Docker y la Nube: Guía

Flujo de trabajo

1 FASE 1: Construcción (Tu PC)

1. Escribir la Función/Aplicación

Tu aplicación es un conjunto de piezas que viven en tu máquina, pero que dependen de ese entorno específico.

Código Fuente (main.py)

Dependencias (requirements.txt)

Tu Sistema Operativo (Windows/Mac)

Resultado: ¡Funciona solo en tu máquina!

2. Dockerizar (Crear la Imagen)

El `Dockerfile` es la **receta inmutable**. Al ejecutar el `build`, Docker sigue la receta paso a paso para crear la **Imagen**.

Paso 2A: La Receta (`Dockerfile`)

```
FROM python:3.11-slim # Base SO/Python
WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt # Instalar deps
COPY . .
CMD ["uvicorn", "main:app", "--host", "0.0.0.0"]
```

Paso 2B: El "Horneado" (Comando)

```
docker build -t mi-monitor-app:v1.0 .
```

- `-t mi-monitor-app:v1.0`: Crea la **etiqueta** (nombre y versión).
- `.`: Le dice dónde buscar el `Dockerfile`.

Resultado: Tienes la IMAGEN guardada localmente en tu Docker Engine. ¡Lista para viajar!

¿Dónde queda la Imagen de Docker?

La Imagen de Docker **no es un archivo simple con extensión (.zip, .exe)** que puedas ver en el explorador de Windows o Finder. Es un concepto más profundo:

1. La Imagen NO es un Archivo (es un conjunto de Capas)

- Se almacena en el **Motor de Docker (Docker Engine)** de tu máquina.
- Es como una colección de capas de cebolla (el SO base, las librerías de Python, tu código, etc.) guardadas en una base de datos interna de Docker.

2. ¿Cómo me aseguro que existe? ¡Usando el comando!

Esta es la forma **correcta** de interactuar con tus imágenes locales:

```
docker images # Muestra todas las imágenes construidas y descargadas
```

Ejemplo de Salida:

```
REPOSITORY        TAG       IMAGE ID      SIZE
mi-monitor-app    v1.0      a1b2c3d4e5f6  400MB (¡Aquí está tu Imagen!)
```

```
python          3.11-slim latest      120MB
```

2 FASE 2: Empaque y Registro (Nube)

3. El Resultado: La Imagen Lista para la Nube

La Imagen es el objeto mágico: es la caja sellada, inmutable, que garantiza la consistencia del entorno en cualquier parte del mundo.

Paso 3A: Nombrar para el Registro (Tag)

Antes de subirla, debemos ponerle el "sticker" de la dirección de la nevera (el Registro de Contenedores):

```
docker tag mi-monitor-app:v1.0 acrname.azurecr.io/mi-monitor-app:v1.0
```

Paso 3B: Subir la Imagen (Push)

Este comando es el que transfiere la Imagen de tu PC al servicio de la nube (ACR o ECR).

```
docker push acrname.azurecr.io/mi-monitor-app:v1.0
```

Registro de Contenedores (La Nevera Global)

- **Azure:** Azure Container Registry (ACR)
- **AWS:** Amazon Elastic Container Registry (ECR)
- **General:** Docker Hub

Resultado: La Nube ya puede ver y descargar tu aplicación sin tener tu código fuente.

3 FASE 3: Despliegue (Producción)

4. Implementación en la Nube

¡LA NUBE NUNCA CLONA EL CÓDIGO!

Usas la consola de AWS o Azure y les indicas qué Imagen quieres que ejecuten.

Mecanismo: PULL (Descarga)

- El servicio de la nube (ECS, EKS, Container Apps) se conecta al Registro (ACR/ECR).
- Realiza un `**docker pull**` de la Imagen que especificaste.
- La Imagen se ejecuta, creando un **CONTENEDOR**.
- El Contenedor corre tu aplicación exactamente como si estuviera en tu PC.

El Contenedor (La Instancia Viva)

La Imagen es el molde estático. El **Contenedor** es la Imagen cobrando vida. A este contenedor se le asigna la IP pública y un dominio para que los usuarios puedan acceder a tu API.

Tu aplicación es ahora un microservicio corriendo en un servidor global.

Flujo Completo: Código → Dockerfile → IMAGEN (local) → IMAGEN (Registro) → CONTENEDOR (Nube)

FASE 4: Iteración Rápida y

Optimización

El Súper Poder de Docker: Caché de Capas

Si te equivocas o quieres mejorar el `Dockerfile`, no tienes que reconstruir todo. Docker es inteligente y aprovecha el **caché de capas**.

Mecanismo: `docker build` y el Caché

- **Capa Inmutable:** Cada línea del `Dockerfile` se convierte en una **capa** almacenada en tu Docker Engine.
- **Acierto (Cache Hit):** Si ejecutas `docker build` de nuevo y la instrucción (y sus archivos de entrada) no ha cambiado, Docker usa la capa ya guardada.
- **Fallo (Cache Miss):** Tan pronto como una línea cambia, Docker la vuelve a ejecutar y **desecha el caché de todas las líneas que le siguen**.

Flujo de Corrección: ¡Solo Vuelve a Construir!

El proceso es el mismo, pero mucho más rápido gracias al caché:

```
# 1. Edita el Dockerfile
# 2. Ejecuta el mismo comando de construcción:
docker build -t mi-monitor-app:v1.0 .
```

Ejemplo de Ahorro de Tiempo (Solo cambias `main.py`):

Instrucción en Dockerfile	¿Se usa el Caché?
`RUN pip install -r requirements.txt` (Parte lenta)	✅ **SÍ** (A menos que cambies `requirements.txt`)
`COPY . .` (Contiene `main.py` actualizado)	❌ **NO** (Se vuelve a ejecutar y actualiza la capa)

Tip de Optimización: Ordena tus Capas

Coloca las instrucciones que **cambian poco** (como la instalación de librerías) **antes** de las instrucciones que **cambian mucho** (como copiar tu código fuente). De esta manera, Docker maximiza el uso del caché.

La consistencia es el súper poder de Docker. Si funciona en tu máquina, funciona en la nube.