

Dia	Mes	Año	Hora	Institución			
Alumno					Código	Materia	
Curso	Bimestre	Semestre	Salón	Hoja No.	de	CALIFICACIÓN	
Profesor							

## Docker

### Introducción a Docker

#### ¿Qué es Docker?

Docker es una plataforma de código abierto que automatiza el despliegue, escalado y gestión de aplicaciones dentro de contenedores de software. Un contenedor es una unidad estandarizada de software que empaqueta el código y todas sus dependencias (bibliotecas, frameworks, etc) para que la aplicación se ejecute de forma rápida y fiable en cualquier entorno informático.

La idea central es "Construirlo una vez, ejecutarlo en cualquier lugar". Al empaquetar la aplicación y su entorno en una unidad aislada y portátil, Docker resuelve el clásico problema de "funciona en mi máquina, pero no en producción". Los contenedores garantizan la consistencia en los entornos de desarrollo, pruebas y producción.

### Historia y Motivación

Docker fue presentado al público por Solomon Hykes en la Rycón de 2013. Inicialmente, era un proyecto interno de dotCloud, una empresa de Platform as a Service (PaaS). La tecnología subyacente de los contenedores no era nueva (se basaba en primitivas del kernel de Linux como cgroups y namespaces que existen desde hace años), pero Docker la hizo accesible, fácil de usar y la estandarizó.

La motivación principal era simplificar y acelerar los flujos de trabajo de desarrollo y despliegue. Antes de Docker, la configuración de entornos era un proceso manual, propenso a errores y lento. Las máquinas virtuales eran la solución pero resultaban pesadas y lentas de iniciar. Docker nació de la necesidad de una alternativa más ligera, rápida y eficiente.

### Casos de uso en la industria

Docker se ha convertido en un estándar de la industria y es utilizado por industrias, empresas de todos los tamaños, desde startups hasta grandes corporaciones como Google, Netflix y Spotify.

- **Microservicios:** Es el caso de uso más popular. Cada microservicio se puede empaquetar en un contenedor independiente, lo que permite desarrollarlos, desplegarlos y escalonarlos de forma aislada.
- **Integración y despliegue continuo (CI/CD):** Docker permite crear entornos de compilación y pruebas consistentes. Con herramientas como Jenkins o GitHub Actions, se pueden construir imágenes Docker, ejecutar pruebas en contenedores y desplegar automáticamente en producción.
- **Migración a la nube:** Dockerizar una aplicación monolítica es a menudo el primer paso para modernizarla y prepararla para su despliegue en proveedores de la nube como AWS, Azure o Google Cloud.
- **Entorno de desarrollo locales:** Los desarrolladores pueden replicar el entorno de producción exacto en sus máquinas locales, eliminando inconsistencias y facilitando la colaboración.
- **Big Data y Machine Learning:** Docker permite empaquetar aplicaciones complejas con todas sus dependencias (como modelos de ML, bibliotecas de Python, etc.) en un formato portátil, facilitando la reproducibilidad de los experimentos.

## Beneficios frente a Máquinas Virtuales

La principal ventaja de Docker sobre las máquinas virtuales es la eficiencia en el uso de recursos.

- **Menor sobrecarga:** Los contenedores comparten el kernel del sistema operativo anfitrión. No necesitan un sistema operativo completo para cada instancia, lo que los hace mucho más ligeros en términos de CPU, RAM y espacio en disco. Una VM, en cambio, incluye un SO invitado completo.
- **Arranque Rápido:** Los contenedores se inician en segundos, mientras que las VMs pueden tardar varios minutos en arrancar su sistema operativo.
- **Portabilidad:** Una imagen de Docker se puede ejecutar en cualquier sistema que tenga Docker instalado sin modificaciones.
- **Mayor densidad:** Debido a su ligereza, se pueden ejecutar muchos más contenedores que VMs en el mismo hardware. Lo que la hace más eficiente a la hora de ejecutar más herramientas.

# Fundamentos de Docker

## Conceptos clave:

- **Imagen (Image):** Una plantilla de solo lectura con instrucciones para crear un contenedor. Es como una receta que contiene el sistema de archivos de la aplicación, el código y las dependencias. Los imágenes se construyen a partir de un Dockerfile y se organizan en capas.
- **Contenedor (Container):** Una instancia en ejecución de una imagen. Es un entorno aislado y ejecutable. Se pueden crear, iniciar, detener, mover y eliminar contenedores. Es la unidad de software viva.
- **Capa (Layer):** Los imágenes de Docker se construyen a partir de una serie de capas apiladas y de solo lectura. Cada instrucción es un Dockerfile crea una nueva capa. Esto hace que las compilaciones y las transferencias sean muy eficientes, ya que las capas son muy prácticas y se almacenan en caché y se reutilizan.
- **Volumen (Volume):** Es el mecanismo preferido para persistir los datos generados y utilizados por los contenedores. Los volúmenes son generados por Docker y se almacenan en una parte del sistema de archivos del host. Permiten que los datos sobrevivan incluso si el contenedor se elimina.
- **Red (Network):** Docker permite crear redes virtuales para que los contenedores puedan comunicarse entre sí de forma aislada o conectarse a la red del host. Los tipos de red más comunes son bridge (por defecto) host y overlay.

## Explicación de Docker Engine y su arquitectura

Docker Engine es una aplicación cliente-servidor con estos componentes principales:

- **Demonio de Docker (dockerd):** Es un proceso persistente que gestiona los objetos de Docker (imágenes, contenedores, redes, volúmenes). Escucha las solicitudes de la API de Docker y se encarga de todo el trabajo pesado.
- **API REST:** Especifica una interfaz que los programas pueden usar para comunicarse y darle instrucciones

• Cliente Docker (Docker): Es la herramienta de linea de comandos (CLI) que permite a los usuarios interactuar con el demonio de Docker o a traves de la API. Cuando se ejecuta un comando como docker run, el cliente esta solicitando al demonio dockerd.

Flujo basico: Dockerfile → imagen → contenedor

• Dockerfile: El desarrollador escribe un archivo de texto llamado Dockerfile. Este archivo contiene una serie de instrucciones secuenciales sobre como construir la imagen. Por ejemplo:

- FROM python:3.9-alpine (usa una imagen base ligero de Python).
- WORKDIR /app (Establece el directorio de trabajo).
- COPY . . (Copia los archivos del proyecto a la imagen).
- RUN pip install -r requirements.txt (Instala las dependencias).
- CMD ["python", "app.py"] (Define el comando a ejecutar cuando se inicie el contenedor).

• Imagen (Build): Se ejecuta el comando docker build -t mi-app. Docker lee el Dockerfile, ejecuta cada instrucion paso a paso, crea una capa para cada una y las apila para formar la imagen final mi-app.

• Contenedor (Run): Se ejecuta el comando docker run mi-app. Docker Engine toma la imagen mi-app, crea un contenedor a partir de ella y ejecuta el comando definido en CMD.  
La aplicacion ahora esta corriendo dentro de un contenedor aislado.

## Instalacion de Docker

La forma recomendada de (s) instalar Docker es a traves de Docker Desktop, Una aplicacion facil de instalar para Mac, Windows o Linux que incluye Docker engine, el cliente Docker CLI, Docker Compose y mas.

Día	Mes	Año	Hora	Institución			
Alumno					Código	Materia	
Curso	Bimestre	Semestre	Salón	Hoja No.	de	CALIFICACIÓN	
Profesor							

## Trabajando con imágenes

- docker search <nombre> : Busca imágenes públicas en Docker Hub.
  - Ejemplo: docker search nginx
- docker pull <nombre\_imagen>:<tag>: Descarga una imagen desde un registro (por defecto, Docker Hub).
  - Ejemplo: docker pull ubuntu:22.04
- docker images: Lista todas las imágenes que están descargadas localmente.
- docker rmi <id\_imagen\_o\_nombre>: Elimina una o más imágenes locales.
  - Ejemplo: docker rmi ubuntu:22.04

## Crear imágenes propias con Dockerfile

Un Dockerfile es la receta para construir la imagen. Ejemplo con python y Flask:

FROM python:3.9-slim → usa imagen oficial de python

WORKDIR /app → Establece el directorio de trabajo dentro del contenedor

COPY requirements.txt .

RUN pip install --no-cache-dir -r requirements.txt → se copian los archivos de requerimientos y los instala.

COPY .. → copia el resto del código de la aplicación

EXPOSE 5000 → Expone el puerto en el que la aplicación se ejecuta

ENV FLASK\_APP=app.py → Define la variable de entorno para Flask

CMD ["flask", "run", "--host=0.0.0.0"] → comando para ejecutar la aplicación

## Ciclo de vida de un contenedor

### Crear y ejecutar (docker run)

El comando docker run es el más utilizado. Crea un contenedor a partir de una imagen y lo inicia. Combina los comandos docker create y docker start.

docker run [OPCIONES] IMAGEN [COMANDO] [ARG...]

Ejemplo básico:

- docker run ubuntu

- Con mapeo de puertos:

- docker run -p 8080:80 nginx (mapea el puerto 8080 del host al puerto 80 del contenedor)

- Asignando un nombre:

- docker run --name mi-servidor -p 8080:80 nginx

### Modo interactivo VS. detached

- Modo interactivo (-it)

docker run -it ubuntu bash. Conecta la terminal del host a la terminal del contenedor. -i (interactivo) mantiene STDIN abierto y -t asigna un pseudo-TTY. Es útil para depurar o ejecutar comandos dentro del contenedor.

- Modo detached (-d)

docker run -d -p 8080:80 nginx. Ejecuta el contenedor en segundo plano. La terminal queda libre. Este es el modo estándar para ejecutar servicios como servidores web o bases de datos.

### Administración de contenedores

- docker ps → Lista los contenedores en ejecución

- docker ps -a → Lista todos los contenedores (en ejecución y detenidos)

- docker stop <nombre> → Detiene el contenedor en ejecución de forma gradual (enviando una señal SIGTERM).

- docker kill <nombre> → Detiene un contenedor de forma abrupta (enviando una señal SIGKILL).

- docker rm <nombre> → Elimina uno o más contenedores detenidos
  - docker rm -f <nombre> → Fuerza la eliminación de un contenedor en ejecución.
- docker logs <nombre> → Muestra los logs (salida estandar) de un contenedor.
  - docker logs -f <nombre> → Sigue los logs en tiempo real.
- docker exec -it <nombre> <comando> → Ejecuta el comando dentro de un contenedor en ejecución.
  - Ejemplo: docker exec -it mi-servidor bash (abre una shell dentro del contenedor mi-servidor).

## Persistencia con Volumenes

Los datos dentro de un contenedor son efímeros. Si eliminamos el contenedor, los datos se pierden.

Los volumenes solucionan este inconveniente.

- docker volume create mi-volumen
  - Montar un volumen al crear el contenedor:
- ```
docker run -d --name mi-db -v mi-volumen:/var/lib/mysql mysql
```

Aquí mi-volumen el (el) es el nombre gestionado por docker, y /var/lib/mysql es la ruta dentro del contenedor donde se almacenaran los datos de la base de datos. Los datos en esa ruta ahora persistirán en el volumen mi-volumen independientemente del ciclo de vida del contenedor.

## Redes entre contenedores

Por defecto los contenedores se pueden comunicar entre si usando sus direcciones IP internas si están en la misma red (brig) bridge. Sin embargo es mejor practica crear una red personalizada.

### Crear una red

```
- docker network create mi-red
```

Lanzar contenedores en esa red

```
- docker run -d --name mi-db --network mi-red mysql
```

```
- docker run -d --name mi-app --network -p 80:80 mi-imagen
```

Ahora desde el contenedor mi-app, se puede conectar a la base de datos usando el nombre del contenedor como si fuera un hostname.

## Docker Compose

Docker Compose es una herramienta para definir y ejecutar aplicaciones Docker multi-contenedor. Utiliza un archivo de configuración en formato YAML para configurar los servicios, redes y volúmenes de la aplicación.

### Comandos básicos

- docker-compose up: Crea e inicia todos los servicios definidos
- docker-compose down: Detiene y elimina los contenedores
- docker-compose ps: Lista los contenedores de la aplicación
- docker-compose logs: Muestra los logs de los servicios
- docker-compose exec <nombre> <comando>: Ejecuta un comando en un servicio en ejecución

### Ejemplo con WordPress + MySQL

Se crea un archivo docker-compose.yml

#### services:

##### db:

image: mysql:5.7  
container\_name: WordPress\_db

##### volumes:

- db\_data:/var/lib/mysql

restart: always

##### environment:

MYSQL\_ROOT\_PASSWORD: root

MYSQL\_DATABASE: wordpress

MYSQL\_USER: root

MYSQL\_PASSWORD: root

##### wordpress:

image: wordpress:latest

container\_name: WordPress\_app

##### depends\_on:

- db

##### ports:

- "8080:80"

restart: always

##### environment:

WORDPRESS\_DB\_HOST: db:3306

WORDPRESS\_DB\_USER: wordpressuser

WORDPRESS\_DB\_PASSWORD: 12345\_

WORDPRESS\_DB\_NAME: prueba

##### Volumes:

db\_data: