IES Miguel Romero Esteo

Curso 2025/26

Unidad 3 - Dockerfiles

UNIDAD 3 – DOCKERFILES

ÍNDICE

1.- Estructura de Dockerfile

Un Dockerfile es un archivo de texto que contiene una serie de instrucciones que Docker utiliza para construir una imagen. En otras palabras, es como una “receta” que le dice a Docker qué pasos debe seguir para crear un entorno listo para ejecutar una aplicación.

En un Dockerfile se indican cosas como:

- Imagen base: por ejemplo, FROM ubuntu:20.04 indica que se parte de una imagen de Ubuntu.
- Instalación de dependencias: comandos como RUN apt-get install -y python3 sirven para instalar programas necesarios.
- Copiar archivos: COPY . /app permite copiar el código de la aplicación dentro de la imagen.
- Configurar el directorio de trabajo: WORKDIR /app establece dónde se ejecutarán los comandos.
- Definir el comando principal: con CMD ["python3", "app.py"] se indica qué se ejecutará cuando se inicie un contenedor basado en esa imagen.

☞ En resumen:

Un Dockerfile es la plantilla a partir de la cual se construyen imágenes Docker. Gracias a él, se pueden crear entornos reproducibles, portables y automatizados, lo que facilita mucho el despliegue de aplicaciones en distintos sistemas.

En el siguiente codigo vemos los parámetros más habituales que se suelen escribir

en un fichero dockefile

```
# 1. Imagen base
FROM <imagen_base>[:tag]

# 2. Metadatos (opcional)
LABEL maintainer="tu_email@dominio.com"
LABEL version="1.0"
LABEL description="Descripción de la imagen"

# 3. Variables de entorno
ENV NOMBRE=valor

# 4. Copiar/añadir archivos al contenedor
COPY ./archivo_local /ruta_en_contenedor/
ADD ./archivo.tar.gz /ruta_en_contenedor/     # (similar a COPY, pero con más funciones)

# 5. Establecer directorio de trabajo
WORKDIR /ruta/de/trabajo

# 6. Instalar dependencias o configurar el entorno
RUN apt-get update && apt-get install -y paquete1 paquete2
RUN pip install -r requirements.txt

# 7. Configurar puertos expuestos
EXPOSE 8080

# 8. Definir el usuario que ejecutará los procesos (opcional)
USER nombre_usuario

# 9. Variables de build
ARG VAR_DE_BUILD=valor

# 10. Comandos por defecto (ejecución)
CMD ["comando", "arg1", "arg2"]
```

Ejemplos del uso de estos parámetros son:

```
# Crear un usuario no root
RUN addgroup -S appgroup && adduser -S appuser -G appgroup
```

Hay que tener en cuenta que cada RUN añade una capa a la imagen, por eso resulta interesante agruparlas todas en una sola capa.

Otro ejemplo podría ser:
Copiamos un script de instalación
COPY setup.sh /usr/local/bin/setup.sh
Lo ejecutamos
RUN bash /usr/local/bin/setup.sh

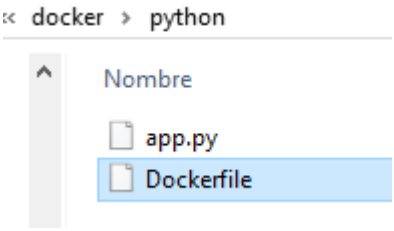
También,

Crea un directorio y cambia permisos
RUN mkdir -p /app/logs && chmod 755 /app/logs

2.- Creando una imagen

Vamos a crear una imagen a partir de la imagen de python de forma que el directorio de trabajo contenga ficheros fuente en python.

En primer lugar, creamos una carpeta para nuestro proyecto y en dicha carpeta creamos un fichero en python y el fichero dockerfile



EL fichero app.py es el script que vamos a pasarle al contenedor para que lo ejecute y su contenido es simplemente

```
print("¡Hola Mundo desde Docker!")
```

Ahora creamos el fichero Dockerfile

```
# Usamos Python como base
FROM python:3.10-slim

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

# Copiamos nuestro script al contenedor
COPY app.py .

# Indicamos el comando a ejecutar cuando arranque el contenedor
CMD ["python", "app.py"]
```

Ya tenemos todo lo necesario para construir la imagen. Ahora nos vamos a la carpeta de nuestro proyecto y lanzamos el comando

docker build -t hola_mundo_python .

Fíjate que no es necesario decirle donde está el fichero Dockerfile ya que le estamos diciendo que la busque en el mismo directorio desde el que lanzamos el comando.

Una vez que hemos construido la imagen, la lanzamos en un contenedor

docker run --rm hola_mundo_python

Se crea un contenedor con python y se ejecuta el script dando como resultado

¡Hola Mundo desde Docker!

En Docker, un contenedor vive mientras el proceso principal está en ejecución. En este caso, ese proceso es python app.py. Como el script solo imprime un mensaje y termina, el proceso acaba inmediatamente → el contenedor se detiene y desaparece.

Si quieres que quede "encendido" aunque el script termine, puedes ejecutar algo como:

docker run -it hola_mundo_python bash

Esto abrirá una terminal dentro del contenedor en vez de ejecutar solo el script. En Docker el contenedor no es una máquina virtual que se queda encendida por sí sola, sino un proceso. Cuando ese proceso termina → el contenedor se apaga. Utilizando este comando se crea el contenedor con un nombre arbitrario a partir de la imagen creada y se abre una shell.

Si queremos que el contenedor tenga un nombre concreto le añadimos el parámetro –name micontenedor. Así,

docker run -it --name python hola_mundo_python bash
root@62997c697e38:/app#

Crea el contenedor con nombre python, listo para recibir órdenes en la shell bash abierta.

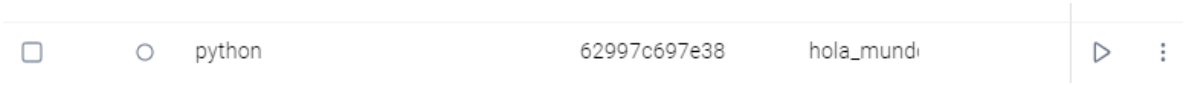
Lo podemos ver desde Docker Desktop

O bien desde la línea de comandos

docker ps

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
a6891d163aea	hola_mundo_python	"bash"	14 seconds ago	Up

Si nos salimos de la shell con exit simplemente detenemos el contenedor pero no lo hemos eliminado



Ten en cuenta que no puedes tener dos contenedores con el mismo nombre al mismo tiempo. Si lo intentas, Docker te dirá que ya existe. Por tanto, deberíamos eliminarlo después de usarlo:

docker rm python

3.- Escribiendo ficheros

Cuando trabajamos con el contenedor para ejecutar scripts tenemos dos opciones,entrar en el contenedor y editar el script que queremos lanzar o bien trabajar en el local y copiar después los ficheros editados al contenedor.

Para esta segunda opción tenemos el comando

docker cp archivo_local.txt mi_contenedor:/ruta/dentro/contenedor/

que en nuestro caso quedaría

PS C:\docker\python> docker cp ejer1.py python:/app/
Successfully copied 2.05kB to python:/app/

Con esto hemos copiado un nuevo script llamado ejer1.py en la carpeta de trabajo del contenedor. Si ahora nos vamos al contenedor tendremos
root@a6891d163aea:/app# ls
app.py ejer1.py

y podríamos ejecutar este nuevo script

root@a6891d163aea:/app# python ./ejer1.py

4.- Buenas prácticas

En esta sección te muestro algunas sugerencias para reducir el tamaño de las imágenes eliminando capas innecesarias.

1. **Combinar comandos en una sola capa.**

Cada RUN crea una capa nueva en la imagen. En lugar de escribir:

RUN apt-get update
RUN apt-get install -y curl git
RUN apt-get clean

Deberías escribir

RUN apt-get update && apt-get install -y curl git && rm -rf /var/lib/apt/lists/*

Resultado: menos capas y menos espacio ocupado.

2. **Evitar instalar dependencias innecesarias**

```
RUN apt-get update && apt-get install -y --no-install-recommends \
      git \
      wget \
  && rm -rf /var/lib/apt/lists/*
```

3. **Instalar dependencias de Python/Node con limpieza**

RUN pip install --no-cache-dir -r requirements.txt

Esto evita que queden cachés innecesarias en la imagen.

4. **Encadenar creación de directorios y permisos**

En lugar de escribir
RUN mkdir -p /app
RUN chown -R appuser:appuser /app

Deberíamos plantearlo como

RUN mkdir -p /app && chown -R appuser:appuser /app

Eliminando así capas y haciendo que la imagen sea más ligera.

5. **Ejecutar scripts en un solo RUN**

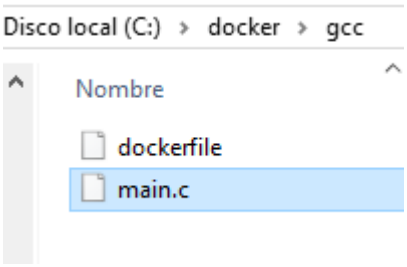
Por ejemplo,

RUN bash /tmp/setup.sh && rm /tmp/setup.sh

5.- Ejercicios

Ejercicio 1.- Crear una imagen basada en Alpine pero con un compilador gcc instalado

Creamos una carpeta para nuestro proyecto



El fichero dockerfile sería

```
# Imagen base: Alpine
FROM alpine:latest

# Instalamos gcc, libc-dev y make (para compilar programas en C)
```

```
RUN apk update && apk add --no-cache \
    gcc \
    g++ \
    make \
    musl-dev

# Definimos el directorio de trabajo
WORKDIR /app

# Copiamos el código fuente (por ejemplo main.c) al contenedor
COPY . /app

# Comando por defecto: mostrar versión de gcc
CMD ["gcc", "--version"]
```

También creamos el fichero main.c que vamos a copiarlo en el directorio de trabajo en el contenedor

```
#include <stdio.h>

int main() {
    printf("Hola desde C en Docker!\n");
    return 0;
}
```

Lanzamos el contenedor abriendo una consola interactiva para poder trabajar

docker run -it --name mi_gcc -v ${PWD}:/app alpine-gcc sh

El significado de los parámetros es el siguiente:
-it → modo interactivo con terminal.
--name mi_gcc → le damos un nombre al contenedor.
-v $(pwd):/app → monta tu carpeta actual en /app dentro del contenedor (ahí estará tu main.c).
alpine-gcc → la imagen que creaste.
sh → abre un shell dentro del contenedor.

En la shell podemos ver que en la carpeta de trabajo se ha copiado directamente todos los ficheros de la carpeta de trabajo del local

```
/app # ls
dockerfile   main.c
```

Ahora lo compilamos

```
/app # gcc main.c -o programa.exe
/app # ls
dockerfile      main.c          programa.exe
```

Ejercicio 2.- Crear una imagen Alpine con un usuario sin privilegios definiendo su carpeta de trabajo

Dockerfile

```
FROM alpine:3.20

# Crear usuario
RUN addgroup -S appgroup && adduser -S appuser -G appgroup

# Crear carpeta y asignar permisos
RUN mkdir /app && chown -R appuser:appgroup /app

WORKDIR /app
USER appuser
```

Construimos la imagen a partir de este dockerfile

```
docker build -t alpine-user .
```

y la lanzamos en un contenedor

```
docker run -it --name mi_alpine -v ${PWD}:/app alpine-user sh
```

y se nos abrirá una shell para el usuario appuser en el directorio de trabajo app. El parámetro

```
${PWD}:/app
```

Le dice a Docker que nuestra carpeta de trabajo en el local se enlace con la carpeta de trabajo en el remoto de forma que si yo creo un nuevo fichero en mi directorio de

trabajo, automáticamente se replica en el contenedor.

Podemos ver que el usuario y el directorio de entrada son los que le indicamos a Docker en el dockerfile

```
/app $ pwd
/app
/app $ whoami
Appuser
```

También podemos comprobar que el usuario está dado de alta en el sistema

```
/app $ cat /etc/passwd
appuser:x:100:101:Linux User,,,:/home/appuser:/sbin/nologin
```

Vamos a explorar la red

```
/app $ ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:AC:11:00:02
          inet addr:172.17.0.2  Bcast:172.17.255.255  Mask:255.255.0.0
```

o hacer un ping al exterior

```
/app $ ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8): 56 data bytes
64 bytes from 8.8.8.8: seq=0 ttl=42 time=20.617 ms
64 bytes from 8.8.8.8: seq=1 ttl=42 time=16.205 ms
```