UCAB UNIVERSIDAD CATÓLICA ANDRÉS BELLO

Tópicos Especiales de Programación

Manejadores de Paquetes

¿Qué es un manejador de paquetes?

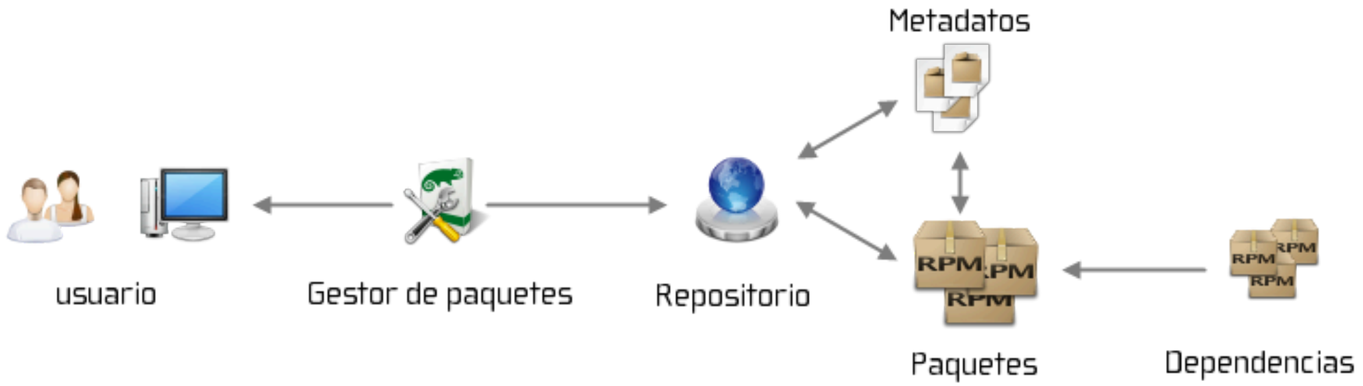
Un **paquete** (*package*) es código que alguien más escribió y que pueden reutilizar en sus aplicaciones, básicamente, una librería empaquetada y lista para usar.

Cuando descargan un paquete y lo usan en su proyecto, este se convierte en una dependencia: su proyecto depende de él para funcionar.

Un **gestor de paquetes** (*package manager*) es una herramienta que:

- Se conecta a un repositorio de paquetes en internet
- Descarga paquetes por nombre y los instala en su máquina
- Maneja actualizaciones y diferentes versiones
- Automatiza todo el proceso de forma reproducible

Las dependencias pueden volverse complicadas, y debemos usar una herramienta que gestione ese caos.

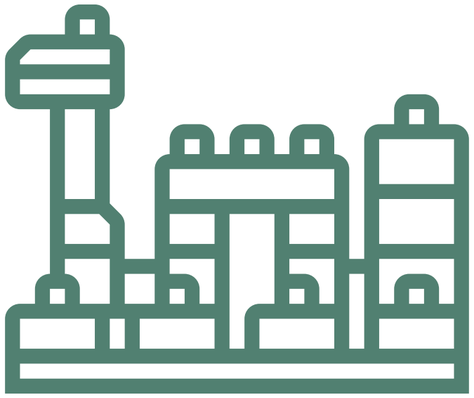


Sets de LEGO

Imaginen que entran a una casa repleta de legos, sets coloridos exhibidos en todos lados. Hay una flor en la repisa del espejo del baño, un dragón cerca de las escaleras, un set de una serie de televisión cerca de la mesa de la cocina y una escuela de magia cerca de la TV.

En la mayoría de los casos, una gran creación es una colección de sets más pequeños.

Imaginen que el enorme castillo de la escuela de magia es su aplicación y, para que esté completa, necesitan agregar un montón de extras como una estación de tren, un árbol mágico o un fénix. Cada uno de ellos es una entidad separada que pueden mover como deseen.



¿Qué tiene esto que ver?

Si el castillo de la escuela de magia es su **aplicación**, entonces cada uno de los extras sería un **módulo de Node o un paquete**.

Digamos que el castillo necesita un árbol mágico. Podrían diseñarlo ustedes mismos, pero es mucho más fácil conseguir el set completo, diseñado y empaquetado en una caja con instrucciones. En esta analogía, un set listo para armar es un "paquete" y las instrucciones son el archivo "package.json".

Un módulo de Node, o un paquete, es un fragmento de código que ayuda a agregar alguna funcionalidad a una aplicación. El archivo package.json nos dice quién creó el paquete, cuándo se creó, qué otras "dependencias" necesita para funcionar, y así sucesivamente.

La documentación oficial de npm hace una distinción técnica entre "módulos de Node" y "paquetes". Sin embargo, en la práctica se usan indistintamente.

Aquí entran los gestores de paquetes

Ahora, si solo tienen uno o dos sets de LEGO en su casa, gestionar un inventario **no sería tan difícil**. Pero,

¿Qué pasa si son coleccionistas de toda la vida, las piezas se rompen a menudo, salen nuevos sets constantemente y el esquema de colores disponible cambia muy rápido?

Idealmente encontrarían un sistema para llevar el registro de todas las piezas que poseen, las que necesitan reemplazar, las que ya no necesitan y las que tienen que comprar.

Es por esto que en el software existe la necesidad de los llamados "**gestores de paquetes**" (*package managers*). Un gestor de paquetes es una herramienta que te ayuda a llevar el registro de todas las dependencias de tu aplicación de una manera consistente.

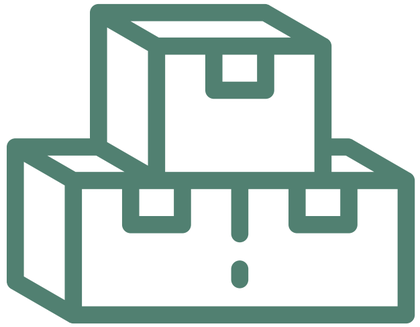
Y algo super importante: automatiza las tareas de instalación, actualización, configuración y eliminación de dependencias. Se asegura de que todos los paquetes que tu aplicación necesita estén instalados, en la versión correcta o actualizados.

Así se ve un package.json

```
{
  "name": "topicos-especiales",
  "version": "1.0.0",
  "description": "Ejercicios y ejemplos de Tópicos Especiales de Programación (TypeScript)",
  "main": "dist/index.js",
  "types": "dist/index.d.ts",
  "scripts": {
    "build": "tsc -p tsconfig.json",
    "dev": "ts-node src/index.ts",
    "start": "node dist/index.js",
    "lint": "eslint \"src/**/*.ts\"",
    "test": "jest",
    "format": "prettier --write \"src/**/*.{ts,md}\""
  },
  "repository": {
    "type": "git",
    "url": "git+https://github.com/usuario/topicos-especiales.git"
  },
```

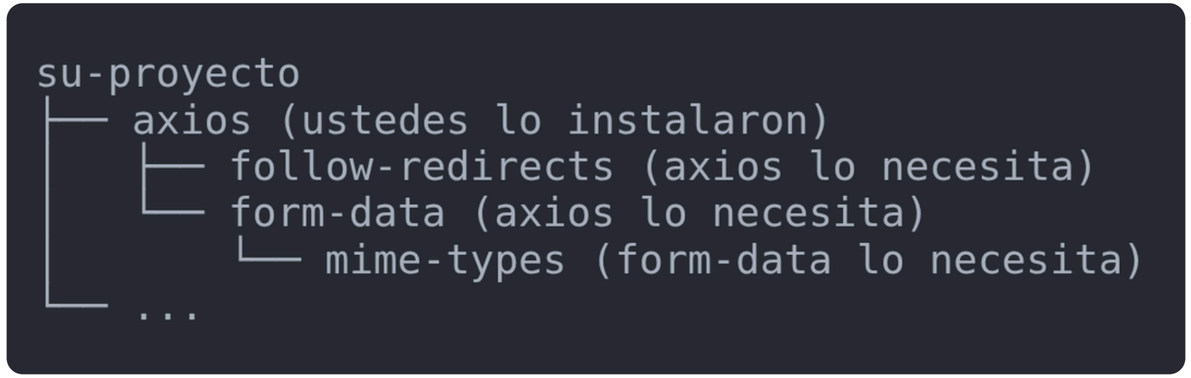
```
"keywords": ["typescript", "educacion", "programacion"],
  "author": "Prof. Italo Visconti",
  "license": "MIT",
  "bugs": {
    "url": "https://github.com/usuario/topicos-especiales/issues"
  },
  "homepage": "https://github.com/usuario/topicos-especiales#readme",
  "engines": {
    "node": ">=18.0.0"
  },
  "dependencies": {
    "axios": "^1.4.0"
  },
  "devDependencies": {
    "typescript": "^5.1.3",
    "@types/node": "^18.16.0",
    "ts-node": "^10.9.1",
    "eslint": "^8.50.0",
    "jest": "^29.6.4",
    "ts-jest": "^29.1.0",
    "prettier": "^2.8.8"
  }
}
```

¿Por qué necesitamos gestores de paquetes?



1. Dependencias transitivas (el árbol de dependencias)

Cuando armamos un proyecto, solemos usar muchos paquetes, y cada paquete puede tener sus propias dependencias. **Por ejemplo**, si usamos **axios** para hacer solicitudes HTTP, este paquete podría depender internamente de otros paquetes para funcionar. El gestor de paquetes se encarga de descargar e instalar todas estas **dependencias transitivas** automáticamente.



Sin un gestor de paquetes, tendrían que rastrear e instalar cada una de estas dependencias manualmente.

¿Se han dado cuenta?

Cuando instalan un paquete y luego revisan **node_modules**, aparecen un montón de carpetas nuevas que no esperaban. Esas son las dependencias transitivas.

2. Control de versiones (para evitar el caos)

El gestor de paquetes mantiene un registro de las versiones específicas de cada paquete. Esto es crucial para evitar conflictos entre versiones incompatibles.

Los gestores no simplemente instalan la versión más nueva; respetan las versiones especificadas en los archivos de configuración (package.json, package-lock.json, yarn.lock).

¿Qué son los "breaking changes"?

Un *breaking change* es cuando una actualización de un paquete introduce cambios incompatibles con versiones anteriores. Por ejemplo:

- Una función cambia de nombre o parámetros
- Se elimina una funcionalidad
- Cambia el formato de los datos de retorno

Si su proyecto depende de una versión específica, el gestor se asegurará de instalar esa versión (y no la más reciente) evitando errores inesperados.

Algo que van a experimentar (si no lo han hecho ya): dos librerías que ustedes usan **dependen de versiones diferentes de una misma dependencia interna**.

```
su-proyecto
├── axios → necesita follow-redirects@1.15.0
└── otra-libreria → necesita follow-redirects@1.14.0  Conflicto
```

Resolver esto manualmente sería un dolor de cabeza. El gestor de paquetes maneja estas situaciones automáticamente, ya sea instalando versiones compatibles o aislando dependencias cuando es necesario.

3. Actualizaciones seguras

Si quieren actualizar un paquete a una versión más reciente, el gestor puede hacerlo de manera segura, verificando compatibilidad con las demás dependencias.

```
# Ver qué paquetes tienen actualizaciones disponibles
npm outdated

# Actualizar un paquete específico
npm update axios

# Actualizar todo (con cuidado)
npm update
```

Siempre revisa el changelog de un paquete antes de actualizar a una versión major (ej: de 1.x a 2.x). Los *breaking changes* suelen ocurrir ahí.

4. Scripts y automatización

Los gestores de paquetes permiten definir scripts personalizados para automatizar tareas comunes:

```
"scripts": {
  "dev": "ts-node src/index.ts",
  "build": "tsc",
  "test": "jest",
  "lint": "eslint src/",
  "format": "prettier --write src/"
}
```

Ejecutas con **npm run dev**, **npm run test**, **etc**. Esto estandariza cómo se trabaja en el proyecto, cualquier persona nueva puede ejecutar los mismos comandos.

5. Separación dev vs producción

Pueden distinguir entre:

- **dependencies**: paquetes necesarios para que tu app funcione en producción
- **devDependencies**: paquetes solo para desarrollo (TypeScript, linters, test runners)

```
# Instalar en dependencies (producción)
npm install axios

# Instalar en devDependencies (solo desarrollo)
npm install -D typescript eslint jest
```

En producción, puedes instalar solo lo necesario con **npm install --production**, reduciendo el tamaño del deploy.

6. Reproducibilidad del entorno

Uno de los problemas más comunes en desarrollo es el clásico: "en mi máquina funciona"

Esto ocurre cuando dos desarrolladores tienen versiones ligeramente diferentes de las dependencias. El gestor de paquetes resuelve esto mediante:

1. **Archivo de lock** (package-lock.json, yarn.lock): registra las versiones exactas de todo lo instalado
2. **Comando determinista**: npm ci instala exactamente lo que dice el lock file, sin modificarlo
3. **Entornos consistentes**: CI/CD, producción y desarrollo usan las mismas versiones

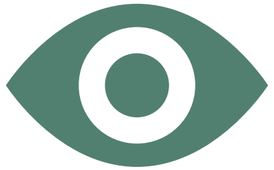
```
# En desarrollo (puede actualizar lock file)
npm install
# o simplemente
npm i

# En CI/CD o producción (respeta lock file estrictamente)
npm ci
```

Archivos de lock: package-lock.json y yarn.lock

Además del **package.json**, los gestores generan un archivo de "lock" que registra las versiones exactas de todas las dependencias instaladas (incluyendo las transitivas).

Archivo	Gestor
package-lock.json	npm
yarn.lock	yarn
pnpm-lock.yaml	pnpm



Siempre incluye el archivo lock en tu repositorio git. Esto garantiza que todos los desarrolladores (y el servidor de producción) instalen exactamente las mismas versiones.

Comandos esenciales

Acción	npm	yarn
Inicializar proyecto	npm init -y	yarn init -y
Instalar dependencias	npm install	yarn
Agregar paquete	npm install axios	yarn add axios
Agregar paquete dev	npm install -D typescript	yarn add -D typescript
Eliminar paquete	npm uninstall axios	yarn remove axios
Ejecutar script	npm run dev	yarn dev
Ver paquetes desactualizados	npm outdated	yarn outdated
Actualizar paquete	npm update axios	yarn upgrade axios
Instalar paquete de forma global	npm install -g typescript	yarn global add typescript
Ejecutar paquete sin instalar	npx create-react-app mi-app	yarn dlx create-react-app mi-app

Estos conceptos aplican a cualquier ecosistema

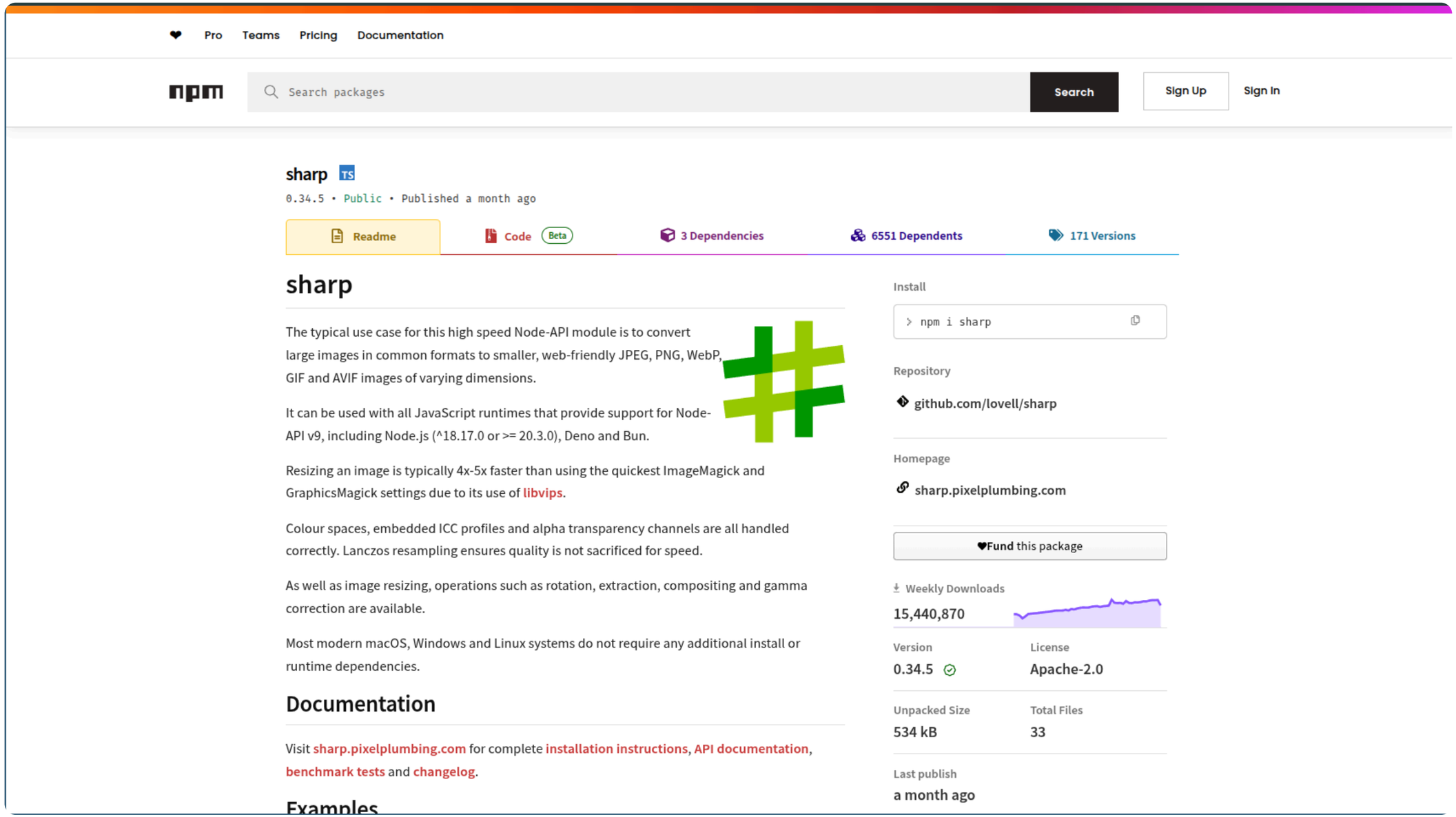
Lenguaje/Sistema	Gestor de paquetes	Archivo de config
JavaScript/Node	npm, yarn, pnpm	package.json
Python	pip, poetry, conda	requirements.txt, pyproject.toml
Ruby	gem, bundler	Gemfile
PHP	composer	composer.json
Rust	cargo	Cargo.toml
Go	go modules	go.mod
Linux (Debian)	apt	—
Linux (Arch)	pacman	—
macOS	brew	—
Windows	choco, winget	—

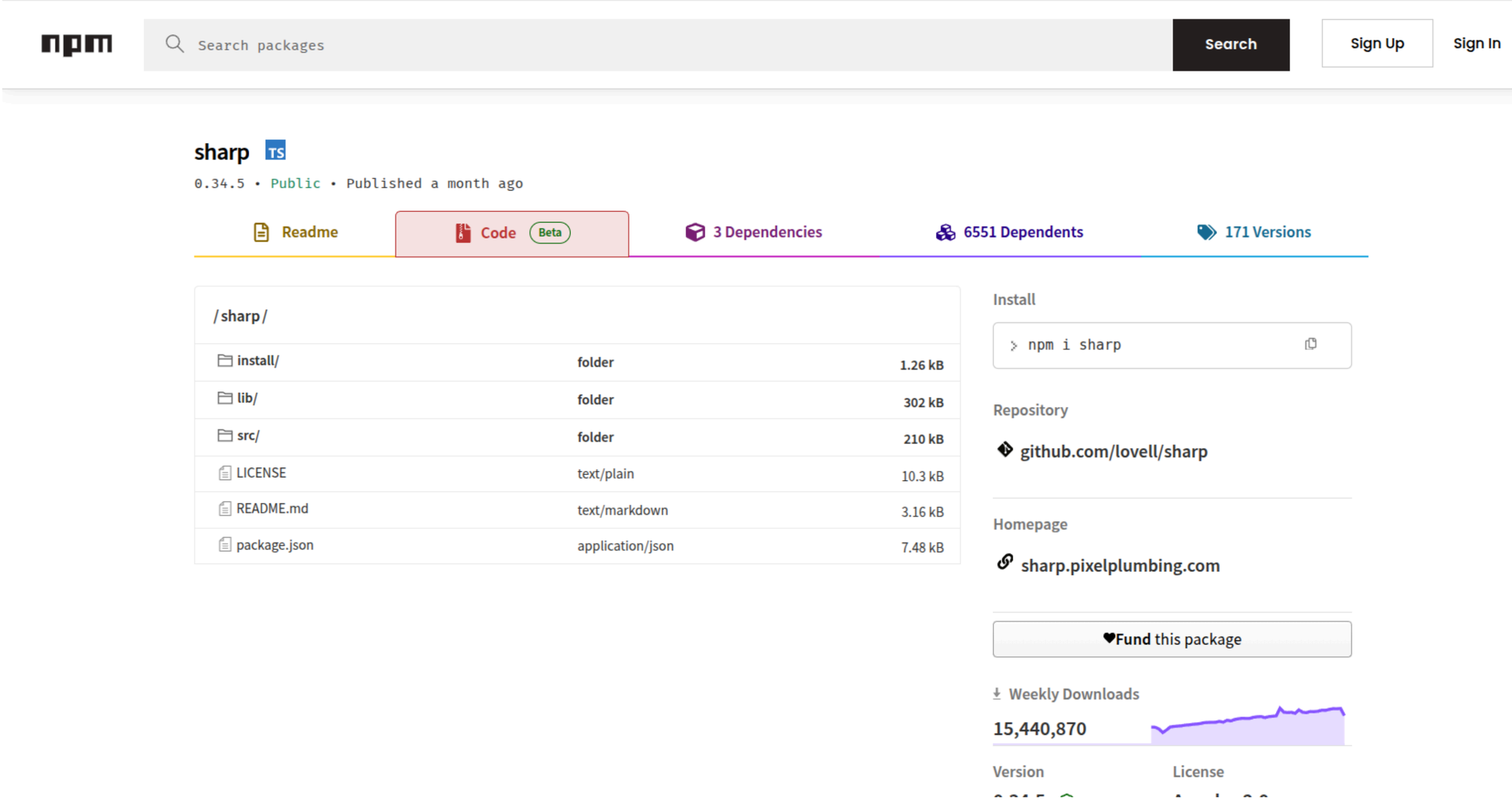
¿Y dónde se alojan todos estos paquetes?

Cada ecosistema tiene su **registro de paquetes** (*package registry*): un servidor centralizado donde los desarrolladores publican sus paquetes y desde donde tú los descargas.

El registro más grande del mundo para node es **npmjs.com**, con más de **2 millones de paquetes**. Cuando ejecutas **npm install axios**, npm consulta este registro, descarga el paquete y lo coloca en tu carpeta **node_modules**.

Otro detalle importante es que en los registros pueden ver datos como: cantidad de descargas, versiones disponibles, documentación, el código fuente, issues reportados, etc.


Pro
Teams
Pricing
Documentation
npm
Search packages
Search
Sign Up
Sign In
sharp
TS
0.34.5 • Public • Published a month ago
Readme
Code
Beta
3 Dependencies
6551 Dependents
171 Versions
sharp
The typical use case for this high speed Node-API module is to convert large images in common formats to smaller, web-friendly JPEG, PNG, WebP, GIF and AVIF images of varying dimensions.
It can be used with all JavaScript runtimes that provide support for Node-API v9, including Node.js (^18.17.0 or >= 20.3.0), Deno and Bun.
Resizing an image is typically 4x-5x faster than using the quickest ImageMagick and GraphicsMagick settings due to its use of libvips.
Colour spaces, embedded ICC profiles and alpha transparency channels are all handled correctly. Lanczos resampling ensures quality is not sacrificed for speed.
As well as image resizing, operations such as rotation, extraction, compositing and gamma correction are available.
Most modern macOS, Windows and Linux systems do not require any additional install or runtime dependencies.
Documentation
Visit sharp.pixelplumbing.com for complete installation instructions, API documentation, benchmark tests and changelog.
Examples
Install
npm i sharp
Repository
github.com/lovell/sharp
Homepage
sharp.pixelplumbing.com
Fund this package
Weekly Downloads
15,440,870
Version
0.34.5
License
Apache-2.0
Unpacked Size
534 kB
Total Files
33
Last publish
a month ago


npm
Search packages
Search
Sign Up
Sign In
sharp
TS
0.34.5 • Public • Published a month ago
Readme
Code Beta
3 Dependencies
6551 Dependents
171 Versions
/sharp/
install/ folder 1.26 kB
lib/ folder 302 kB
src/ folder 210 kB
LICENSE text/plain 10.3 kB
README.md text/markdown 3.16 kB
package.json application/json 7.48 kB
Install
npm i sharp
Repository
github.com/lovell/sharp
Homepage
sharp.pixelplumbing.com
Fund this package
Weekly Downloads
15,440,870
Version
License

sharp

0.34.5 • Public • Published a month ago

Readme | Code Beta | 3 Dependencies | 6551 Dependents | 171 Versions

Dependencies (3)

@img/colour detect-libc semver

Dev Dependencies (18)

@biomejs/biome @cpplint/cli @emnapi/runtime @img/sharp-libvips-dev
@img/sharp-libvips-dev-wasm32 @img/sharp-libvips-win32-arm64
@img/sharp-libvips-win32-ia32 @img/sharp-libvips-win32-x64 @types/node
emnapi exif-reader extract-zip icc jsdoc-to-markdown node-addon-api
node-gyp tar-fs tsd

Install

```
> npm i sharp
```

Repository

github.com/lovell/sharp

Homepage

sharp.pixelplumbing.com

❤Fund this package

Weekly Downloads

15,440,870

Version License

Otros registros populares

Ecosistema	Registro	URL
JavaScript	npm	npmjs.com
Python	PyPI	pypi.org
Ruby	RubyGems	rubygems.org
PHP	Packagist	packagist.org
Rust	crates.io	crates.io
Go	Go Modules	pkg.go.dev

Registros privados

Las empresas grandes suelen tener **registros privados** para paquetes internos que no quieren hacer públicos. Herramientas como **Verdaccio**, **GitHub Packages**, **GitLab Package Registry** o **Artifactory** permiten hospedar tus propios paquetes.

¿Qué quiere decir esto?

Que ustedes pueden crear un repositorio público de GitHub con su propio paquete, y luego usar npm para instalar ese paquete directamente desde GitHub.

¿Por qué hay tantos gestores de paquetes para JavaScript?

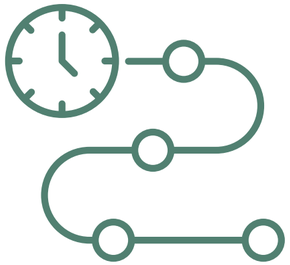
En otros lenguajes suele haber un solo gestor dominante (pip en Python, cargo en Rust, composer en PHP). Pero en JavaScript tenemos npm, yarn, pnpm, bun... ¿por qué?

2010 - npm nace

- npm (Node Package Manager) fue creado junto con Node.js
- Era lento, no tenía lock files, y node_modules podía volverse gigantesco
- Pero funcionaba y se convirtió en el estándar

2016 - Yarn aparece

- Facebook, Google y otros crearon Yarn para resolver los problemas de npm
- Introdujo yarn.lock (instalaciones deterministas)
- Era mucho más rápido que npm en ese momento
- Instalación en paralelo, mejor caché



2017 - npm reacciona

- npm 5 agregó package-lock.json
- Mejoró drásticamente la velocidad
- Cerró la brecha con Yarn

2020 - pnpm gana tracción

- Usa hard links y symlinks para evitar duplicar paquetes
- node_modules ocupa mucho menos espacio
- Más rápido que npm y yarn en muchos casos
- Estructura más estricta (evita acceder a dependencias no declaradas)

2022 - Bun entra al juego

- Runtime de JavaScript escrito en Zig (no en JS)
- Incluye gestor de paquetes integrado
- Extremadamente rápido (instalaciones en segundos)
- Aún en desarrollo activo

Fin