Creación de imágenes Docker

1) ¿Qué es una imagen Docker?

Una **imagen** es un "paquete" **inmutable** que contiene:

- un sistema base (p. ej. alpine, ubuntu, eclipse-temurin, nginx)
- tus ficheros (código, configuración, assets)
- dependencias necesarias
- metadatos: comando de arranque, puertos expuestos, variables, etc.

Con una imagen creas **contenedores**, que son **instancias en ejecución** de esa imagen.

2) ¿Cómo se crea una imagen?

Se crea con un **Dockerfile**, que es una receta con instrucciones.
El comando típico es:

docker build -t nombre:tag .

- -t pone nombre y versión (tag)
- . es el **contexto**: la carpeta cuyos archivos Docker puede copiar durante el build.

3) Dockerfile: instrucciones más importantes

FROM

Define la **imagen base** (punto de partida).

FROM nginx:alpine

WORKDIR

Define carpeta de trabajo dentro de la imagen.

WORKDIR /app

COPY / ADD

Copia archivos desde tu proyecto hacia la imagen.

COPY src /app/src

RUN

Ejecuta comandos **durante el build** (instalar dependencias, compilar…).

RUN apt-get update && apt-get install -y curl

EXPOSE

Indica un puerto que la app usa (informativo, no publica el puerto).

EXPOSE 8080

CMD y ENTRYPOINT

Define qué se ejecuta **cuando arranca el contenedor**.

- CMD se puede sobrescribir fácilmente al hacer docker run ... otrocomando
- ENTRYPOINT fija el ejecutable principal.

Ejemplo:

ENTRYPOINT ["java","-jar","app.jar"]

4) Capas (layers) y caché

Cada instrucción del Dockerfile (sobre todo RUN, COPY, ADD) crea una **capa**.
Docker reutiliza capas si no han cambiado → **builds más rápidos**.

Por eso es buena práctica:

- copiar primero archivos que cambian poco (ej. pom.xml)
- descargar dependencias
- luego copiar el código (que cambia más)

Ejemplo típico en Java:

1. COPY pom.xml
2. RUN mvn dependency:go-offline
3. COPY src
4. RUN mvn package

5) Contexto de build y .dockerignore

Docker solo puede "ver" lo que está dentro del **contexto** (la carpeta del build).
Si tienes archivos pesados (como target/, node_modules/, .git/) conviene ignorarlos con .dockerignore para:

- builds más rápidos
- imágenes más limpias

Ejemplo:

target

.git

node_modules

.env

6) Imagen ≠ Contenedor (idea clave)

- **Imagen**: plantilla, estática, no cambia.
- **Contenedor**: ejecución real, con un sistema de archivos "vivo", procesos, red…

Cuando paras y borras el contenedor, desaparece (si usas --rm).
Los datos que deben persistir van en:

- **volúmenes**
- o bases de datos externas
- o ficheros montados (-v)

7) Multi-stage builds (muy útil en Java)

Sirve para que la imagen final sea **más pequeña y segura**.

Ejemplo mental:

- Etapa 1: imagen grande con Maven/JDK → compila el JAR
- Etapa 2: imagen ligera con JRE → solo ejecuta el JAR

Resultado: imagen final sin Maven, sin código fuente, solo el artefacto.

8) Tags y versionado

Una imagen se identifica por:

- repo:tag (p. ej. spring-mini:1.0)
- o por un digest (hash)

Buenas prácticas:

- no usar solo latest en clase/prod
- usar tags por versión: 1.0, 1.1, 2026-02-03, etc.

9) Errores típicos al crear imágenes (y cómo explicarlos)

- **"No encuentra archivos"** → mal contexto (docker build no se ejecutó en la carpeta correcta)
- **"COPY failed"** → el archivo no está en el contexto o está en .dockerignore
- **Build lento** → no se aprovecha caché (orden de COPY/RUN mal)
- **La app no responde** → puerto mal mapeado (-p host:contenedor)
- **Problemas de permisos** → rutas/volúmenes en Windows/Linux

Ejemplos creación de imágenes Docker (Java y servidor web)

Ejemplo 0: Hello World!

Estructura

```
hello-docker/
  hello.sh
  Dockerfile
```

1) Script hello.sh

```
#!/bin/sh
VALOR="${1:-World}"
echo "Hello World + ${VALOR}!"
```

${1:-World} significa: "usa el primer argumento; si no existe, usa World".

2) Dockerfile

```
FROM alpine:3.20
WORKDIR /app
COPY hello.sh .
RUN chmod +x hello.sh
ENTRYPOINT ["./hello.sh"]
```

3) Build de la imagen

docker build -t hello-world:1.0 .

4) Ejecutar

Sin valor

```
docker run --rm hello-world:1.0
```

Salida:

Hello World + World!

Con valor (por ejemplo “Jose”)

```
docker run --rm hello-world:1.0 Jose
```

Salida:

Hello World + Jose!

Ejemplo 1: Hola Mundo en Java (sin frameworks)

Enunciado: Crear una imagen Docker que compile y ejecute un programa Java sencillo que muestre un mensaje por consola.

Estructura

```
java-hola/
  Hello.java
  Dockerfile
```

Código

Hello.java:

```
public class Hello {
  public static void main(String[] args) {
    System.out.println("Hola desde Docker + Java!");
  }
}
```

Dockerfile (multi-stage: compila y ejecuta):

```
FROM eclipse-temurin:21-jdk AS build
WORKDIR /app
COPY Hello.java .
RUN javac Hello.java

FROM eclipse-temurin:21-jre
WORKDIR /app
COPY --from=build /app/Hello.class .
CMD ["java", "Hello"]
```

Comandos

```
docker build -t java-hola:1.0 .
docker run --rm java-hola:1.0
```

Explicación

- Se usa un build multi-stage: en la primera imagen se compila (JDK) y en la segunda se ejecuta (JRE).
- La imagen final es más ligera porque no incluye el compilador.
- docker run --rm elimina el contenedor cuando termina.

Ejemplo 2: Servidor web estático con Nginx

Enunciado: Crear una imagen Docker basada en Nginx que sirva una página HTML estática.

Estructura

```
web-nginx/
  index.html
  Dockerfile
```

Código

index.html:

```
<!doctype html>
<html>
  <body>
    <h1>Hola desde Nginx en Docker</h1>
  </body>
</html>
```

Dockerfile:

```
FROM nginx:alpine
COPY index.html /usr/share/nginx/html/index.html
EXPOSE 80
```

Comandos

```
docker build -t web-nginx:1.0 .
docker run --rm -p 8080:80 web-nginx:1.0
```

Explicación

- Nginx ya viene configurado para servir contenido estático desde /usr/share/nginx/html.
- Se copia el index.html a esa ruta durante el build.
- El mapeo -p 8080:80 expone el puerto 80 del contenedor como 8080 en el host.

Ejemplo 3: Java lee un fichero externo de alumnos (CSV) y lo muestra por consola

Enunciado: Crear una imagen Docker para una aplicación Java que lea un fichero externo (alumnos.csv) montado como volumen y muestre los datos por consola.

Estructura

```
java-alumnos/
  Main.java
  Dockerfile
  datos/
    alumnos.csv
```

Fichero de datos (externo)

datos/alumnos.csv:

```
id,nombre,curso,email
1,Ana,1ASIR,ana@centro.es
2,Carlos,2DAW,carlos@centro.es
3,Marina,1DAW,marina@centro.es
```

Código

Main.java:

```
import java.nio.file.*;
import java.io.IOException;
import java.util.List;

public class Main {
    public static void main(String[] args) throws IOException {
        // Ruta por defecto dentro del contenedor
        String ruta = (args.length > 0) ? args[0] : "/data/alumnos.csv";

        Path path = Paths.get(ruta);
        if (!Files.exists(path)) {
```

```
            System.err.println("ERROR: No existe el fichero: " +
path.toAbsolutePath());
            System.err.println("Pista: monta el volumen con -v
./datos:/data");
            System.exit(1);
        }

        List<String> lineas = Files.readAllLines(path);
        if (lineas.isEmpty()) {
            System.out.println("El fichero está vacío.");
            return;
        }

        // Saltamos cabecera
        System.out.println("=== LISTADO DE ALUMNOS ===");
        for (int i = 1; i < lineas.size(); i++) {
            String[] partes = lineas.get(i).split(",");
            if (partes.length < 4) continue;

            String id = partes[0].trim();
            String nombre = partes[1].trim();
            String curso = partes[2].trim();
            String email = partes[3].trim();

            System.out.printf("- (%s) %s | %s | %s%n", id, nombre, curso,
email);
        }
    }
}
```

Dockerfile

```
FROM eclipse-temurin:21-jdk AS build
WORKDIR /app
COPY Main.java .
RUN javac Main.java

FROM eclipse-temurin:21-jre
WORKDIR /app
COPY --from=build /app/Main.class .
# La app espera el CSV en /data/alumnos.csv (lo montaremos como volumen)
CMD ["java", "Main", "/data/alumnos.csv"]
```

Comandos

```
docker build -t java-alumnos:1.0 .
```

```
# Linux/macOS
docker run --rm -v "$(pwd)/datos:/data" java-alumnos:1.0

# Windows PowerShell
docker run --rm -v "${PWD}\datos:/data" java-alumnos:1.0
```

Explicación

- El fichero alumnos.csv no se mete en la imagen: se monta desde el host con -v (volúmenes).
- Esto permite cambiar los datos sin reconstruir la imagen.
- Es el patrón habitual: la imagen contiene el software y el volumen contiene datos/configuración.

Ejemplo 4: Spring Boot (JAR) con Docker multi-stage

Enunciado: Crear una imagen Docker para un proyecto Spring Boot (Maven) que compile el JAR dentro de Docker y lo ejecute en una imagen JRE ligera.

Estructura (proyecto típico)

```
spring-mini/
  pom.xml
  mvnw
  .mvn/
  src/
  Dockerfile
```

Dockerfile

```
FROM maven:3.9-eclipse-temurin-21 AS build
WORKDIR /app

COPY pom.xml .
COPY .mvn .mvn
COPY mvnw mvnw
RUN chmod +x mvnw
RUN ./mvnw -q -DskipTests dependency:go-offline

COPY src src
RUN ./mvnw -q -DskipTests package

FROM eclipse-temurin:21-jre
WORKDIR /app
COPY --from=build /app/target/*.jar app.jar
EXPOSE 8080
ENTRYPOINT ["java","-jar","app.jar"]
```

Comandos

```
docker build -t spring-mini:1.0 .
docker run --rm -p 8080:8080 spring-mini:1.0
```

Explicación

- La etapa build usa Maven para compilar el proyecto dentro del contenedor.
- Se separa la descarga de dependencias (dependency:go-offline) para aprovechar la cache en builds posteriores.
- La imagen final solo contiene el JAR y un JRE, por lo que es más ligera y segura.