Optimización de imágenes

Arturo Silvelo

Try New Roads

Introducción: ¿Qué es un Dockerfile?

Un **Dockerfile** es un archivo de texto que contiene instrucciones para construir una imagen Docker. Define el sistema operativo base, las dependencias, el código fuente, variables de entorno y el comando de arranque del contenedor.

Permite automatizar la creación de entornos reproducibles y portables, facilitando el despliegue de aplicaciones en cualquier sistema compatible con Docker.

Ejemplo mínimo:

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

Optimización de imágenes Docker

Optimizar imágenes Docker es fundamental para mejorar el rendimiento, la seguridad y la eficiencia en el despliegue de aplicaciones. Una imagen optimizada:

- Se descarga y despliega más rápido.
- Ocupa menos espacio en disco y consume menos ancho de banda.
- Reduce la superficie de ataque y facilita la actualización.
- Es más fácil de mantener y depurar.

Para lograrlo, es importante aplicar varias técnicas y buenas prácticas, como el uso de multi-stage builds y la optimización de capas.

¿Cómo optimizar imágenes Docker?

- Usar imágenes base ligeras (alpine, slim)
- Multi-stage builds para separar dependencias de build y producción
- Minimizar el número de capas combinando instrucciones RUN, COPY, etc.
- Eliminar archivos temporales y cachés en la misma instrucción

Multi-stage builds

Permite crear imágenes más ligeras y seguras usando varias etapas en el Dockerfile.

- Solo la última etapa contiene lo necesario para ejecutar la aplicación.
- Las etapas previas pueden tener herramientas de compilación y dependencias de desarrollo, que no se copian a la imagen final.

Ventajas:

- Reduce el tamaño de la imagen final.
- Elimina archivos y dependencias de desarrollo.
- Nombra las etapas (`AS build`, `AS produccion`) para facilitar el copiado selectivo.
- Usa imágenes base "slim" o "alpine" cuando sea posible.

No es necesario construir todas las fases del Dockerfile. Usando la opción --target, puedes crear una imagen solo hasta una etapa concreta.

Ejemplo: Multi-stage build

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

COPY --from=build /app/dist ./dist
CMD ["node", "dist/index.js"]
```

En este ejemplo:

- La etapa `build` instala dependencias y compila la app.
- La etapa `production` solo copia lo necesario para ejecutar.

¿Cómo probar y comparar?

Puedes comparar el tamaño de las imágenes construyendo ambos Dockerfiles en la carpeta `ejemplos/multi-stage`:

```
# Construir imagen tradicional
# Construir imagen multi-stage
# Ver tamaños
docker images | findstr app-
```

Luego puedes ejecutar ambas imágenes:

```
docker run --rm -p 3000:3000 app-tradicional
# o
docker run --rm -p 3000:3000 app-multistage
```

Abre http://localhost:3000 para comprobar que ambas funcionan igual, pero la imagen multi-stage será más ligera.

Capas en Docker: ¿Qué son?

Cada instrucción (FROM, RUN, COPY, etc.) en un Dockerfile crea una **capa**.

- Las capas son archivos intermedios que Docker almacena en cache.
- Si una capa no cambia, Docker la reutiliza en builds futuros.
- Esto acelera la construcción y ahorra espacio en disco.

Ejemplo:

```
FROM node:20  # Crea una capa
COPY . .      # Otra capa
RUN npm install  # Otra capa
```

Optimización de capas: Estrategias

Optimizar el número y el contenido de las capas en tus imágenes Docker es clave para reducir el tamaño, acelerar los builds y facilitar el mantenimiento.

Algunas estrategias recomendadas son:

- **Agrupa comandos en una sola instrucción RUN:**
  Así reduces el número de capas y aprovechas mejor la cache de Docker.
- **Elimina archivos temporales y de build en la misma RUN:**
  Si los borras en la misma instrucción donde se crean, no quedan guardados en ninguna capa.
- **Incluye un archivo `.dockerignore`:**
  Excluye archivos y carpetas que no necesitas en la imagen (tests, docs, node_modules, etc.), evitando que se copien y generen capas innecesarias.

El orden de las capas y la cache

- **El orden de las capas influye en el tiempo de build:**
  Si modificas una instrucción, todas las capas posteriores se reconstruyen. Coloca primero las instrucciones que cambian menos (por ejemplo, dependencias) y después las que cambian más (código fuente).

Ejemplo 1: Dockerfile mal optimizado

```
FROM node:20

COPY . .
RUN apt-get update
RUN apt-get install -y build-essential
RUN npm install
RUN npm run build
CMD ["node", "app.js"]
```

Ejemplo 2: Agrupar RUN en una sola instrucción

```
FROM node:20

COPY . .
RUN apt-get update && \
    apt-get install -y build-essential && \
    npm install && \
    npm run build
CMD ["node", "app.js"]
```

Mejora:

- Menos capas, build más rápido.

Ejemplo 3: Eliminar archivos temporales y dependencias de build

```
FROM node:20

COPY . .
RUN apt-get update && \
    apt-get install -y build-essential && \
    npm install && \
    npm run build && \
    apt-get purge -y build-essential && \
    rm -rf /var/lib/apt/lists/*
CMD ["node", "app.js"]
```

Mejora:

- La imagen final es más ligera, sin archivos de build ni temporales.

Ejemplo 4: Reordenar capas para aprovechar la cache

```
FROM node:20

COPY package*.json .
RUN apt-get update && \
    apt-get install -y build-essential && \
    npm install && \
    apt-get purge -y build-essential && \
    rm -rf /var/lib/apt/lists/*
COPY . .
RUN npm run build
CMD ["node", "app.js"]
```

Mejora:

- Si solo cambias el código fuente, la instalación de dependencias se mantiene en cache y el build es mucho más rápido.

Ejemplo 5: Añadir .dockerignore

`.dockerignore`:

```
node_modules
tests
docs
*.log
```

Impacto:

- No se copian archivos innecesarios al contexto de build, la imagen es más pequeña y el build más eficiente.

Variables ARG y ENV en Docker

Las variables `ARG` y `ENV` permiten personalizar tanto la construcción de la imagen como el comportamiento de los contenedores.

- **ARG** se usa para definir valores que solo existen durante el proceso de build (por ejemplo, elegir una versión de Node o una URL de descarga).
- **ENV** define variables de entorno que estarán disponibles en la imagen final y en los contenedores que se creen a partir de ella.

Estas variables ayudan a crear imágenes más flexibles, reutilizables y adaptables a diferentes entornos (desarrollo, producción, etc.).

Buenas prácticas y advertencias

- Usa `ARG` para valores temporales o que no deban quedar en la imagen.
- Usa `ENV` para configuración que necesita el contenedor en ejecución.
- No pongas secretos en `ENV` ni en el Dockerfile.
- Documenta las variables y su propósito.

Ejemplo práctico: Uso conjunto de ARG y ENV

Puedes probar este ejemplo en la carpeta `ejemplos/arg-env`:

```
FROM node:20

# Definimos una variable de build (ARG)
ARG MENSAJE="Hola desde ARG"

# Pasamos el valor de ARG a una variable de entorno (ENV)
ENV MENSAJE_ENV=$MENSAJE

# Script que imprime el valor de la variable
COPY app.js .

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

¿Cómo probarlo?

1. Construye la imagen con un valor personalizado:

```
docker build -t arg-env-demo --build-arg MENSAJE="¡Mensaje personalizado!" .
```

2. Ejecuta el contenedor:

```
docker run --rm arg-env-demo
```

Verás en la salida el valor que pasaste por ARG.

Gestión de secretos en Docker

Los secretos son datos sensibles (contraseñas, tokens, claves API, certificados, etc.) que permiten acceder a recursos protegidos. Es fundamental protegerlos y evitar que queden expuestos en la imagen o el código fuente.

En Docker, los secretos no deben almacenarse en la imagen. Deben gestionarse de forma externa y segura.

Riesgos de una mala gestión:

- Los secretos quedan guardados en la imagen y pueden ser leídos por cualquiera con acceso.
- Si subes la imagen a un registro público, los secretos quedan expuestos.
- Los secretos pueden filtrarse en logs, capas intermedias o sistemas de CI/CD.

Buenas prácticas:

- Nunca incluyas secretos en el Dockerfile ni en la imagen.
- Usa mecanismos externos: Docker secrets (Swarm), variables de entorno solo en ejecución, archivos montados como volúmenes.
- Usa archivos `.env` solo para desarrollo y nunca los subas a git.
- Documenta cómo inyectar los secretos en producción.

Ejemplo: evolución en la gestión de secretos

Veamos tres formas de gestionar un secreto (por ejemplo, una contraseña de base de datos):

1. Dockerfile con secreto en claro (mala práctica)

```
FROM alpine:3.20
ENV DB_PASSWORD=supersecreto
CMD ["sh", "-c", "echo El password es: $DB_PASSWORD"]
```

Problema: El secreto queda guardado en la imagen y es visible para cualquiera.

2. Variable de entorno solo en ejecución

```
FROM alpine:3.20
CMD ["sh", "-c", "echo El password es: $DB_PASSWORD"]
```

Se ejecuta así:

```
docker run -e DB_PASSWORD=supersecreto <imagen>
```

Ventaja: El secreto no está en la imagen, solo se pasa al contenedor.

3. Montar archivo externo

```
FROM alpine:3.20
CMD ["sh", "-c", "echo El password es: $(cat /run/secreto.txt)"]
```

Se ejecuta así:

```
docker run -v $(pwd)/db_password.txt:/run/secreto.txt <imagen>
```

Ventaja: El secreto está en un archivo externo, no en la imagen.