomendada).
 - Forma de **cadena**: `ENTRYPOINT command param1 param2` (menos común y limitada a `/bin/sh`).

Dockerfile (ENTRYPOINT vs CMD)

- **CMD**

- Propósito: **Especifica el comando por defecto que se ejecutará cuando se inicie el contenedor.** Puede ser sobrescrito al ejecutar el contenedor.
- Formato: También puede usarse en dos formas:
 - Lista ejecutable: CMD ["executable", "param1", "param2"].
 - Forma de cadena: CMD command param1 param2 (limitada a /bin/sh).
- Comportamiento:
 - Es más flexible que ENTRYPOINT, ya que puedes **sobrescribir** el **comando** al ejecutar el contenedor.
 - Si ejecutamos: docker run imagen ls (ejecuta ls en vez del comando especificado en CMD)

Buenas prácticas en un DockerFile I

- Utilizar las **imágenes ligeras** siempre que se pueda:
 - Utilizar alpine o las que terminan en Slim
 - FROM python:**3.9-slim** mejor que FROM python:3.9
- **Minimizar** el número de **capas** de la imagen. Cada instrucción ADD, COPY, etc. Crea una nueva capa.
 - Por ejemplo, se pueden unir comandos → RUN apt-get update && apt-get install -y curl ...
- **Eliminar** archivos **temporales**
- Se puede utilizar **.dockerignore** (se pueden excluir archivos innecesarios)

Buenas prácticas en un DockerFile II

- Fijar **versiones** de las **librerías**: flask==2.1.0 en un archivo: requirements.txt (por ejemplo, en Python)
- Utilizar mejor **COPY** que ADD. ADD permite descomprimir archivos.
- Evitar el usuario root. Añadir otros usuarios (por seguridad):
 - **RUN** useradd -m appuser
 - **USER** appuser
- **Eliminar** las herramientas innecesarias (que no tengan sentido en producción)
- Utilizar **comentarios** dentro del fichero.
 - Las líneas se preceden con una #

En C++

- **Crear un Dockerfile personalizado**

- Se define una imagen base (como ubuntu, debian o alpine) que tenga las herramientas necesarias para compilar y ejecutar C++.

- **Instalar dependencias**

- Se instalan compiladores como g++, herramientas de construcción como cmake o make, y cualquier librería adicional que la aplicación necesite.

- **Copiar el código fuente**

- Se copia el código fuente al contenedor y se compila dentro del mismo, o se copia el binario ya compilado si se prefiere compilar fuera.

- **Exponer puertos**

- Si el microservicio C++ ofrece una API (por ejemplo, con gRPC o REST), se expone el puerto correspondiente con EXPOSE.

- **Definir el punto de entrada**

- Se usa CMD o ENTRYPOINT para ejecutar el binario principal del servicio.

- FROM ubuntu:20.04 # **La imagen base**
- **# Instalar herramientas necesarias**
- RUN apt-get update && apt-get install -y **g++ cmake make**
- **# Crear directorio de trabajo**
- WORKDIR /app
- **# Copiar el código fuente**
- COPY ..
- **# Compilar la aplicación**
- RUN make
- **# Exponer el puerto del microservicio**
- EXPOSE 8080
- **# Ejecutar el servicio**
- CMD ["./mi_microservicio"]

docker-compose

Introducción

- Docker Compose es una **herramienta** que se utiliza para definir y manejar aplicaciones Docker con **múltiples contenedores** de manera sencilla.
- Sirve para **automatizar** el **despliegue** y la configuración de estos **contenedores**, lo cual es muy útil cuando tienes varios servicios que necesitan comunicarse entre sí, como una base de datos, un backend y un frontend.
- Con Docker Compose podemos:
 - Definir la infraestructura en un archivo **YAML** (**docker-compose.yml**), especificando los contenedores, redes, volúmenes, etc.
 - **Levantar** todos los **servicios** con un solo comando (docker-compose up).
 - Normalmente se lanza el comando con la opción **-d** (detached, en 2º plano)
 - **docker-compose up -d**
 - **Simplificar** la **gestión de servicios complejos**, eliminando la necesidad de configurar manualmente cada contenedor.

docker compose

- El **comando ya se instala en Windows al instalar Docker desktop.**
 - La versión 1 se hizo en Python
 - La versión 2 en go y que se integre como un comando de Docker.
- Se puede comprobar desde una consola lanzando el comando:
 - **docker compose** (*tiende a utilizarse esta*)
 - **docker-compose**
 - Uno u otro funcionan desde la consola de Windows

```
C:\Users\Anton>docker compose version
Docker Compose version v2.32.4-desktop.1

C:\Users\Anton>docker-compose version
Docker Compose version v2.32.4-desktop.1
```

Fichero de configuración

- El fichero que busca por defecto se llama: **docker-compose.yml**
- También **compose.yml**
- Se puede cambiar el nombre con la opción **-f**
 - `docker compose -f otro_fichero.yml`
- Para iniciar y ejecutar los servicios se utiliza la opción: **up**
 - **docker compose up**
 - Busca el fichero por defecto e inicia y ejecuta servicios

docker compose up

- Comportamiento de docker-compose **up**:
 - **Construcción de imágenes**: Si no existen imágenes para los servicios definidos, las construirá usando las instrucciones del Dockerfile.
 - **Creación de contenedores**: Inicia los contenedores necesarios para los servicios especificados.
 - **Conexión de redes**: Configura las redes entre los contenedores según lo indicado en el archivo.
 - **Persistencia de volúmenes**: Crea y conecta los volúmenes definidos para almacenar datos.

Comandos relacionados

- **docker-compose up**: Inicia los servicios y muestra los logs directamente en la terminal.
- **docker-compose up -d**: Ejecuta los servicios en segundo plano (modo "detached").
- **docker-compose up --build**: Fuerza la reconstrucción de las imágenes antes de iniciar los servicios.
 - Si hemos estado haciendo cambios en los dockerfile.
- **docker-compose down**: Elimina contenedores y redes
- **docker-compose down -v**: Además de los contenedores y las redes también borra los volúmenes.

Apartados del fichero

- Sólo es obligatorio el apartado de services:
- versión → version: “**3.9**” # **No es obligatorio poner la versión**
- **services** → equivale a los contenedores
- volumes
- networks
- configs
- secret

- Comentarios con #

YAML

- Formato de serialización
- Se utiliza para definir listas, propiedades, etc.
- <https://es.wikipedia.org/wiki/YAML>
- Trabaja con indentaciones como en Python, ojo, no utilizar tab, es con espacios en blanco.

Ejemplo

- El directorio donde especificamos la estructura y se encuentra el fichero `compose.yml` se trata como un proyecto.
- Mantener carpetas separadas con cada proyecto.
 - `pr_nginx`
 - `compose.yml`
 - `version: '3.9'`
 - `services:`
 - `nginx:`
 - `image: nginx` (aquí también se puede indicar `build` para construir la imagen a partir de un `Dockerfile`)
 - `ports:`
 - `"80:80"`
- Se recomienda utilizar nuestras propias redes (bridge personalizadas)
 - Si no se indica la red construye una por defecto.

Services

- Es el apartado clave donde defines los servicios que compondrán tu aplicación.
- **Cada servicio corresponde a un contenedor Docker.**
- Dentro de **cada servicio** puedes **definir**:
 - **image**: Imagen base a usar.
 - **build**: Configuración para construir la imagen desde un Dockerfile.
 - **ports**: Mapeo de puertos entre el host y el contenedor.
 - **volumes**: Volúmenes para persistir o compartir datos.
 - **environment**: Variables de entorno.
 - **depends_on**: Dependencias de otros servicios.

Services

- Podemos tener varios:
 - services:
 - servicio_1:
 - # configuración s1
 - servicio_2:
 - # configuración s2
 - servicio_3:
 - # configuración s3

Ejemplo

```
services:
  web:
    image: nginx
    ports:
      - "8080:80"
  db:
    image: mysql
    environment:
      - MYSQL_ROOT_PASSWORD=example
```

Ejemplo 2

```
version: "3.9"

services:
  database:
    image: mysql:8.0
    container_name: mi_base_datos
    environment:
      MYSQL_ROOT_PASSWORD: supersegura
      MYSQL_DATABASE: mi_app
    ports:
      - "3306:3306"
    volumes:
      - datos_db:/var/lib/mysql

volumes:
  datos_db:
```

- **Servicio database:**

- Se utiliza la imagen oficial de MySQL, versión 8.0.
- Se configura con las variables de entorno para establecer la contraseña del usuario root y crear una base de datos llamada mi_app.
- Se mapea el puerto 3306 del contenedor al host para que sea accesible.

- **Volumen datos_db:**

- Está definido en la sección **volumes** del archivo.
- Este volumen se monta en **/var/lib/mysql**, que es donde MySQL almacena los datos en el contenedor. Esto asegura que los datos persistan incluso si el contenedor se detiene o elimina.

- **Persistencia:**

- Los datos almacenados en **datos_db** permanecerán disponibles, ya que Docker gestiona el volumen fuera del ciclo de vida del contenedor.

Volumes

- Define volúmenes que pueden ser utilizados por los servicios para persistir datos o compartir información entre contenedores.

```
volumes:  
  datos_app:
```

Networks

- Especifica redes personalizadas que permiten la comunicación entre los servicios o con el host.

```
networks:  
  red_interna:
```

Configs

- Define configuraciones que los servicios pueden consumir, como archivos de configuración.
- Se suele utilizar con Docker Swarm (similar a Kubernetes)

```
configs:  
  app_config:  
    file: ./config/app.conf
```

Secret

- Permite manejar información sensible, como contraseñas, de manera segura.
- Se suele utilizar con Docker Swarm (similar a Kubernetes)

```
secrets:  
  db_password:  
    file: ./password.txt
```

Fichero entero

- Tiene 3 servicios
 - web, app y db
- Cada servicio va en un contenedor distinto.

```
version: "3.9"

services:
  web:
    image: nginx
    ports:
      - "8080:80"
    networks:
      - red_interna

  app:
    build:
      context: ./app
    depends_on:
      - db
    environment:
      - DATABASE_URL=mysql://db:3306
    networks:
      - red_interna

  db:
    image: mysql
    environment:
      - MYSQL_ROOT_PASSWORD=example
    volumes:
      - datos_db:/var/lib/mysql
    networks:
      - red_interna

volumes:
  datos_db:

networks:
  red_interna:
```


Comandos docker compose

- Los comandos permiten administrar el ciclo completo de una aplicación:
 - **docker compose up**
 - Crea e inicia los contenedores. Con `-d` se ejecutan en 2 plano
 - **docker compose down**
 - Detiene los contenedores que están en ejecución y elimina las redes.
 - Con `-v` elimina todos los volúmenes (de la sección de volumes) y los creados por los contenedores.
 - **docker compose ps**
 - Estado de los contenedores de la aplicación
 - **docker compose logs**
 - Muestra stdout (salida estándar) y stderr (errores) de los contenedores
 - Con `-f` actualizar el tiempo real.
 - **docker compose exec**
 - Ejecuta un comando en un contenedor que está en ejecución

Comandos docker compose II

- **docker compose top**
 - Los procesos que se están ejecutando en cada uno de los contenedores de la aplicación.
- **docker compose start**
 - Inicia los servicios (contenedores) de la aplicación
- **docker compose stop**
 - Detiene los servicios (contenedores) de la aplicación
- **docker compose build**
 - Crea las imágenes de los servicios que utilizan ficheros dockerFile, en lugar de utilizar la imagen de un repositorio (Docker hub)

Archivos con variables de entorno

- Hay veces que necesitamos utilizar variables de entorno dentro del archivo de configuración y las podemos pasar a través de otro fichero: **.env**
- Estos ficheros **tienen que estar en el mismo directorio** que el fichero: docker-compose.yml
- Formato:
 - VARIABLE=valor
 - Líneas que empiezan con # están comentadas

Archivos con variables de entorno

- Fichero:
 - MYSQL_ROOT_PASSWORD=root
 - MYSQL_DATABASE=database
 - ...
- Para utilizarlo:
 - services
 - mysql:
 - environment:
 - MYSQL_ROOT_PASSWORD=\${MYSQL_ROOT_PASSWORD}

Apéndice

- **1. docker build** – To build Docker Image from Dockerfile
- **2. docker pull** – To pull Docker Image from Docker Hub Registry
- **3. docker tag** – To add Tag to Docker Image
- **4. docker images** – To list Docker Images
- **5. docker push** – To push Docker Images to repository
- **6. docker history** – To show history of Docker Image

Apéndice II

- **7. docker inspect**– To show complete information in JSON format
- **8. docker save** – To save an existing Docker Image
- **9. docker import** – Create Docker Image from Tarball
- **10. docker export** – To export existing Docker container
- **11. docker load**– To load Docker Image from file or archives
- **12. docker rmi**– To remove docker images