EcomLabs

GUÍA PASO A PASO

Next.js consume mucha memoria: mide y decide

Cómo leer la memoria de tu servidor Next.js, recorrer tu sitemap con y sin imágenes, decidir si images.unoptimized te conviene y volver atrás si no. Con el script de recorrido que usé, sin dependencias.

PARA	Next.js autoalojado con next start (Railway, VPS, Docker)
PROBADO CON	Node 22.22.3 · PowerShell 5.1 · Windows 11
ACTUALIZADA	2 de noviembre de 2026
DOCUMENTACIÓN	nextjs.org/docs/15/app/api-reference/components/image
ARTÍCULO	ecomlabs.dev/g/ram
AUTOR	Luis Morales · @luismoralesdev

Índice

Paso 01	Dónde leer la memoria
Paso 02	Recorre el sitemap
Paso 03	Decide si te conviene
Paso 04	El cambio y cómo volver atrás
Al terminar	Checklist

PASO 01

Dónde leer la memoria

Primero, la cifra de partida. En ecomlabs.dev (Next.js 15.5 en Railway) pasé de 0.99 GB de promedio a 0.29 GB tras recorrer las 276 páginas del sitemap; sin una medición antes y otra después, no sabrás si el cambio te sirvió.

- **En Railway:** haz clic en el servicio, en el canvas del proyecto, y abre la pestaña **Metrics**. Ahí está la gráfica de memoria, con hasta 30 días de datos. Con réplicas, la vista Sum las suma y la vista Replica las separa. Anota el promedio de una semana antes de tocar nada.
- **En Linux o macOS:** `ps -o rss= -p <pid>` imprime la memoria residente del proceso en KiB.
- **En Windows (PowerShell):** `(Get-Process -Id <pid>).WorkingSet64` da el working set en bytes. No es idéntico al RSS de Linux: compara siempre en la misma máquina.

Para leerla cada 2 segundos mientras corre el recorrido, en PowerShell basta con este bucle (contra mi servidor de prueba imprimió 62, 62 y 61 MB). El artículo trae `sample-rss.mjs`, que hace lo mismo con `ps` en Linux y macOS y guarda el pico:

```
$id = 95172
1..3 | ForEach-Object { "{0:N0} MB" -f ((Get-Process -Id $id).WorkingSet64 / 1MB); Start-Sleep 2 }
```

PASO 02

Recorre el sitemap

Guarda `crawl-sitemap.mjs`. Cambia el host de cada `<loc>` por el del sitemap (el sitemap de un build local lista tu dominio de producción), y con `--images` pide también cada `src` y `srcset` del mismo host, que es lo que pone a trabajar al optimizador:

```
// Requests every <loc> in a sitemap, on the sitemap's own host, and prints a summary.
// usage: node crawl-sitemap.mjs <sitemap-url> [--concurrency 1-4] [--images]
import { pathToFileURL } from "node:url";

const decode = (s) => s.replaceAll("&amp;", "&");

export function pageUrls(xml, sitemapUrl) {
  const host = new URL(sitemapUrl);
  // A local build's sitemap still lists the production domain: keep the path, swap the host.
  return [...xml.matchAll(/<loc>\s*([^<]+?)\s*<\/loc>/g)].map((m) => {
    const u = new URL(decode(m[1]), host);
    return new URL(u.pathname + u.search, host.origin).href;
  });
}

export function imageUrls(html, pageUrl) {
  const found = [];
  for (const [tag] of html.matchAll(/<img\b[^>]*>/gi)) {
    for (const [, name, value] of tag.matchAll(/\s(src|srcset)="([^"]*)"/gi)) {
```

```
      const list = name.toLowerCase() === "srcset" ? value.split(",").map((c) => c.trim().split(/\s+/)[0]) :
[value];
      for (const v of list) if (v && !v.startsWith("data:")) found.push(new URL(decode(v), pageUrl).href);
    }
  }
  // Same host only: the crawl measures your server, never someone else's CDN.
  return found.filter((u) => new URL(u).origin === new URL(pageUrl).origin);
}

export async function crawl(sitemapUrl, { concurrency = 1, images = false, log = () => {} } = {}) {
  const started = performance.now();
  const res = await fetch(sitemapUrl);
  if (!res.ok) throw new Error(`sitemap: HTTP ${res.status}`);
  const pages = pageUrls(await res.text(), sitemapUrl);
  if (pages.length === 0) throw new Error("no <loc> found (a sitemap index is not supported)");

  const statuses = {};
  const seen = new Set();
  let bytes = 0;
  const get = async (url) => {
    try {
      const r = await fetch(url, { signal: AbortSignal.timeout(30_000) });
      const body = Buffer.from(await r.arrayBuffer());
      bytes += body.length;
      statuses[r.status] = (statuses[r.status] ?? 0) + 1;
      if (r.status >= 400) log(`${r.status} ${url}`);
      return { r, body };
    } catch (err) {
      statuses.error = (statuses.error ?? 0) + 1;
      log(`error ${url}: ${err.message}`);
      return null;
    }
  };

  let next = 0;
  const worker = async () => {
    while (next < pages.length) {
      const url = pages[next++];
      const got = await get(url);
      if (!images || !got?.r.headers.get("content-type")?.includes("text/html")) continue;
      for (const img of imageUrls(got.body.toString("utf8"), url)) {
        if (seen.has(img)) continue;
        seen.add(img);
        await get(img);
      }
    }
  };
  await Promise.all(Array.from({ length: Math.min(concurrency, pages.length) }, worker));
  return { pages: pages.length, images: seen.size, statuses, bytes, ms: Math.round(performance.now() -
started) };
}
```

```
if (import.meta.url === pathToFileURL(process.argv[1]).href) {
  const args = process.argv.slice(2);
  const url = args.find((a) => /^https?:\/\//.test(a));
  const i = args.indexOf("--concurrency");
  const concurrency = Math.min(4, Math.max(1, Number(i >= 0 ? args[i + 1] : 1) || 1));
  if (!url) {
    console.error("usage: node crawl-sitemap.mjs <sitemap-url> [--concurrency 1-4] [--images]");
    process.exit(2);
  }
  const s = await crawl(url, { concurrency, images: args.includes("--images"), log: console.log });
  const codes = Object.entries(s.statuses).map(([code, n]) => `${code} x ${n}`).join(", ");
  console.log(`pages:     ${s.pages}`);
  console.log(`images:    ${s.images}`);
  console.log(`responses: ${codes}`);
  console.log(`received:  ${(s.bytes / 1024 / 1024).toFixed(1)} MB`);
  console.log(`time:      ${(s.ms / 1000).toFixed(1)} s (concurrency ${concurrency})`);
  process.exitCode = Object.keys(s.statuses).some((c) => c === "error" || Number(c) >= 400) ? 1 : 0;
}
```

Con el servidor en modo producción (`next build` y `next start`) y la memoria a la vista, corre `node crawl-sitemap.mjs http://localhost:3000/sitemap.xml`, anota la memoria, y repite con `--concurrency 2 --images`. Esta es la salida real de ese segundo comando contra mi servidor de prueba (13 URLs, una de ellas un 404 a propósito); sale con código 1 si hubo un error o un 4xx/5xx:

```
404 http://localhost:4321/gone
pages:     13
images:    10
responses: 200 x 22, 404 x 1
received:  0.4 MB
time:      0.1 s (concurrency 2)
```

> **CUIDADO**
>
> Si la memoria sube igual en el recorrido sin `--images`, el optimizador no es (solo) tu problema: apagarlo no lo va a arreglar. Y una cifra local no predice producción: tu computadora no reproduce el asignador de memoria de glibc que usa Linux en Railway.

PASO 03

Decide si te conviene

TU CASO	¿UNOPTIMIZED?
Todas las imágenes en `/public`, ya en WebP o AVIF y del tamaño final	Sí
Fotos grandes en `/public`	No, o conviértelas antes
Imágenes remotas (CMS, bucket)	No, salvo que ya lleguen del tamaño final
Imágenes que suben tus usuarios	No
Necesitas `srcset` para fotos grandes en celular	No, o escribe el `srcset` a mano
Un servicio externo ya optimiza tus imágenes	Mejor un loader de `next/image`
La memoria crece igual sin `--images`	Todavía no: busca otra causa

Si solo algunas imágenes no ganan nada (SVG, GIF animados, menos de 1 KB), usa la prop `unoptimized` en esas y deja el optimizador para el resto.

PASO 04

El cambio y cómo volver atrás

En `next.config.ts`, deja `images: { unoptimized: true }` y borra lo que solo servía al optimizador (`formats`, `minimumCacheTTL`, `experimental.imgOptConcurrency`). Junto a la opción, escribe la regla y cuándo deshacerla; en mi repositorio dice: `shortcut: no srcset, so phones get the desktop-sized file; turn the optimizer back on if /public gains large photos.` Despliega, recorre producción, lee Metrics y, una semana después, compara promedio con promedio.

Para volver atrás: quita `unoptimized: true` (o revierte el commit) y despliega. En Next.js 15.5, sharp viene como dependencia opcional de `next`: no hay que instalar nada. En Next.js 15 el `minimumCacheTTL` por defecto es de 60 segundos y `/public` se sirve con `max-age=0`, así que cada variante se recodifica al minuto: vuelve a poner un valor largo. En Next.js 16 el valor por defecto ya es de 4 horas.

AL TERMINAR

Checklist

- [ ] Anotaste el promedio de memoria de una semana en Railway Metrics.
- [ ] Todas tus imágenes salen de `/public`; no hay remotas ni subidas por usuarios.
- [ ] Cada imagen ya está en WebP o AVIF, del tamaño de su visualización más grande.
- [ ] Recorriste el sitemap sin y con `--images` mientras medías la memoria.
- [ ] Después del deploy recorriste producción y leíste Metrics.
- [ ] Dejaste un comentario que dice cuándo volver a activar el optimizador.