

# Dockerfile Avanzado

---

**Arturo Silvelo**

Try New Roads

## Ejemplos

---

Esta presentación contiene ejemplos prácticos que demuestran las técnicas avanzadas de Dockerfile. Todos los ejemplos utilizan una aplicación de demostración ubicada en la carpeta `app/`.

La aplicación consiste en un servidor web desarrollado en **Node.js** con **Express** que expone los siguientes endpoints:

- `GET /` - Endpoint principal que devuelve un mensaje de bienvenida
- `GET /secret` - Endpoint que muestra el valor de la variable de entorno `SECRET`

La aplicación utiliza las siguientes **variables de entorno**:

- `SECRET` - Variable que contiene un valor secreto mostrado en el endpoint `/secret`
- `PORT` - Puerto en el que se ejecuta el servidor (por defecto: 3000)

**Nota:** Para ejecutar los ejemplos, asegúrate de estar en el directorio `ejemplos/` y usar los comandos tal como se muestran en cada sección.

# Ejemplo 1: Mínimo

- Contenido del Dockerfile

```
FROM alpine:3.20  
CMD ["echo", "Hola Docker!"]
```

- Construcción de la imagen

```
docker build -f 1.minimo/Dockerfile -t app-minimo 1.minimo
```

- Ejecución de la imagen

```
docker run --rm app-minimo
```

## Ejemplo 2: Multi-stage

Este ejemplo muestra cómo crear una imagen Docker optimizada usando `multi-stage`.

- Dockerfile.before: Imagen sin multi-stage

```
FROM node:20
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm test
CMD ["node", "app.js"]
```

- Creación:

```
docker build -f 2.multi-stage/Dockerfile.before -t app-simple app
```

- Ejecución y comprobación:

```
docker run --rm --init -p 3000:3000 app-simple
curl http://localhost:3000
```



- Dockerfile.after: Imagen con multi-stage.

```
FROM node:20 AS build
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .

FROM node:20 AS test
WORKDIR /app
COPY --from=build /app /app
RUN npm test

FROM node:20 AS production
WORKDIR /app
COPY --from=build /app/app.js .
COPY --from=build /app/package.json .
RUN npm install --production
CMD ["node", "app.js"]
```

- Creación:

```
docker build -f 2.multi-stage/Dockerfile.after -t app-multi-stage app
```

- Ejecución y comprobación:

```
docker run --rm --init -p 3000:3000 app-multi-stage  
curl http://localhost:3000
```

- Comprobación reducción:

```
docker image ls
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
app-simple	latest	2fa07e3c943c	4 minutes ago	1.26GB
app-multi-stage	latest	e57316ecfb25	5 minutes ago	1.2GB

# Ejemplo 3: Optimización Capas

Este ejemplo muestra cómo optimizar las capas de Docker progresivamente para mejorar el rendimiento y reducir el tamaño de las imágenes.

- **Dockerfile:** Imagen con capas mal optimizadas (versión original)

```
FROM node:20
WORKDIR /app
COPY . .
RUN apt-get update
RUN apt-get install -y build-essential
RUN npm install
RUN npm test
CMD ["node", "app.js"]
```

- Creación:

```
docker build -f 3.optimizacion-capas/Dockerfile -t app-mal-optimizado app
```

- Dockerfile.1-agrupa-run: Agrupando comandos RUN

```
FROM node:20
WORKDIR /app
COPY . .
RUN apt-get update && \
    apt-get install -y build-essential && \
    npm install
RUN npm test
CMD ["node", "app.js"]
```

- Creación:

```
docker build -f 3.optimizacion-capas/Dockerfile.1 -t app-fase1 app
```

- Dockerfile.2-limpieza: Agrupando RUN y limpiando archivos temporales

```
FROM node:20
WORKDIR /app
COPY . .
RUN apt-get update && \
    apt-get install -y build-essential && \
    npm install && \
    apt-get clean && \
    rm -rf /var/lib/apt/lists/*
RUN npm test
CMD ["node", "app.js"]
```

- Creación:

```
docker build -f 3.optimizacion-capas/Dockerfile.2 -t app-fase2 app
```

```
FROM node:20
WORKDIR /app

# Instalar dependencias del sistema primero (se cachea)
RUN apt-get update && \
    apt-get install -y build-essential && \
    apt-get clean && \
    rm -rf /var/lib/apt/lists/*

# Copiar solo package.json primero (mejor cache)
COPY package*.json ./
RUN npm install

# Copiar código fuente al final
COPY . .
RUN npm test

CMD ["node", "app.js"]
```



- Dockerfile.3-optimizado: Capas optimizadas y bien organizadas
- Creación:

```
docker build -f 3.optimizacion-capas/Dockerfile.3 -t app-fase3 app
```

- Comparación de mejoras:

```
# Ver tamaños de las imágenes
docker images | grep app-

# Ver número de capas
docker history app-mal-optimizado
docker history app-fase1
docker history app-fase2
docker history app-fase3
```

- Prueba de eficiencia del cache:

```
# Modificar código de la aplicación
res.send('¡Hola desde Node.js en Docker!'); -> res.send('¡Hola Mundo!');

# Comparar tiempos de rebuild
docker build -f 3.optimizacion-capas/Dockerfile -t app-mal-optimizado app
docker build -f 3.optimizacion-capas/Dockerfile.3 -t app-fase3 app
```

**Nota:** La versión optimizada será significativamente más rápida al hacer cambios en el código, ya que reutiliza las capas de instalación de dependencias del sistema y npm.

## Ejemplo 4: ARG y ENV

Este ejemplo demuestra el uso de variables `ARG` (build time) y `ENV` (runtime) en Docker.

```
FROM node:20

# ARG para el puerto (se puede pasar en build time)
ARG PORT=3000

# ENV para el secreto (variable de entorno)
ENV SECRET="mi-secreto-por-defecto"

# Pasar el valor de ARG a una ENV para que esté disponible en runtime
ENV PORT=$PORT

WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm test

# Exponer el puerto
EXPOSE $PORT

CMD ["node", "app.js"]
```

- **Dockerfile:** Uso de ARG para PORT y ENV para SECRET
- Creación con valores por defecto:

```
docker build -f 4.arg-env/Dockerfile -t app-arg-env app
```

- Creación con puerto personalizado:

```
docker build -f 4.arg-env/Dockerfile --build-arg PORT=8080 -t app-arg-env-8080 app
```

- Verificación:

```
# Ver variables de entorno del contenedor
docker run --rm app-arg-env env | grep -E "(PORT|SECRET)"
docker run --rm app-arg-env-8080 env | grep -E "(PORT|SECRET)"
```

- Ejecución y comprobación:

```
docker run --rm --init -p 3000:3000 app-arg-env
docker run --rm --init -p 8080:8080 app-arg-env-8080
curl http://localhost:3000
curl http://localhost:3000/secret
curl http://localhost:3000
curl http://localhost:3000/secret
```

# Ejemplo 5: Secretos



Este ejemplo demuestra las diferentes formas de gestionar secretos en Docker: la **mala práctica** (hardcodeado) y las **buenas prácticas** (variables de entorno y archivos montados).

- Dockerfile: Secreto hardcodeado

```
FROM node:20

# ENV para el secreto (variable de entorno)
ENV SECRET="mi-secreto-por-defecto"
....
```

- Creación (mala práctica):

```
docker build -f 5.secretos/Dockerfile -t app-secreto-malo app
```

- Dockerfile.1: Sin secretos hardcoded

```
FROM node:20  
# ENV SECRET="mi-secreto-por-defecto"  
...
```

- Creación (buena práctica):

```
docker build -f 5.secretos/Dockerfile.1 -t app-secreto-bueno app
```

- Ejecución **sin secreto** (muestra "undefined"):

```
docker run --rm --init -p 3000:3000 app-secreto-bueno  
curl http://localhost:3000/secret
```

- Ejecución con **variable de entorno**:

```
docker run --rm --init -p 3000:3000 -e SECRET="mi-secreto-desde-env" app-secreto-bueno  
curl http://localhost:3000/secret
```

- Ejecución con **archivo montado**:

```
docker run --rm --init -p 3000:3000 -v "$(pwd)/5.secretos/my_secret.txt:/run/secrets/secret.txt" app-secreto-bueno  
curl http://localhost:3000/secret
```

- Verificación del problema de seguridad:

```
# Ver que el secreto está expuesto en la imagen mala  
docker run --rm app-secreto-malo env | grep SECRET
```

```
# Ver que no hay secreto hardcodeado en la imagen buena  
docker run --rm app-secreto-bueno env | grep SECRET
```