Guía Completa de Docker para Principiantes

Desde Cero hasta Despliegue de Aplicaciones

M.T.I.E. Irving Ulises Hernández Miguel

octubre 2025

Índice

1 Introducción a Docker

1.1 ¿Qué es Docker?

Docker es una plataforma de código abierto que permite desarrollar, implementar y ejecutar aplicaciones dentro de **contenedores**.

¿Qué es un contenedor?

- Es un paquete software que incluye todo lo necesario para ejecutar una aplicación
- Incluye: código, runtime, system tools, system libraries, configuraciones
- Es **aislado** del sistema anfitrión
- Es **portable** - funciona igual en cualquier máquina

Analogía: Contenedores de transporte

- Los contenedores de barco pueden transportar cualquier carga
- El barco (sistema operativo) solo necesita saber manejar contenedores
- No importa qué hay dentro del contenedor
- Se pueden apilar y organizar fácilmente

1.2 ¿Por qué usar Docker?

- **Consistencia**: Funciona igual en desarrollo, testing y producción
- **Aislamiento**: Las aplicaciones no interfieren entre sí
- **Eficiencia**: Menos recursos que las máquinas virtuales
- **Portabilidad**: "Build once, run anywhere"
- **Escalabilidad**: Fácil de replicar y distribuir

1.3 Docker vs Máquinas Virtuales

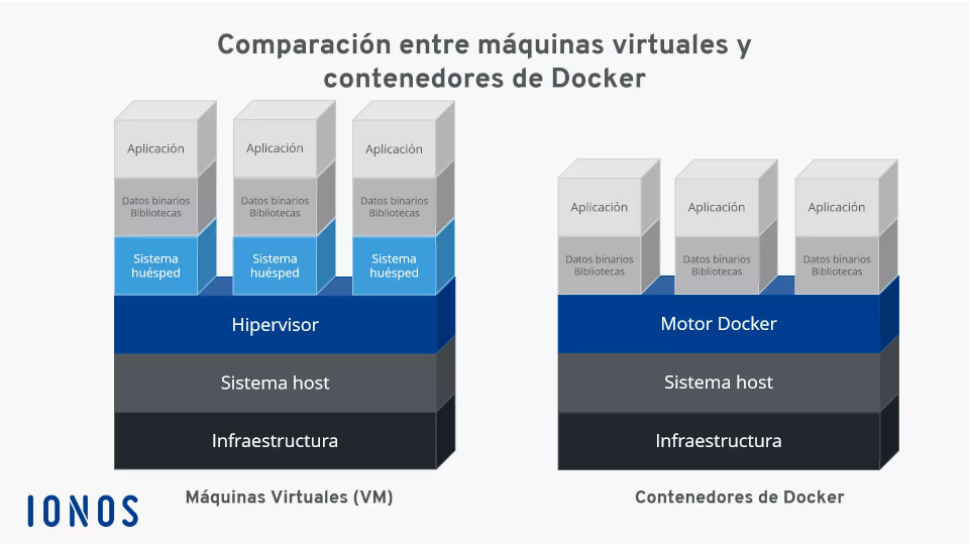

Figura 1: Comparación entre Docker y Máquinas Virtuales

Característica	**Docker**	**Máquina Virtual**
Tamaño	MB	GB
Tiempo de inicio	Segundos	Minutos
Rendimiento	Casi nativo	Overhead significativo
Aislamiento	A nivel de proceso	A nivel de hardware
Portabilidad	Alta	Media/Baja

Cuadro 1: Comparación detallada Docker vs VM

2 Instalación de Docker

2.1 Instalación en Windows

1. Descargar Docker Desktop desde `https://www.docker.com/products/docker-desktop`
2. Ejecutar el instalador
3. Reiniciar el equipo cuando se solicite
4. Verificar la instalación:

```
docker --version
docker-compose --version
docker run hello-world
```

Listing 1: Verificar instalación en Windows

2.2 Instalación en macOS

1. Descargar Docker Desktop para Mac
2. Arrastrar Docker a la carpeta Applications
3. Ejecutar Docker.app
4. Verificar la instalación

2.3 Instalación en Linux (Ubuntu)

```
# Actualizar paquetes
sudo apt update

# Instalar dependencias
sudo apt install apt-transport-https ca-certificates curl gnupg lsb-release

# Agregar clave GPG oficial de Docker
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg

# Agregar repositorio
echo "deb [arch=amd64 signed-by=/usr/share/keyrings/docker-archive-keyring.gpg] https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null

# Instalar Docker Engine
sudo apt update
sudo apt install docker-ce docker-ce-cli containerd.io

# Agregar usuario al grupo docker
sudo usermod -aG docker $USER

# Reiniciar sesi n o ejecutar:
newgrp docker

# Verificar instalaci n
docker --version
```

Listing 2: Instalación en Ubuntu

3 Conceptos Fundamentales de Docker

3.1 Imágenes y Contenedores

Imagen Docker:

- Plantilla de solo lectura con instrucciones para crear un contenedor
- Se compone de capas (layers)
- Se almacena en registros (Docker Hub, registros privados)
- Ejemplos: nginx, mysql, python:3.9

Contenedor Docker:

- Instancia en ejecución de una imagen
- Tiene su propio sistema de archivos, procesos y red
- Es efímero (los datos se pierden al eliminar el contenedor)
- Se puede iniciar, detener, eliminar

3.2 Ejemplos Prácticos: Primeros Pasos

```
# Descargar y ejecutar un contenedor de prueba
docker run hello-world

# Ejecutar un contenedor Nginx
docker run -d -p 80:80 nginx

# Ver contenedores en ejecuci n
docker ps

# Ver todos los contenedores (incluyendo detenidos)
docker ps -a

# Ver im genes descargadas
docker images

# Detener un contenedor
docker stop <container_id>

# Eliminar un contenedor
docker rm <container_id>

# Eliminar una imagen
docker rmi <image_name>
```

Listing 3: Primeros comandos con Docker

3.3 Mapeo de Puertos

El mapeo de puertos permite conectar puertos del contenedor con puertos del sistema anfitrión.

```
# Mapear puerto 80 del contenedor al 8080 del host
docker run -d -p 8080:80 nginx

# Mapear puerto aleatorio
docker run -d -p 80 nginx

# Mapear m ltiples puertos
docker run -d -p 8080:80 -p 3000:3000 myapp

# Mapear puerto espec fico de la interfaz
docker run -d -p 127.0.0.1:8080:80 nginx
```

Listing 4: Ejemplos de mapeo de puertos

Ejercicio práctico:

1. Ejecutar: `docker run -d -p 8080:80 nginx`
2. Abrir navegador en: `http://localhost:8080`
3. Deberías ver la página de bienvenida de Nginx

4 Dockerfile: Creando Nuestras Propias Imágenes

4.1 ¿Qué es un Dockerfile?

Un Dockerfile es un archivo de texto con instrucciones para construir una imagen Docker.

4.2 Sintaxis Básica del Dockerfile

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

# Metadatos
LABEL maintainer="tu-email@ejemplo.com"

# Establecer directorio de trabajo
WORKDIR /app

# Copiar archivos de requisitos
COPY requirements.txt .

# Instalar dependencias
RUN pip install -r requirements.txt

# Copiar el c digo de la aplicaci n
COPY . .

# Exponer puerto
EXPOSE 5000
```

```

# Comando para ejecutar la aplicaci n
CMD ["python", "app.py"]
```

Listing 5: Estructura básica de un Dockerfile

4.3 Instrucciones Principales del Dockerfile

Instrucción	Propósito
FROM	Imagen base
RUN	Ejecutar comandos durante la construcción
COPY	Copiar archivos del host al contenedor
ADD	Similar a COPY pero con funcionalidades extra
CMD	Comando por defecto al ejecutar el contenedor
ENTRYPOINT	Punto de entrada de la aplicación
EXPOSE	Documentar puertos expuestos
ENV	Variables de entorno
WORKDIR	Directorio de trabajo
USER	Usuario para ejecutar comandos
VOLUME	Directorios persistentes

Cuadro 2: Instrucciones principales de Dockerfile

4.4 Ejemplo Práctico: Aplicación Web Python

Paso 1: Crear estructura de archivos

```
mi-app-python/
          Dockerfile
          requirements.txt
          app.py
```

Listing 6: Estructura del proyecto

Paso 2: Crear app.py

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello():
    return '<h1> Hola  desde Docker!</h1><p>Esta es mi primera app en
   contenedor.</p>'

@app.route('/info')
def info():
    return {'version': '1.0', 'name': 'Mi App Python'}

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000, debug=True)
```

Listing 7: app.py - Aplicación Flask simple

Paso 3: Crear requirements.txt

```
Flask==2.3.3
```

Listing 8: requirements.txt

Paso 4: Crear Dockerfile

```
FROM python:3.9-slim

WORKDIR /app

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

COPY . .

EXPOSE 5000

CMD ["python", "app.py"]
```

Listing 9: Dockerfile para la aplicación Python

Paso 5: Construir y ejecutar

```
# Construir la imagen
docker build -t mi-app-python .

# Ver im genes
docker images

# Ejecutar el contenedor
docker run -d -p 5000:5000 mi-app-python

# Verificar en navegador: http://localhost:5000
```

Listing 10: Construir y ejecutar la aplicación

5 Docker Compose

5.1 ¿Qué es Docker Compose?

Docker Compose es una herramienta para definir y ejecutar aplicaciones Docker multi-contenedor.

5.2 Archivo docker-compose.yml

```
version: '3.8'

services:
  web:
    build: .
```

```
    ports:
      - "5000:5000"
    volumes:
      - .:/app
    environment:
      - FLASK_ENV=development
    depends_on:
      - redis

  redis:
    image: "redis:alpine"
    ports:
      - "6379:6379"

  database:
    image: "postgres:13"
    environment:
      - POSTGRES_DB=mydb
      - POSTGRES_USER=user
      - POSTGRES_PASSWORD=password
    volumes:
      - db_data:/var/lib/postgresql/data
    ports:
      - "5432:5432"

volumes:
  db_data:
```

Listing 11: Ejemplo de docker-compose.yml

5.3 Ejemplo Práctico: Aplicación Web + Base de Datos

Paso 1: Crear estructura del proyecto

```
app-completa/
          docker-compose.yml
          web/
                Dockerfile
                requirements.txt
                app.py
          database/
              init.sql
```

Listing 12: Estructura del proyecto con compose

Paso 2: docker-compose.yml

```
version: '3.8'

services:
  web:
    build: ./web
    ports:
      - "8000:5000"
    volumes:
      - ./web:/app
```

```
    environment:
      - DATABASE_URL=postgresql://user:password@db:5432/mydb
    depends_on:
      - db
    restart: unless-stopped

  db:
    image: postgres:13
    environment:
      - POSTGRES_DB=mydb
      - POSTGRES_USER=user
      - POSTGRES_PASSWORD=password
    volumes:
      - postgres_data:/var/lib/postgresql/data
      - ./database/init.sql:/docker-entrypoint-initdb.d/init.sql
    ports:
      - "5432:5432"
    restart: unless-stopped

  redis:
    image: redis:6-alpine
    ports:
      - "6379:6379"
    restart: unless-stopped

volumes:
  postgres_data:
```

Listing 13: docker-compose.yml para aplicación completa

Paso 3: Comandos de Docker Compose

```
# Iniciar todos los servicios
docker-compose up

# Iniciar en segundo plano
docker-compose up -d

# Ver logs
docker-compose logs

# Ver logs de un servicio espec fico
docker-compose logs web

# Detener servicios
docker-compose down

# Reconstruir y ejecutar
docker-compose up --build

# Ver servicios en ejecuci n
docker-compose ps

# Ejecutar comando en un servicio
docker-compose exec web python manage.py migrate
```

Listing 14: Comandos esenciales de Docker Compose

6 Gestión de Docker

6.1 Gestión de Contenedores

```
# Ejecutar contenedor interactivo
docker run -it ubuntu:20.04 /bin/bash

# Ejecutar comando en contenedor existente
docker exec -it <container_id> /bin/bash

# Ver logs del contenedor
docker logs <container_id>

# Ver logs en tiempo real
docker logs -f <container_id>

# Inspeccionar contenedor (informaci n detallada)
docker inspect <container_id>

# Ver estad sticas de uso de recursos
docker stats

# Ver procesos dentro del contenedor
docker top <container_id>

# Copiar archivos desde/hacia contenedor
docker cp <container_id>:/ruta/archivo ./
docker cp ./archivo <container_id>:/ruta/
```

Listing 15: Comandos para gestión de contenedores

6.2 Gestión de Imágenes

```
# Construir imagen desde Dockerfile
docker build -t mi-imagen:1.0 .

# Construir con contexto diferente
docker build -t mi-imagen:1.0 https://github.com/user/repo.git

# Etiquetar imagen
docker tag mi-imagen:1.0 mi-registry.com/mi-imagen:1.0

# Subir imagen a registro
docker push mi-registry.com/mi-imagen:1.0

# Descargar imagen
docker pull nginx:latest

# Ver historial de imagen
docker history mi-imagen:1.0

# Guardar imagen como archivo
docker save -o mi-imagen.tar mi-imagen:1.0

```

```
# Cargar imagen desde archivo
docker load -i mi-imagen.tar

# Limpiar im genes no utilizadas
docker image prune
```

Listing 16: Comandos para gestión de imágenes

6.3 Gestión de Redes

```
# Ver redes disponibles
docker network ls

# Crear red personalizada
docker network create mi-red

# Inspeccionar red
docker network inspect mi-red

# Conectar contenedor a red
docker network connect mi-red mi-contenedor

# Desconectar contenedor de red
docker network disconnect mi-red mi-contenedor

# Eliminar red
docker network rm mi-red

# Ejemplo: dos contenedores en misma red
docker run -d --name contenedor1 --network mi-red nginx
docker run -it --name contenedor2 --network mi-red alpine ping
   contenedor1
```

Listing 17: Comandos para gestión de redes

6.4 Gestión de Volúmenes

```
# Crear volumen
docker volume create mi-volumen

# Listar vol menes
docker volume ls

# Inspeccionar volumen
docker volume inspect mi-volumen

# Eliminar volumen
docker volume rm mi-volumen

# Usar volumen en contenedor
docker run -d -v mi-volumen:/data nginx

# Usar bind mount (montaje de directorio)
docker run -d -v /ruta/local:/app/data nginx
```

```
18
19 # Limpiar vol menes no utilizados
20 docker volume prune
```

Listing 18: Comandos para gestión de volúmenes

7 Ejercicios Prácticos

7.1 Ejercicio 1: Aplicación Node.js con MongoDB

Objetivo: Crear una aplicación Node.js con MongoDB usando Docker Compose.

```
1 node-mongo-app/
2           docker-compose.yml
3           backend/
4                 Dockerfile
5                 package.json
6                 server.js
7                 models/
8           frontend/
9               Dockerfile
10              package.json
11              public/
```

Listing 19: Estructura del proyecto

backend/Dockerfile:

```
1 FROM node:16-alpine
2 WORKDIR /app
3 COPY package*.json ./
4 RUN npm install
5 COPY . .
6 EXPOSE 3000
7 CMD ["npm", "start"]
```

docker-compose.yml:

```
1 version: '3.8'
2 services:
3   backend:
4     build: ./backend
5     ports:
6       - "3000:3000"
7     environment:
8       - MONGODB_URI=mongodb://mongo:27017/mydb
9     depends_on:
10      - mongo
11    volumes:
12      - ./backend:/app
13      - /app/node_modules
14
15  mongo:
16    image: mongo:5
```

```
    ports:
      - "27017:27017"
    volumes:
      - mongo_data:/data/db

volumes:
  mongo_data:
```

7.2 Ejercicio 2: WordPress con MySQL

```
version: '3.8'
services:
  wordpress:
    image: wordpress:latest
    ports:
      - "8080:80"
    environment:
      - WORDPRESS_DB_HOST=db
      - WORDPRESS_DB_USER=wordpress
      - WORDPRESS_DB_PASSWORD=password
      - WORDPRESS_DB_NAME=wordpress
    volumes:
      - wp_data:/var/www/html
    depends_on:
      - db

  db:
    image: mysql:8.0
    environment:
      - MYSQL_ROOT_PASSWORD=rootpassword
      - MYSQL_DATABASE=wordpress
      - MYSQL_USER=wordpress
      - MYSQL_PASSWORD=password
    volumes:
      - db_data:/var/lib/mysql

volumes:
  wp_data:
  db_data:
```

Listing 20: WordPress con Docker Compose

8 Buenas Prácticas

8.1 Security Best Practices

- **No ejecutar como root**: Usar usuario no privilegiado
- **Imágenes oficiales**: Usar imágenes de proveedores oficiales
- **Escaneo de vulnerabilidades**: Usar docker scan

- **Secrets management**: No incluir credenciales en imágenes

```
FROM node:16-alpine

# Crear usuario no root
RUN addgroup -g 1001 -S nodejs
RUN adduser -S nextjs -u 1001

WORKDIR /app
COPY --chown=nextjs:nodejs . .

USER nextjs

EXPOSE 3000
CMD ["npm", "start"]
```

Listing 21: Dockerfile seguro

8.2 Optimización de Imágenes

- Usar imágenes base pequeñas (alpine, slim)
- Minimizar número de capas
- Usar .dockerignore
- Multi-stage builds para aplicaciones compiladas

```
# Stage 1: Build
FROM golang:1.19 as builder
WORKDIR /app
COPY . .
RUN go build -o myapp

# Stage 2: Runtime
FROM alpine:latest
RUN apk --no-cache add ca-certificates
WORKDIR /root/
COPY --from=builder /app/myapp .
CMD ["./myapp"]
```

Listing 22: Multi-stage build para Go

9 Solución de Problemas Comunes

9.1 Comandos de Diagnóstico

```
# Ver uso de recursos
docker system df

# Informaci n del sistema
docker system info

# Eventos de Docker en tiempo real
docker system events

# Ver espacio utilizado
docker system df -v

# Limpiar recursos no utilizados
docker system prune

# Limpiar todo (incluyendo vol menes)
docker system prune -a --volumes

# Ver detalles de contenedor
docker inspect <container_id>

# Ver variables de entorno del contenedor
docker exec <container_id> env
```

Listing 23: Comandos para troubleshooting

9.2 Problemas Comunes y Soluciones

Problema	**Solución**
"Port already in use"	Cambiar puerto o detener proceso que usa el puerto
Çannot connect to the Docker daemon"	Verificar que Docker esté ejecutándose
"No space left on device"	Limpiar imágenes y contenedores no utilizados
"Permission denied"	Ejecutar con sudo o agregar usuario al grupo docker
Çontainer exits immediately"	Verificar CMD/ENTRYPOINT y logs del contenedor
"Build context too large"	Usar .dockerignore y optimizar estructura

Cuadro 3: Problemas comunes y sus soluciones

10 Conclusión

Docker es una herramienta poderosa que ha revolucionado la forma en que desarrollamos, empaquetamos y desplegamos aplicaciones. Con esta guía, has aprendido:

- Los conceptos fundamentales de Docker
- Cómo crear y gestionar contenedores
- Cómo construir imágenes personalizadas

- Cómo orquestar múltiples servicios con Docker Compose
- Mejores prácticas de seguridad y optimización