**DOCKERIZAR UNA APLICACIÓN.**

# INTRODUCCIÓN

En este documento vamos a aprender a Dockerizar una aplicación desde cero.

Nos centraremos en una aplicación en Python, pero lo que aquí aprendamos nos servirá, con unos mínimos ajustes, para trabajar con cualquier lenguaje y contexto.

Empezaremos dockerizando una aplicación sencilla, es decir, sin base de datos, para entender las técnicas usadas y capacitarnos para emplearlas en nuestros propios proyectos.

Después, en la segunda parte de este manual, veremos como Dockerizar una aplicación que dependa de una base de datos, para Dockerizar ambas: la aplicación y la base de datos.

Y ya sin más, vamos a ello.

# 01 – DOCKERIZAR UNA APLICACIÓN SIMPLE

## DOCKERIZAR UNA APLICACIÓN

Empecemos viendo una idea general.

1. **Dockerizar una aplicación:** Dockerizar una aplicación implica crear una imagen para su distribución. Sin embargo, la dockerización también se refiere a la práctica de empaquetar una aplicación y sus dependencias en un contenedor Docker para lograr una ejecución consistente y aislada.
2. **Imagen y contenedor:** Una imagen Docker es un paquete estático que contiene la aplicación y sus dependencias, pero no ejecuta la aplicación. Los contenedores son instancias en ejecución de esas imágenes. Los contenedores son entornos aislados que contienen la aplicación y todas las dependencias necesarias para su ejecución.
3. **Lenguaje base:** En el contexto de Docker, la imagen suele estar basada en un sistema operativo y una versión específicos del lenguaje de programación utilizado para la aplicación. La imagen base incluye tanto el sistema operativo como el entorno de ejecución del lenguaje (por ejemplo, Python, Node.js, Java, etc.). Por lo tanto, la imagen base proporciona el lenguaje base sobre el que se ha construido la aplicación.

En resumen. Dockerizar una aplicación implica crear una imagen de contenedor que incluye la propia aplicación y todas sus dependencias, incluyendo el sistema operativo y el lenguaje base sobre el que ha sido creada. A partir de esta imagen, se pueden crear contenedores para ejecutar la aplicación de manera aislada y consistente en diferentes entornos.

## DOCKER VS VM

Docker es una alternativa para compartir aplicaciones plenamente operativas con varias ventajas frente al uso de VM’s (Virtual Machines)

Las principales diferencias entre una imagen Docker y una VM son las siguientes:

1. **Tamaño y eficiencia:** Las imágenes de Docker son mucho más pequeñas que una VM, ya que comparten el kernel del sistema operativo del anfitrión y solo incluyen las bibliotecas y dependencias necesarias para la aplicación específica. Las VMs incluyen un sistema operativo completo y son más pesadas en términos de recursos y espacio en disco.
2. **Arranque más rápido:** Los contenedores de Docker se inician en segundos, mientras que las VMs pueden tardar más en arrancar. Esto hace que los contenedores sean más ágiles y escalables.
3. **Aislamiento:** Los contenedores de Docker proporcionan aislamiento a nivel de proceso y recursos, mientras que las VMs ofrecen aislamiento a nivel de sistema operativo. Esto significa que los contenedores son más eficientes en la utilización de recursos y permiten una mayor densidad en un sistema anfitrión.
4. **Portabilidad:** Las imágenes de Docker son altamente portátiles y pueden ejecutarse en cualquier sistema que ejecute Docker, siempre que estén diseñadas para la misma arquitectura. Las VMs suelen estar diseñadas para una plataforma de hipervisor específica y pueden no ser tan portátiles.

En resumen, aunque las imágenes de Docker incluyen un sistema operativo, se enfocan en la eficiencia, la portabilidad y el aislamiento a nivel de proceso, lo que las hace muy diferentes de las máquinas virtuales tradicionales. Docker es una tecnología poderosa para crear entornos de contenedor livianos y portátiles que permiten ejecutar aplicaciones de manera aislada y eficiente.

**ESCALABILIDAD**

En el contexto de la dockerización de aplicaciones, la escalabilidad se refiere a la capacidad de aumentar o disminuir la cantidad de contenedores en función de la demanda de tu aplicación. Esto significa que puedes ajustar dinámicamente el número de contenedores que ejecutan tu aplicación para manejar cargas de trabajo más pesadas o ligeras de manera eficiente.

La escalabilidad en Docker es una característica importante que permite que una aplicación sea más flexible y receptiva a las variaciones en la demanda. Puedes crear o eliminar contenedores fácilmente según sea necesario, lo que ayuda a mantener un rendimiento constante y ahorra recursos cuando no se necesitan.

En términos prácticos, esto se logra utilizando orquestadores de contenedores, como Docker Compose o Kubernetes (temas que desarrollaremos más adelante en este mismo documento), que pueden gestionar automáticamente la creación, el escalado y la eliminación de contenedores en función de las métricas de carga y los requisitos de la aplicación. Esto se conoce como escalabilidad horizontal, que implica agregar o eliminar instancias de la aplicación en lugar de modificar una instancia existente.

La escalabilidad es especialmente útil en aplicaciones modernas y en entornos de nube, donde la demanda puede variar constantemente. Docker y los orquestadores de contenedores brindan una forma eficaz de administrar y escalar aplicaciones de manera eficiente y dinámica.

## CREAR EL DOCKERFILE.

El primer paso es crear un fichero con el nombre Dockerfile, en el mismo directorio del proyecto. Este fichero tendrá el siguiente aspecto en lineas generales.

# Usar una imagen base de Python

FROM python:3.11.2

# Establecer el directorio de trabajo en el contenedor

WORKDIR /app

# Copiar el archivo requirements.txt e instalar las dependencias

COPY requirements.txt requirements.txt

RUN pip install -r requirements.txt

# Copiar el resto de los archivos de la aplicación a la imagen

COPY . .

# Comando para ejecutar la aplicación

CMD ["python", "main.py"]

Analicémoslo.

# Usar una imagen base de Python

FROM python:3.11.2

Esto indica que la aplicación corre sobre python 3.11.2, y que esa versión de python se incluirá en la imagen.

# Establecer el directorio de trabajo en el contenedor

WORKDIR /app

Se crea un WORKDIR para la aplicación dockerizada. Se usa el nombre /app por convencionalismo. No tiene por qué coincidir con el nombre del directorio donde está la aplicación.

# Copiar el archivo requirements.txt e instalar las dependencias

COPY requirements.txt requirements.txt

RUN pip install -r requirements.txt

Este fragmento de código hace lo siguiente:

1. Copiar el archivo requirements.txt: La línea COPY requirements.txt requirements.txt indica que se copiará el archivo requirements.txt desde el directorio local (fuera del contenedor) al directorio de trabajo /app dentro del contenedor. Esto permite que Docker acceda al archivo y lo utilice para instalar las dependencias.
2. Instalar las dependencias: La línea RUN pip install -r requirements.txt ejecuta el comando pip install dentro del contenedor y utiliza el archivo requirements.txt para instalar todas las dependencias, librerías, modelos y paquetes que se enumeran en ese archivo. Esta es una práctica común en Python para gestionar las dependencias de una aplicación.

Dado que, como buena práctica, nuestra aplicación se ha creado y desarrollado en un entorno virtual donde hemos instalado todas las dependencias, deberemos haber creado el archivo requirements.txt con pip freeze > requirements.txt.

# Copiar el resto de los archivos de la aplicación a la imagen

COPY . .

Esta línea lo que hace es copiar todos los archivos y directorios de la aplicación en la imagen Docker, con la salvedad de los que indiquemos expresamente que no se copien, como veremos un poco más adelante.

# Comando para ejecutar la aplicación

CMD ["python", "main.py"]

Esta línea se usará para que un contenedor pueda lanzar, en primer lugar, Python y, después, el archivo principal de la aplicación, que suele llamarse así: main.py.

Supongamos que la aplicación requiere parámetros para su funcionamiento. La línea CMD la expresaremos así:

# Comando para ejecutar la aplicación

CMD ["python", "main.py", "param1\_value", "param2\_value", "param3\_value"]

La aplicación deberá incluir la librería argparse, con import argparse, y crear los parámetros.

### SI LA APLICACIÓN ES WEB, STREAMLIT O FAUST.

Si la aplicación hace uso de puertos de Apache, esto también deberemos especificarlo en el Dockerfile, añadiendo cierto código entre las líneas

COPY . .

y

CMD ["python", "main.py"]

En concreto, pensemos en que nuestra aplicación es HTTP. Como sabemos, estas peticiones se envían por el puerto 80. En nuestro Dockerfile deberemos poner:

# Exponer el puerto 80

EXPOSE 80

y

# Activar el puerto de Apache en remoto, y un puerto en nuestro host

docker run -p 8080:80 mi\_imagen

En primer lugar, le decimos a Dockerfile que vamos a abrir el puerto 80 de Apache, con EXPOSE 80. Después le decimos que vamos a mapear el puerto 80 en remoto al puesto 8080 de nuestro host, con docker run -p 8080:80 mi\_imagen, donde mi\_imagen deberá ser el nombre que hayamos decidido darle a la imagen cuando la creemos más adelante. Cuando usamos comunicaciones por http siempre deberemos usar esta sintaxis.

* Si nuestra aplicación en lugar de correr por http, corre por https, estas líneas deberemos cambiarlas así:

# Exponer el puerto 443

EXPOSE 443

y

# Activar el puerto de Apache en remoto, y un puerto en nuestro host

docker run -p 8443:443 mi\_imagen

* Si nuestra aplicación es Streamlit, esas líneas deberemos cambiarlas por:

# Exponer el puerto 8501

EXPOSE 8501

y

# Activar el puerto de Apache en remoto, y un puerto en nuestro host

docker run -p 8502:8501 mi\_imagen

Además, deberemos cambiar la línea CMD por:

CMD ["streamlit", "run", "main.py"]

- Si nuestra aplicación fuera faust, las líneas de puertos deberían quedar así:  
  
# Exponer el puerto 6066

EXPOSE 8501

y

# Activar el puerto de Apache en remoto, y un puerto en nuestro host

docker run -p 6067:6066 mi\_imagen

Y la línea CMD debería quedar así:

CMD ["faust", "-A", "main", "worker"]

## IGNORANDO FICHEROS.

Es muy posible (ocurre muy a menudo) que hay determinados ficheros de la aplicación que no queremos que se incluyan en la imagen cuando la creemos. Tales pueden ser, por ejemplo, los directorios \_\_pycache\_\_, o los archivos reales de configuración con los datos de conexión, credenciales, etc.

Para solucionar esto, en el directorio de nuestra aplicación crearemos un archivo llamado .dockeringnore, donde se anotarán todos los archivos que no queramos incluir en nuestra imagen. Un ejemplo de .dockerignore, a tenor de lo que hemos dicho, podría ser el siguiente:

\*\*/\_\_pycache\_\_

La notación \*\* indica que la coincidencia debe ser recursiva en todos los subdirectorios. Esto asegurará que se excluyan todos los directorios \_\_pycache\_\_ sin importar su profundidad en la estructura de directorios de tu aplicación.

Vamos a suponer que tenemos un directorio config con los archivos de configuración, con los datos reales de autenticación, user.py y connect.py. También tenemos unos archivos llamados example\_user.py y example\_connect.py, donde están las muestras de esos datos, para que se sepa como deben incluirse, pero no están los datos reales. A los usuarios que descarguen nuestra imagen deberemos pasarle los achivos example\_\*.py, pero no los que tienen los datos reales de configuración. Cada usuario deberá poder poner sus propios datos de configuración, sin tener acceso a los nuestros. En el archivo .dockerignore pondremos lo siguiente:

# Excluir todos los archivos en /config que no comienzan con "example\_" y tienen extensión .py

/config/\*.py

!/config/example\_\*.py

## CONSTRUIR LA IMAGEN

Este es el siguiente paso en la dockerización de nuestro proyecto. Lo haremos con el comando:

docker build -t nombre\_de\_la\_imagen .

El nombre de la imagen lo elegiremos a nuestro criterio. No tiene por qué coincidir con el nombre del directorio ni con ninguna otra cosa.

En este momento se puede usar el siguiente comando:

docker images

para listar las imágenes que tenemos disponibles. Debería aparecer la imagen que acabamos de crear.

## EJECUTAR UN CONTENEDOR

Una vez que tenemos la imagen vamos a ejecutar un contenedor para comprobar que puede crearse y ejecutarse sin problemas. Lo haremos con el comando siguiente:

docker run -p 8080:80 nombre\_de\_la\_imagen

La parte -p 8080:80 mapea el puerto 80 del contenedor al puerto 8080 de nuestro sistema host. Los puertos deberán coincidir con los que especificamos en nuestro Dockerfile.

Aquí podemos usar el comando

docker ps

para verificar que el contenedor se ha creado.

Para acceder a la aplicación abriremos el navegador y teclearemos http://localhost:8080.

Para crear y probar un contenedor si nuestra aplicación no usa la web y no necesita un puerto específico, lo crearemos así:

docker run -it nombre\_de\_la\_imagen

La marca -it hace que se abra la consola del contenedor. En ella teclearemos python main.py (suponiendo que así se llama el script de inicio de nuestra aplicación) y deberá ejecutarse desde el contenedor sin problemas.

Una vez probada la aplicación podemos salir de la sesión interactiva con el contenedor usando la combinación de teclas Ctrl - D, o el comando exit.

## PUBLICAR LA IMAGEN EN INTERNET

Para publicar la imagen deberemos hacerlo en dockerhub ([https://hub.docker.com](https://hub.docker.com/)). Deberemos crear una cuenta (es gratis) para poder utilizar el servicio.

Con la cuenta creada deberemos, desde nuestra terminal, identificarnos en Dockerhub, con el comando:  
  
docker login

Si no se conecta directamente con nuestra cuenta en Dockerhub, se nos pedirán las credenciales (usuario y contraseña).

Una vez creada la cuenta, deberemos etiquetar la imagen. Lo haremos con el siguiente comando:

docker tag nombre\_de\_la\_imagen usuario/nombre\_etiqueta

donde:

* nombre\_de\_la\_imagen es el nombre con el que hemos creado la imagen.
* usuario es el nombre de usuario con el que abriste la cuenta en Dockerhub.
* nombre\_etiqueta es un nombre que le daremos a la imagen en Dockerhub.

La cuestión aquí es que la combinación usuario/nombre\_etiqueta debe ser única en Dockerhub.

A continuación, publicaremos la imagen en Dockerhub con el siguiente comando.

docker push usuario/nombre\_etiqueta

Podremos comprobar que la imagen está subida a Dockerhub con el comando siguiente:

docker search tu\_usuario\_en\_dockerhub/nombre\_etiqueta

También se puede comprobar desde la página de Dockerhub, donde se pueden listar las imágenes subidas por el usuario logueado.

## ELIMINAR UNA IMAGEN

Podemos eliminar una imagen de nuestro sistema local con el comando:

docker rmi nombre\_de\_la\_imagen

Una vez eliminada en nuestro sistema local, podemos, si lo deseamos, eliminarla en Dockerhub, así:

docker rmi tu\_usuario\_en\_dockerhub/nombre\_etiqueta

Esta acción es permanente, y no se puede deshacer. Para eliminarla de Dockerhub es preciso estar identificados en nuestra cuenta y haberla eliminado previamente de nuestro sistema local.

## TRANSMITIR LA IMAGEN A OTRO USUARIO

Para transmitir la imagen a otro usuario deberemos pasarle los siguientes datos:

nuestro\_usuario\_en\_dockerhub/nombre\_etiqueta

El usuario destino deberá tener en su ordenador la herramienta Docker Desktop instalada y funcionando. Además, deberá tener una cuenta en Dockerhub. Deberá ingresar a su cuenta en Dockerhub con el comando:

docker login

A continuación, tecleará en su terminal:

docker pull usuario/etiqueta

donde usuario es el nombre del usuario que creó y subió la imagen, y etiqueta es la que el usuario creador empleó para subir la imagen a Dockerhub.

Una vez descargada la imagen, el usuario destino deberá teclear en su terminal el siguiente comando:

docker run -it --name contenedor usuario/nombre\_etiqueta

Donde:

contenedor es el nombre que le quiera dar a su contenedor.

usuario es el nombre del usuario autor de la imagen.

nombre\_etiqueta es la que le puso el autor a la imagen cuando la subió.

Si la aplicación dockerizada es de tipo web, el usuario creador deberá pasarle al usuario destino los dos puertos (el de la aplicación, y el que se mapea en el ordenador local). Entonces el contenedor se creará y ejecutará así:

docker run -p 8080:80 -d --name contenedor usuario\_creador/ etiqueta

Si tu aplicación web requiere configuración a través de variables de entorno, asegúrate de pasar esas variables al contenedor cuando lo crees. Puedes hacerlo utilizando la opción -e al crear el contenedor. Por ejemplo:

docker run -e VARIABLE\_DE\_ENTORNO=valor -d --name contenedor usuario\_creador/nombre\_etiqueta

Toda esta información deberá proporcionársela el usuario creador al usuario destino.

Para detener el contenedor, el usuario destino deberá teclear:

docker stop nombre\_del\_contenedor

y si quiere eliminar el contenedor de su máquina deberá teclear:

docker rm nombre\_del\_contenedor

# DOCKERIZAR UNA APLICACIÓN CON BASE DE DATOS

Entramos en un tema espinoso: Dockerizar una aplicación que hace uso de una base de datos, de forma que podamos transmitir a otros usuarios tanto la aplicación como la base de datos. Vamos a detallar el proceso paso a paso.

## PREPARANDO EL CONTEXTO

Crear la estructura de directorios.

mi\_aplicacion/

|-- app/

| |-- tu\_app.py

| |-- requirements.txt

| |-- Dockerfile\_aplication

| |-- ...

|-- db\_data/

Básicamente lo que hemos hecho es añadir un nivel más de profundidad al árbol de directorios de la aplicación. Hemos creado un subdirectorio llamado app/ donde tenemos todo lo relativo a la aplicación: el archivo main.py, los subdirectorios de clases, el entorno virtual… Todo. También hemos preparado un directorio **db\_data** que, por ahora, queda vacío, pero que servirá para almacenar la base de datos.

## EL DOCKERFILE

Crearemos el Dockerfile, con el nombre Dockerfile\_aplication. Este fichero no necesariamente tiene que llamarse Dockerfile. Basta con que el nombre empiece por Dockerfile. Este archivo debe de estar dentro del directorio /app.

El contenido de Dockerfile\_aplication es el siguiente:

# Usar una imagen base de Python

FROM python:3.11

# Establecer el directorio de trabajo en el contenedor

WORKDIR /container\_dir

# Copiar el archivo requirements.txt e instalar las dependencias

COPY requirements.txt /container\_dir/

RUN pip install -r /container\_dir/requirements.txt

# Comando para ejecutar la aplicación

CMD ["python", "main.py"]

Vamos a comentar algunos puntos esenciales. En primer lugar, encontramos:

# Usar una imagen base de Python

FROM python:3.11

Esto ya lo conocemos. Le indicamos al fichero que la aplicación va a correr sobre python 3.11.

La siguiente línea también la conocemos:

# Establecer el directorio de trabajo en el contenedor

WORKDIR /container\_dir

En esta línea establecemos el directorio de trabajo del contenedor. Esta línea no se modifica por el hecho de tener más profundidad en el árbol de directorios. Lo único que hemos renombrado el directorio de trabajo para ver esto más claro.

A continuación, encontramos:

# Copiar el archivo requirements.txt e instalar las dependencias

COPY requirements.txt /container\_dir/

RUN pip install -r /container\_dir/requirements.txt

Le decimos al Dockerfile\_aplication que el contenido del directorio /app (donde se encuentra toda la aplicación) lo copiaremos al directorio de trabajo del contenedor. Aquí sí se ha tenido en cuenta la nueva estructura de subdirectorios. Como se ve, esto también afecta a la línea que indica que se deben instalar las dependencias de requirements.txt.

El comando final como vemos, no cambia:

# Comando para ejecutar la aplicación

CMD ["python", "tu\_app.py"]

## EL FICHERO docker-compose.yml

Sí, este es nuevo. Se trata de un “orquestador” para compaginar el Dockerfile\_aplication con la base de datos. La mayoría de los motores de base de datos tienen una imagen oficial en Dockerhub. Vamos a suponer que nuestra aplicación emplea MySQL. Se utilizará la imagen oficial de MySQL en Dockerhub. La orquestación se define en un fichero llamado docker-compose.yml, como el que vemos en la siguiente página:



**Sigue leyendo**

version: '3'

services:

aplicacion:

build:

context: ./app

dockerfile: ./app/Dockerfile\_aplication

ports:

- "8000:80" # Ajusta el puerto según sea necesario

depends\_on:

- base\_datos

base\_datos:

image: mysql:latest # Usa la imagen oficial de MySQL o tu imagen personalizada

environment:

MYSQL\_ROOT\_PASSWORD: mi\_contraseña\_root

MYSQL\_DATABASE: mi\_base\_de\_datos

MYSQL\_USER: mi\_usuario

MYSQL\_PASSWORD: mi\_contraseña

volumes:

- db\_data:/var/lib/mysql

Este fichero se almacenará en el directorio principal del proyecto, al mismo nivel que el directorio de la aplicación, como se aprecia en el siguiente esquema:

mi\_aplicacion/

│

├── app/

│ ├── Dockerfile\_aplicacion

│ └── ... (archivos de tu aplicación)

│

├── db\_data/

│

└── docker-compose.yml

Vamos a analizar el fichero docker-compose.yml. En primer lugar encontramos la siguiente línea:

version: '3'

Esto indica que se trata de la versión 3 de la sintaxis de Docker Compose, que es la actualmente en vigor. Cuando tengas que construir tu propio fichero docker-compose.yml deberás consultar la versión actual en la página oficial (<https://docs.docker.com/compose/>).

Lo siguiente que encontramos es el bloque services que, a su vez, está dividido en un bloque para la aplicación y la base de datos, en una estructura así:

services:

aplicacion:

# Datos de la dockerización de la aplicación

base\_datos:

# Datos del servicio de la base de datos

### EL BLOQUE aplicacion

Dentro de aplicacion encontramos lo siguiente:

build:

context: ./app

dockerfile: ./app/Dockerfile\_aplication

El bloque build dentro del servicio de la aplicación (aplicacion) en el archivo docker-compose.yml se utiliza para definir cómo se construirá la imagen del contenedor de la aplicación. Aquí está la explicación de los campos:

**context:** Especifica el contexto en el que buscar los archivos necesarios para construir la imagen. En este caso, ./app se refiere al directorio app de nuestro árbol de directorios, lo que significa que Docker buscará los archivos relacionados con la construcción de la imagen en el directorio donde está Dockerfile\_aplication, que se utilizará para construir la imagen de la aplicación.

**dockerfile:** Este campo permite especificar el nombre del Dockerfile a utilizar. En este caso, se establece como Dockerfile\_aplication, lo que significa que Docker buscará este archivo en el contexto definido por context. El Dockerfile contiene las instrucciones para construir la imagen del contenedor de la aplicación.

En resumen, estas líneas indican a Docker que use el archivo Dockerfile\_aplication ubicado en el contexto /app para construir la imagen del contenedor de la aplicación. El Dockerfile\_aplication ya contiene las instrucciones específicas para crear esta imagen, como la instalación de dependencias, configuración, etc.

A continuación encontramos:

ports:

- "8000:80" # Ajusta el puerto según sea necesario

Esto se indica si nuestra aplicación hace uso de puertos HTTP, cómo sería el caso de una aplicación web. En el ejemplo se supone que la aplicación corre sobre el puerto 80, que se mapea al puerto 8080 de la máquina host. Estas indicaciones también estarían, en su caso, en el fichero Dockerfile\_aplication. En caso de que no sea una aplicación web estas líneas no se incluyen.

Lo siguiente que encontramos es:

depends\_on:

- base\_datos

El bloque depends\_on en el servicio de la aplicación en el archivo docker-compose.yml establece una dependencia entre la aplicación y la base de datos. Esto significa que el servicio de la aplicación se iniciará una vez que el servicio de la base de datos esté en funcionamiento.

En el ejemplo, se indica que el servicio de la aplicación depende del servicio de la base de datos llamado base\_datos. Esto es importante en escenarios donde la aplicación necesita que la base de datos esté disponible antes de iniciar su propio funcionamiento, ya que necesita conectarse a la base de datos.

Establecer esta dependencia ayuda a asegurarse de que los servicios se ejecuten en el orden correcto y que la aplicación no intente conectarse a la base de datos antes de que esta esté lista.

Es una práctica común en aplicaciones web o en cualquier escenario donde la aplicación dependa de servicios externos, como bases de datos o servicios de cola.

El término “servicios de cola” se refiere a sistemas de mensajería y procesamiento asíncrono que se utilizan comúnmente en aplicaciones para gestionar tareas y comunicación entre componentes. Estos servicios permiten la distribución de tareas encoladas y la comunicación entre diferentes partes de una aplicación o entre aplicaciones diferentes. Algunos ejemplos de servicios de cola populares son RabbitMQ, Apache Kafka y Redis, entre otros.

Los servicios de cola son útiles en situaciones en las que es necesario procesar tareas en segundo plano, garantizar la entrega de mensajes, administrar eventos y lograr una comunicación asíncrona entre componentes de la aplicación. Pueden ser especialmente útiles en aplicaciones que requieren escalabilidad, alta disponibilidad y procesamiento en tiempo real.

Al usar Docker y Docker Compose en combinación con servicios de cola, puedes definir, orquestar y escalar contenedores que manejen estas tareas y comunicación de manera eficiente. Docker Compose permite administrar fácilmente la interacción entre diferentes componentes de una aplicación, incluidos los servicios de cola, como parte de un entorno de desarrollo y pruebas.

### EL BLOQUE base\_datos

Dentro de este bloque lo primero que encontramos es lo siguiente:

image: mysql:latest # Usa la imagen oficial de MySQL o tu imagen personalizada

Especifica la imagen de Docker que se utilizará para crear el contenedor de la base de datos MySQL.

**mysql:latest**: En este caso, se está utilizando la imagen oficial de MySQL que está disponible en Docker Hub. El latest indica que se usará la versión más reciente de la imagen oficial de MySQL. Docker Hub es un registro de imágenes de Docker donde se pueden encontrar imágenes oficiales y de la comunidad.

A continuación encontramos las credenciales de la base de datos en el siguiente bloque:

environment:

MYSQL\_ROOT\_PASSWORD: mi\_contraseña\_root

MYSQL\_DATABASE: mi\_base\_de\_datos

MYSQL\_USER: mi\_usuario

MYSQL\_PASSWORD: mi\_contraseña

Las credenciales y nombre de la base de datos que se especifican en el bloque environment son las que tiene el constructor de la aplicación y de la imagen. Luego el destinatario que se descargue la imagen deberá poner el mismo nombre de base de datos, pero con sus propias credenciales. Incluso podrá cambiar el nombre de la base de datos, si lo cambia también en la aplicación.

En resumen, el desarrollador proporciona configuraciones predeterminadas, pero los usuarios finales pueden ajustarlas según sus requisitos específicos al crear y ejecutar sus propios contenedores a partir de la imagen y el archivo de Docker Compose. Por ejemplo. La mayoría de nosotros, en nuestra máquina local, tenemos en MySQL un usuario root sin contraseña. El bloque quedaría así:

environment:

MYSQL\_ROOT\_PASSWORD:

MYSQL\_DATABASE: gente

MYSQL\_USER: root

MYSQL\_PASSWORD:

Por supuesto, en una aplicación real en producción deberían existir unas contraseñas seguras.

La última parte de este bloque es:

volumes:

- db\_data:/var/lib/mysql

La sección volumes en el bloque del servicio de la base de datos en el archivo docker-compose.yml se utiliza para definir un volumen de Docker. En este caso, se está especificando un volumen llamado db\_data y se está asignando a la ruta /var/lib/mysql dentro del contenedor MySQL. Esto se hace con el formato nombre\_volumen:ruta\_dentro\_del\_contenedor.

La función de este volumen es permitir que los datos de la base de datos en el contenedor persistan entre reinicios del contenedor o entre la creación y eliminación de contenedores. En lugar de almacenar los datos directamente en el sistema de archivos del contenedor, se utilizan volúmenes para mantenerlos fuera del contenedor y facilitar la administración y la persistencia de datos.

La ruta /var/lib/mysql en docker-compose.yml se refiere a una ubicación dentro del contenedor MySQL. No se trata de una ruta en la máquina local del desarrollador ni en la del destinatario. Es una ruta interna del contenedor, que es donde MySQL almacena sus datos de base de datos de forma predeterminada. Se creará en el contenedor en el momento de ponerlo en marcha.

## CREAR LAS IMÁGENES

Ahora llega el momento de crear las imágenes. Cuando dockerizamos un proyecto así, con un docker compose donde se refiere a una aplicación y una base de datos, hay que generar dos imágenes, no una: una imagen para la aplicación y otra para la base de datos. Lo hacemos con la siguiente instrucción:

docker-compose build

Con esto se crean las imágenes a partir del fichero docker-compose.yml. La sintaxis para crear estas imágenes es más simple que cuando no teníamos un orquestador, ya que, ahora, no tenemos que especificar opciones, ni nombrar las imágenes. Docker Compose se ocupa de eso por nosotros.

Ahora podemos ver las dos imágenes con el comando

docker images

Un ejemplo teórico de cómo podría salir el resultado sería el siguiente:

REPOSITORY TAG IMAGE ID CREATED SIZE

aplicacion latest abcdef123456 2 hours ago 300MB

base\_datos latest 7890ghijklmn 2 hours ago 500MB

Esto es sólo un ejemplo teórico, los nombre e identificadores variarán en cada caso.

## LOS CONTENEDORES

Ahora, para ver que todo funciona en nuestra máquina local vamos a lanzar los contenedores de ambas imágenes. En esta ocasión lo hacemos con:

docker-compose.up

Ahora podemos comprobar que se han lanzado los contenedores con:

docker ps

Una salida típica sería la siguiente:

CONTAINER ID IMAGE COMMAND 17f4b29c12ed mi\_aplicacion:latest “python main.py” 22efbb0ceab2 mysql:latest “docker-entrypoint.s…” 2CREATED STATUS PORTS

1 minute ago Up 1 minute 0.0.0.0:8000->8000/tcp

minutes ago Up 2 minutes 3306/tcp

En esta salida:

* **CONTAINER ID** es el identificador único del contenedor.
* **IMAGE** muestra la imagen utilizada para crear el contenedor.
* **COMMAND** muestra el comando que se ejecuta dentro del contenedor.
* **CREATED** indica cuándo se creó el contenedor.
* **STATUS** muestra el estado actual del contenedor (en este caso, Up significa que el contenedor está en ejecución).
* **PORTS** muestra la asignación de puertos entre el host y el contenedor.

Podemos detener todos los contenedores en ejecución con:

docker-compose down

## SUBIR LAS IMÁGENES A LA NUBE

Llegados a este punto asumiremos que ya tenemos una cuenta activa en Dockerhub. Nos identificaremos en la consola con:

docker login

Una cosa importante es tener un repositorio para estas imágenes. En realidad, es una buena práctica tener un repositorio para cada proyecto, de la misma forma que lo hacemos en Git Hub. Esto nos proporcionará un mayor orden y control sobre nuestras imágenes.

Podemos subir las imágenes simplemente con:

docker-compose push

Sin embargo, si queremos tener un mayor control deberemos incluir una modificación en el fichero docker-compose.yml, indicando el nombre de usuario, el nombre del repositorio y la etiqueta de cada imagen, así:



**¿Terminarás lo  
que has empezado?**

version: '3'

services:

aplicacion:

build:

context: ./app

dockerfile: ./app/Dockerfile\_aplication

image: usuario/repositorio/mi\_aplicacion:latest

ports:

- "8000:80" # Ajusta el puerto según sea necesario

depends\_on:

- base\_datos

base\_datos:

image: usuario/repositorio/mysql:latest # Usa la imagen oficial de MySQL o tu imagen personalizada

environment:

MYSQL\_ROOT\_PASSWORD: mi\_contraseña\_root

MYSQL\_DATABASE: mi\_base\_de\_datos

MYSQL\_USER: mi\_usuario

MYSQL\_PASSWORD: mi\_contraseña

volumes:

- db\_data:/var/lib/mysql

Donde encontramos que:

* **usuario** es el nombre de usuario en Dockerhub.
* **repositorio** es el nombre del repositorio en Dockerhub
* **mi\_aplicacion** es un alias que usaremos para la imagen de la aplicación.
* **latest** es una etiqueta (tag) para la imagen.

Por supuesto, estos cambios deberemos incluirlos en docker-compose.yml ANTES de crear las imágenes.

## DESCARGAR Y USAR LAS IMÁGENES

La persona destinataria de estas imágenes, para poder descargárselas, deberá, en primer lugar, tener en su máquina Docker Desktop y, por supuesto, una cuenta en Dockerhub, como en el caso anterior.

A continuación, deberá entrar en el directorio que vaya a usar para trabajar con nuestros contenedores y teclear:

docker pull usuario/repositorio/mi\_aplicacion:latest

docker pull usuario/repositorio/mi\_base\_de\_datos:latest

Donde tendrá que tener en cuenta que:

* **usuario** es el nombre del usuario creador de las imágenes.
* **repositorio** es el nombre del repositorio donde están las imágenes.
* **mi\_aplicacion** y **mi\_base\_de\_datos** son los nombres de las imágenes.

## LO QUE EL DESTINATARIO DEBE RECIBIR

En este caso, para poder crear y ejecutar los contenedores, el usuario creador de la aplicación debe compartir con el destinatario varias cosas:

* El archivo docker-compose.yml, omitiendo las credenciales de su base de datos.
* Etiquetas de las imágenes.
* El nombre de usuario en Dockerhub del creador.
* En su caso, los puertos de la aplicación y de mapeo en la máquina host, si la aplicación los usa. Forman parte, en ese caso, del docker-compose.yml.

El destinatario deberá grabar el docker-compose.yml en su directorio de trabajo, en su propia máquina, y colocar las credenciales de su base de datos.

## EJECUTAR LOS CONTENEDORES

Una vez el usuario destino tenga en su máquina local las imágenes podrá ejecutar los contenedores. Hay que tener en cuenta que los contenedores deben ejecutarse en un orden específico atendiendo a las dependencias. Por ejemplo, en nuestro caso la aplicación necesita la base de datos para trabajar, y así está especificado en el docker-compose.yml donde, en el bloque del servicio de la aplicación, tenemos:

depends\_on:

- base\_datos

Por lo tanto, en primer lugar, deberá poner en marcha el contenedor de la base de datos, y luego, cuando este ya esté corriendo, el de la aplicación.

Para poner en marcha el contendor de la base de datos el usuario destino deberá teclear algo similar a lo siguiente:

docker-compose up base\_datos

Dónde:

**base\_datos** es el nombre de la base de datos en docker-compose.yml.

Para comprobar que el contenedor se haya puesto en marcha correctamente, el usuario destino puede teclear:

docker ps

Esto le mostrará la lista de contenedores que tienen en ejecución en su máquina.

Si quiere comprobar específicamente si está funcionando el contenedor que acaba de lanzar, puede teclear:

docker inspect nombre\_del\_contenedor

Para arrancar el contenedor de la aplicación, el destinatario deberá teclear:

docker-compose up aplicación

Donde:

**aplicacion** es el nombre del servicio de la aplicación en docker-compose.yml

## PONER EN MARCHA LA APLICACIÓN

Si la aplicación es de tipo web, el usuario destino puede ponerla en marcha desde su propio navegador, usando los puertos que le haya pasado el creador de la aplicación. Por ejemplo, si la aplicación debe correr sobre el puerto 80, mapeando al puerto 8080 de la máquina host, en el navegador se llamará como <http://localhost:8080>.

Si la aplicación es de consola, el usuario destino debe teclear en su terminal:

docker exec -it mi\_app\_contenedor bash

Donde mi\_app\_contenedor es el nombre del contenedor de la aplicación.

## PERSISTIENDO LA BASE DE DATOS

Supongamos que el usuario destino desea persistir la base de datos en una ruta concreta de su máquina local. En ese caso, antes de poner en marcha los contenedores, deberá hacer una modificación en el archivo docker-compose.yml. En concreto, en la línea:

volumes:

- db\_data:/var/lib/mysql

deberá poner:

volumes:

- ./ruta/local/a/la/base/de/datos:/var/lib/mysql

# WSL Y WSL2

Todo lo que hemos comentado es válido para usuarios de Linux y de MacOS. Los usuarios de Windows deberán instalar en su máquina un emulador de Linux que se llama WSL o, en su versión más moderna, WSL 2.

Para ello deberán ir a la página de [Microsoft Store](https://learn.microsoft.com/es-es/windows/wsl/install) y seguir los pasos que allí se indican.

# GLOSARIO DE TÉRMINOS

* **Contenedor:** Un entorno aislado que incluye una aplicación y todas sus dependencias, permitiendo la ejecución de aplicaciones de manera consistente en diferentes entornos.
* **Docker Compose:** Una herramienta que permite definir y administrar aplicaciones multi-contenedor utilizando un archivo YAML.
* **Docker Engine:** El núcleo de Docker que proporciona la funcionalidad fundamental para la creación y administración de contenedores.
* **Docker Hub:** Un repositorio en línea que almacena imágenes Docker públicas y privadas, lo que facilita la distribución y el acceso a imágenes Docker predefinidas.
* **Docker Swarm:** Una característica de Docker que permite crear y administrar un clúster de contenedores para alta disponibilidad y escalabilidad.
* **Dockerfile:** Un archivo de texto que contiene instrucciones para construir una imagen Docker personalizada.
* **Dockerizar:** El proceso de crear una imagen Docker y ejecutar contenedores a partir de ella.
* Imagen: Un paquete ligero y autónomo que incluye todo lo necesario para ejecutar una pieza de software, incluyendo código, un entorno de ejecución y dependencias.
* **Orquestador:** Una herramienta que facilita la administración, el escalado y la orquestación de contenedores en un entorno de clúster.
* **Orquestar:** El acto de administrar y coordinar múltiples contenedores para garantizar su correcto funcionamiento y escalado.
* **Volumen:** Un mecanismo de Docker que permite persistir datos fuera de los contenedores y compartir datos entre contenedores. Los volúmenes son útiles para almacenar datos que deben sobrevivir al ciclo de vida de un contenedor.



**No subestimes el poder del lado oscuro de la Fuerza**