GUÍA PASO A PASO

Imagen de Docker más ligera para Next.js

El Dockerfile multi-stage completo y comentado para Next.js con output standalone, el .dockerignore, cómo elegir la imagen base, un script que comprueba que los estáticos responden y cómo medir el antes y el después. Con la checklist.

PARA	Apps de Next.js (App Router o Pages) que se despliegan como imagen de Docker
PROBADO CON	Next.js 16.4.6 y Node 22.22.3: salida standalone, server.js y el script de comprobación. El Dockerfile no se construyó en esta máquina
ACTUALIZADA	19 de noviembre de 2026
DOCUMENTACIÓN	nextjs.org/docs/app/api-reference/config/next-config-js/output · docs.docker.com/build/building/multi-stage
ARTÍCULO	ecomlabs.dev/g/docker
AUTOR	Luis Morales · @luismoralesdev

Índice

Antes de empezar	Qué necesitas y qué se probó
Paso 01	Activar standalone
Paso 02	El .dockerignore
Paso 03	El Dockerfile completo, comentado
Paso 04	Construir, correr y comprobar
Paso 05	Elegir la base y medir
Al terminar	Checklist

ANTES DE EMPEZAR

Qué necesitas y qué se probó

Necesitas Next.js 12.2 o posterior (15 o posterior para `next.config.ts`; probado con 16.4.6, la más reciente en npm el 19 nov 2026), un `package-lock.json`, una carpeta `public` aunque esté vacía, y Docker con BuildKit.

Corrimos `next build` con standalone, `server.js` y el script del paso 04. El `docker build` no: el motor de Docker no arrancó en la máquina de prueba. El Dockerfile sigue el ejemplo oficial `with-docker` de Next.js.

PASO 01

Activar standalone

```
// next.config.ts
import type { NextConfig } from "next";
const nextConfig: NextConfig = {
  output: "standalone",
};
export default nextConfig;
```

Corre `npm run build` en local y revisa `.next/standalone`: debe tener `server.js`, un `package.json` y un `node_modules` reducido. No trae `public` ni `.next/static`; eso lo copia el Dockerfile.

Archivos leídos con rutas armadas en tiempo de ejecución: `outputFileTracingIncludes`. Monorepo: `outputFileTracingRoot` en la raíz.

PASO 02

El .dockerignore

```
# Lo genera el contenedor; el tuyo trae binarios de tu sistema
node_modules
.next
.git
# Next.js copia .env y .env.production a .next/standalone
.env*
!.env.example
npm-debug.log*
coverage
test-results
playwright-report
*.md
Dockerfile*
.dockerignore
```

Va en la raíz del contexto, junto al Dockerfile. Si guardas valores no secretos del build en `.env.production`, agrega la línea `!.env.production`; gana la última regla que coincide.

PASO 03

El Dockerfile completo, comentado

```
# syntax=docker/dockerfile:1
# Versión exacta para las tres etapas (24.22.3: parche del 10 nov 2026).
ARG NODE_VERSION=24.22.3-bookworm-slim
# ---------- 1) deps: instala todas las dependencias ----------
FROM node:${NODE_VERSION} AS deps
WORKDIR /app
# Solo manifiesto y lockfile: capa en caché hasta que cambien las deps.
COPY package.json package-lock.json ./
# npm ci respeta el lockfile e incluye devDependencies (para compilar).
RUN npm ci --no-audit --no-fund
# ---------- 2) build: compila con output "standalone" ----------
FROM node:${NODE_VERSION} AS build
WORKDIR /app
ENV NEXT_TELEMETRY_DISABLED=1
COPY --from=deps /app/node_modules ./node_modules
# El .dockerignore evita que tu node_modules local pise este.
COPY . .
# Variables NEXT_PUBLIC_: se incrustan en el JS, van aquí.
# ARG NEXT_PUBLIC_SITE_URL
# ENV NEXT_PUBLIC_SITE_URL=$NEXT_PUBLIC_SITE_URL
RUN npm run build
# ---------- 3) runner: lo único que se publica ----------
FROM node:${NODE_VERSION} AS runner
WORKDIR /app
# HOSTNAME=0.0.0.0: Docker define HOSTNAME con el hostname del
# contenedor y server.js escucharía solo en esa dirección.
ENV NODE_ENV=production \
    NEXT_TELEMETRY_DISABLED=1 \
    PORT=3000 \
    HOSTNAME=0.0.0.0
# Servidor mínimo + node_modules rastreado.
COPY --from=build --chown=node:node /app/.next/standalone ./
# Standalone no trae estos dos: sin ellos, 404 en JS, CSS e imágenes.
COPY --from=build --chown=node:node /app/.next/static ./.next/static
COPY --from=build --chown=node:node /app/public ./public
# Sin root. El usuario node ya existe en la imagen oficial.
USER node
EXPOSE 3000
CMD ["node", "server.js"]
```

CUIDADO

Los secretos no van en el Dockerfile ni en un `.env` del contexto: pásalos al arrancar el contenedor (`docker run -e`, la configuración de tu plataforma o tu gestor de secretos). Un `ARG` tampoco sirve para secretos: Docker advierte que los argumentos de build se ven en `docker history`.

PASO 04

Construir, correr y comprobar

```
docker build -t mi-app:despues .
docker run --rm -d --name mi-app -p 3000:3000 mi-app:despues
node check-image.mjs http://localhost:3000 /logo.png
docker stop mi-app
```

Cambia `/logo.png` por un archivo que sí esté en tu `public`. Guarda `check-image.mjs` (Node 18 o posterior, sin dependencias):

```
// Usage: node check-image.mjs <base-url> <file-in-public>
// Example: node check-image.mjs http://localhost:3000 /logo.png
// Checks that the page, one /_next/static file and one public file return 200.
const base = process.argv[2] ?? "http://localhost:3000";
const publicFile = process.argv[3];
if (!publicFile) {
  console.error("Usage: node check-image.mjs <base-url> <file-in-public>");
  process.exit(2);
}
async function status(path) {
  try {
    const res = await fetch(new URL(path, base), { redirect: "manual" });
    return { code: res.status, body: await res.text() };
  } catch (err) {
    return { code: 0, body: "", error: err.cause?.code ?? err.message };
  }
}
const page = await status("/");
const asset = page.body.match(/\/_next\/static\/[^"'\s)]+/)?.[0];
const checks = [
  ["page /", page],
  [`static ${asset ?? "(none found in the HTML)"}`, asset ? await status(asset) : { code: 0 }],
  [`public ${publicFile}`, await status(publicFile)],
];
let failed = false;
for (const [name, r] of checks) {
  const ok = r.code === 200;
  if (!ok) failed = true;
  console.log(`${ok ? "OK  " : "FAIL"} ${r.code || r.error || "-"} ${name}`);
```

```
}
process.exit(failed ? 1 : 0);
```

Sale con código 1 si la página, el primer archivo de `/_next/static` del HTML o el de `public` no responden 200. Contra `server.js` sin los estáticos marcó `FAIL 404` en los dos archivos; después de copiarlos, `OK 200` en los tres.

PASO 05

Elegir la base y medir

ETIQUETA	LIBC	DESCARGA	CUÁNDO
`node:24.22.3-bookworm`	glibc	410 MB	Etapas de build que compilan módulos nativos
`node:24.22.3-bookworm-slim`	glibc	81 MB	Punto de partida para las tres etapas
`node:24.22.3-alpine`	musl	63 MB	Si el tamaño es crítico y tus deps nativas tienen build para musl

Tamaño comprimido de las capas, linux/amd64, según los manifiestos de Docker Hub del 19 de noviembre de 2026; en disco ocupan más. Para probar alpine: `docker build --build-arg NODE_VERSION=24.22.3-alpine -t mi-app:alpine .` y repite el paso 04.

Para medir, guarda tu Dockerfile anterior como `Dockerfile.naive` y construye las dos con la misma app:

```
docker build -f Dockerfile.naive -t mi-app:antes .
docker build -t mi-app:despues .
docker image ls mi-app
docker image history mi-app:despues
```

`SIZE` es el espacio acumulado de la imagen y sus padres. `docker image history` muestra cada capa con la instrucción que la creó: ahí ves qué capa pesa de más. Anota el `SIZE` de las dos con la base y la fecha.

AL TERMINAR

Checklist

- [ ] `output: "standalone"` en `next.config` y `.next/standalone/server.js` existe tras el build.
- [ ] `.dockerignore` en la raíz con `node_modules`, `.next`, `.git` y `.env*`.
- [ ] Tres etapas con la misma base en un `ARG`, versión exacta; lockfile copiado antes que el código.
- [ ] La etapa final copia `.next/standalone`, `.next/static` y `public`.
- [ ] `PORT` y `HOSTNAME=0.0.0.0` definidos; `CMD ["node", "server.js"]`.
- [ ] Corre como `USER node`, sin secretos en la imagen.
- [ ] `check-image.mjs` pasa y anotaste el `SIZE` de antes y después.