

# THE NODE.JS HANDBOOK

---

FLAVIO COPES

# Tabla de contenido

## [El manual de Node.js](#)

### [Introducción](#)

[Introducción al nodo](#)

[Una breve historia de Nodo](#)

[Cómo instalar Nodo](#)

[¿Cuánto JavaScript necesitas saber para usar Node?](#)

[Diferencias entre Node y el Navegador](#)

[v8](#)

### [Lo esencial](#)

[Ejecute los scripts de Node.js desde la línea de comandos](#)

[Cómo salir de un programa Node.js](#)

[Cómo leer variables de entorno](#)

[Opciones de alojamiento de nodos](#)

### [Línea de comando](#)

[Utilice el nodo REPL](#)

[Pasar argumentos desde la línea de comando](#)

[Salida a la línea de comando](#)

[Aceptar entrada desde la línea de comando](#)

### [Módulos de nodo y npm](#)

[Exponga la funcionalidad de un archivo de nodo mediante exportaciones](#)

[npm](#)

[¿Dónde instala npm los paquetes?](#)

[Cómo usar o ejecutar un paquete instalado usando npm](#)

[El archivo package.json](#)

[El archivo package-lock.json](#)

[Encuentre la versión instalada de un paquete npm](#)

[Cómo instalar una versión anterior de un paquete npm](#)

[Cómo actualizar todas las dependencias de Node a su última versión](#)

[Reglas de versionado semántico](#)

[Desinstalación de paquetes npm](#)

[Paquetes globales o locales](#)

---

[dependencias npm y devDependencies](#)

---

[npx](#)

---

[Trabajar con el bucle de eventos](#)

---

[El bucle de eventos](#)

---

[siguienteTick](#)

---

[establecerInmediato](#)

---

[Temporizadores](#)

---

[Programación asíncrona](#)

---

[devoluciones de llamada](#)

---

[promesas](#)

---

[asíncrono/espera](#)

---

[El emisor de eventos del nodo](#)

---

[Redes](#)

---

[HTTP](#)

---

[Cómo funcionan las solicitudes HTTP](#)

---

[Construir un servidor HTTP](#)

---

[Realización de solicitudes HTTP](#)

---

[Axios](#)

---

[Websockets](#)

---

[HTTPS, conexiones seguras](#)

---

[Sistema de archivos](#)

---

[Descriptores de archivo](#)

---

[Estadísticas de archivo](#)

---

[Rutas de archivo](#)

---

[Lectura de archivos](#)

---

[Escribir archivos](#)

---

[Trabajando con carpetas](#)

---

[Algunos módulos básicos esenciales](#)

---

[El módulo fs](#)

---

[El módulo de ruta](#)

---

[El módulo del sistema operativo](#)

---

[El módulo de eventos](#)

---

[El módulo http](#)

---

[Misceláneas](#)

---

[Corrientes](#)

---

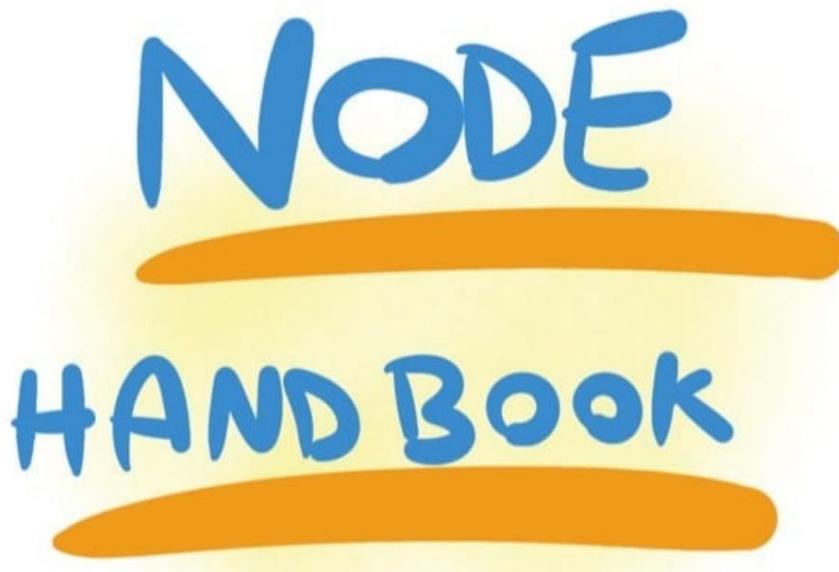
[Trabajando con MySQL](#)

---

[Diferencia entre desarrollo y producción.](#)

---

# El manual de Node.js



El Manual del Nodo sigue la regla del 80/20: aprende en el 20% del tiempo el 80% de un tema.

Creo que este enfoque ofrece una visión completa. Este libro no trata de cubrirlo todo.  
bajo el sol relacionado con Node. Si crees que debería incluirse algún tema en concreto, dímelo.

Puedes contactarme en Twitter [@flaviocopes](#).

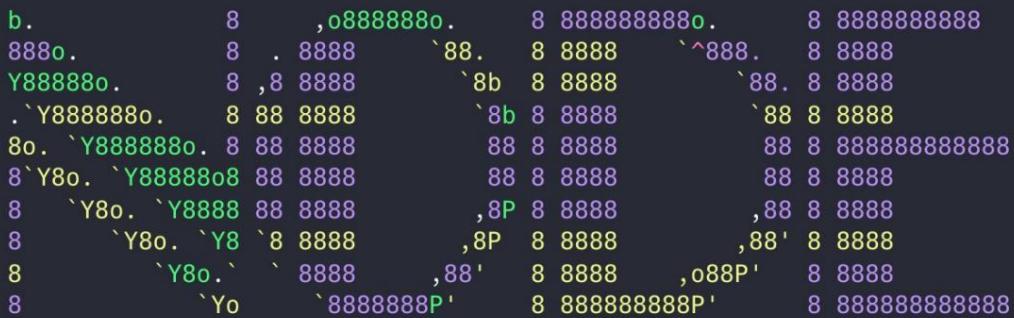
Espero que el contenido de este libro lo ayude a lograr lo que desea: **aprender los conceptos básicos de Node.js.**

Este libro está escrito por Flavio. Publico **tutoriales de desarrollo web** todos los días en mi sitio web [flaviocopes.com](http://flaviocopes.com).

¡Disfrutar!

# Introducción al nodo

Esta publicación es una guía de introducción a Node.js, el entorno de tiempo de ejecución de JavaScript del lado del servidor. Node.js se basa en el motor de JavaScript Google Chrome V8 y se usa principalmente para crear servidores web, pero no se limita a eso.



```
b.          8      ,o888888o.      8 8888888888o.      8 88888888888
888o.      8 . 8888     `88. 8 8888     `^888. 8 8888
Y888888o.  8 ,8 8888     `8b 8 8888     `88. 8 8888
.`Y8888888o. 8 88 8888     `8b 8 8888     `88 8 8888
8o. `Y8888888o. 8 88 8888     88 8 8888     88 8 888888888888
8`Y8o. `Y88888888 88 8888     88 8 8888     88 8 8888
8   `Y8o. `Y8 8888     ,8P 8 8888     ,88 8 8888
8       `Y8o. `Y8 8888     ,8P 8 8888     ,88' 8 8888
8       `Y8o. ` 8888     ,88' 8 8888     ,o88P' 8 8888
8           `Yo     `88888888P' 8 8888888888P' 8 888888888888
```

- Visión general
- Las mejores características de Node.js
  - Rápido
  - Simple
  - JavaScript
  - V8
  - Plataforma asíncrona Una gran
  - cantidad de bibliotecas Un ejemplo
- de aplicación Node.js
- Frameworks y herramientas de Node.js

## Visión general

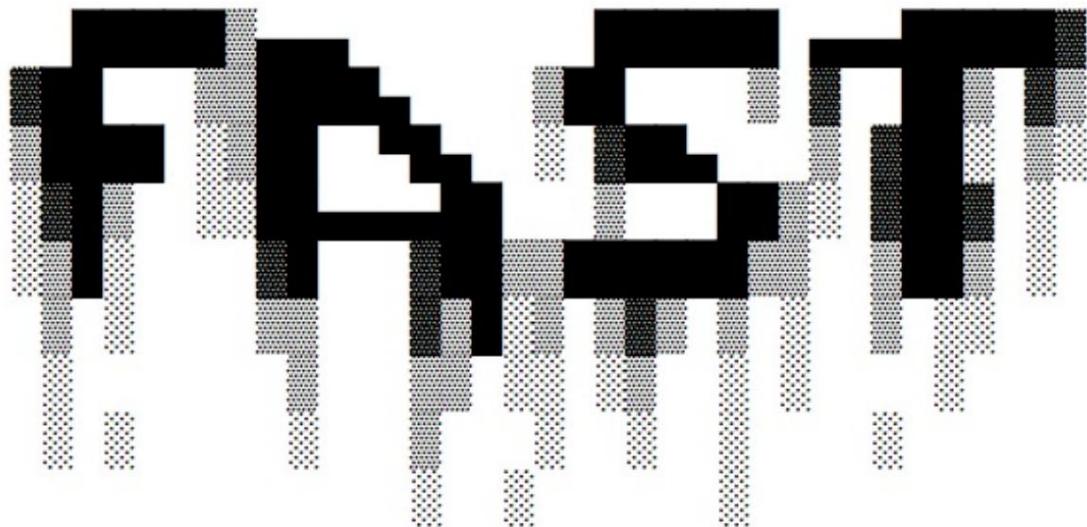
Node.js es un **entorno de tiempo de ejecución para JavaScript** que se ejecuta en el **servidor**.

Node.js es de código abierto, multiplataforma y, desde su introducción en 2009, se hizo muy popular y ahora juega un papel importante en la escena del desarrollo web. Si las estrellas de GitHub son un factor indicador de popularidad, tener más de 46000 estrellas significa ser muy popular.

Node.js se basa en el motor JavaScript de Google Chrome V8 y se utiliza principalmente para crear servidores web, pero no se limita a eso.

## Las mejores características de Node.js

### Rápido



Uno de los principales puntos de venta de Node.js es la **velocidad**. El código JavaScript que se ejecuta en Node.js (según el punto de referencia) puede ser el doble de rápido que los lenguajes compilados como C o Java, y mucho más rápido que los lenguajes interpretados como Python o Ruby, debido a su paradigma sin bloqueo.

### Simple

Node.js es simple. Extremadamente simple, en realidad.

### JavaScript

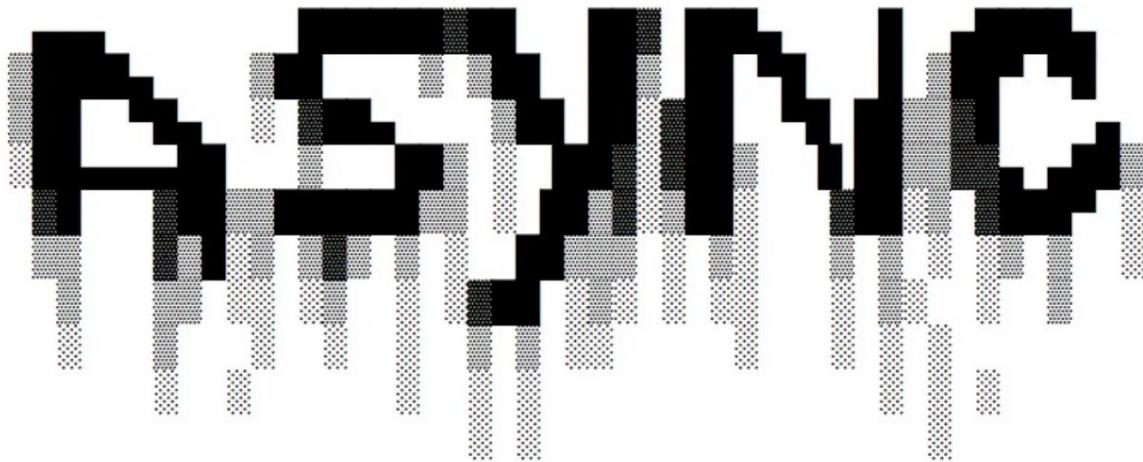
Node.js ejecuta código JavaScript. Esto significa que millones de desarrolladores frontend que ya usan JavaScript en el navegador pueden ejecutar el código del lado del servidor y el código del lado frontend usando el mismo lenguaje de programación sin la necesidad de aprender una herramienta completamente diferente.

Los paradigmas son todos iguales, y en Node.js el nuevo [ECMAScript](#) los estándares se pueden usar primero, ya que no tiene que esperar a que todos sus usuarios actualicen sus navegadores; usted decide qué versión de ECMAScript usar cambiando la versión de Node.js.

### V8

Ejecutándose en el [motor JavaScript de Google V8](#), que es de código abierto, Node.js puede aprovechar el trabajo de miles de ingenieros que hicieron (y seguirán haciendo) el tiempo de ejecución de Chrome JavaScript increíblemente rápido.

## Plataforma asíncrona



En los lenguajes de programación tradicionales (C, Java, Python, PHP) todas las instrucciones se bloquean de forma predeterminada a menos que "opte" explícitamente para realizar operaciones asíncronas. Si realiza una solicitud de red para leer algo de JSON, la ejecución de ese subproceso en particular se bloquea hasta que la respuesta esté lista.

**JavaScript permite crear código asíncrono y sin bloqueo de una manera muy sencilla**, mediante el uso de un **solo hilo, funciones de devolución de llamada y programación dirigida por eventos**. Cada vez que ocurre una operación costosa, pasamos una función de devolución de llamada que se llamará una vez que podamos continuar con el procesamiento. No estamos esperando a que termine para continuar con el resto del programa.

Dicho mecanismo deriva del navegador. No podemos esperar hasta que se cargue algo de una solicitud AJAX antes de poder interceptar los eventos de clic en la página. **Todo debe suceder en tiempo real** para proporcionar una buena experiencia al usuario.

Si ha creado un controlador onclick para una página web, ya ha utilizado técnicas de programación asíncrona con detectores de eventos.

Esto permite que Node.js maneje miles de conexiones simultáneas con un solo servidor sin presentar la carga de administrar la concurrencia de subprocesos, lo que sería una fuente importante de errores.

Node proporciona primitivas de E/S sin bloqueo y, en general, las bibliotecas en Node.js se escriben utilizando paradigmas sin bloqueo, lo que hace que un comportamiento de bloqueo sea una excepción en lugar de lo normal.

Cuando Node.js necesita realizar una operación de E/S, como leer de la red, acceder a una base de datos o al sistema de archivos, en lugar de bloquear el subproceso, Node.js simplemente reanudará las operaciones cuando regrese la respuesta, en lugar de desperdiciar ciclos de CPU esperando.

## Una gran cantidad de bibliotecas.

npm con su estructura simple ayudó al ecosistema de node.js a proliferar y ahora el npm registro alberga casi 500.000 paquetes de código abierto que puede usar libremente.

# Una aplicación de ejemplo de Node.js

El ejemplo más común Hello World de Node.js es un servidor web:

```
constante http = require('http')

const nombre de host = '127.0.0.1'
puerto constante = 3000

const servidor = http.createServer((req, res) => {
  res.statusCode = 200
  res.setHeader('Content-Type', 'text/plain') res.end('Hello
  World\n')
})

server.listen(puerto, nombre de host, () => {
  console.log(' Servidor ejecutándose en http://${hostname}:${port}/' ) })
```

Para ejecutar este fragmento, guárdelo como un archivo server.js y ejecute node server.js en su terminal.

Este código primero incluye el módulo http de Node.js.

Node.js tiene una biblioteca estándar increíble , incluyendo un soporte de primera clase para la creación de redes.

El método createServer() de http crea un nuevo servidor HTTP y lo devuelve.

El servidor está configurado para escuchar en el puerto y el nombre de host especificados. Cuando el servidor está listo, se llama a la función de devolución de llamada, en este caso nos informa que el servidor se está ejecutando.

Cada vez que se recibe una nueva solicitud, se llama al evento de solicitud , proporcionando dos objetos: una solicitud (un objeto http.IncomingMessage ) y una respuesta (un objeto http.ServerResponse ).

Esos 2 objetos son esenciales para manejar la llamada HTTP.

El primero proporciona los detalles de la solicitud. En este ejemplo simple, esto no se usa, pero puede acceder a los encabezados de la solicitud y a los datos de la solicitud.

El segundo se utiliza para devolver datos a la persona que llama.

En este caso con

```
res.statusCode = 200
```

establecemos la propiedad statusCode en 200, para indicar una respuesta exitosa.

Establecemos el encabezado Content-Type:

```
res.setHeader('Tipo de contenido', 'texto/simple')
```

y terminamos de cerrar la respuesta, agregando el contenido como argumento a end() :

```
res.end('Hola Mundo\n')
```

## Frameworks y herramientas de Node.js

Node.js es una plataforma de bajo nivel, y para hacer las cosas más fáciles e interesantes para los desarrolladores, se crearon miles de bibliotecas sobre Node.js.

Muchos de los establecidos con el tiempo como opciones populares. Aquí hay una lista no exhaustiva de los que considero muy relevantes y que vale la pena aprender:

- [Expresar](#), una de las formas más simples pero poderosas de crear un servidor web. Su enfoque minimalista, sin opiniones, centrado en las características principales de un servidor, es clave para su éxito.
- [Meteorito](#), un marco de trabajo de pila completa increíblemente poderoso, que lo impulsa con un enfoque isomorfo para crear aplicaciones con JavaScript, compartiendo código en el cliente y el servidor. Una vez que fue una herramienta lista para usar que proporcionó todo, ahora se integra con las librerías frontend React, [Vue](#) y Angular. También se puede utilizar para crear aplicaciones móviles. [koa](#), construido por el mismo equipo detrás de Express, pretende ser aún más simple y más pequeño, basándose en años de conocimiento. El nuevo proyecto nació de la necesidad de crear cambios incompatibles sin interrumpir la comunidad existente.
- [siguiente.js](#), un marco para renderizar [React](#) renderizado del lado del servidor aplicaciones
- [Micro](#), un servidor muy ligero para crear microservicios HTTP asíncronos.
- [Socket.io](#), un motor de comunicación en tiempo real para crear aplicaciones de red.

# Una breve historia de Nodo

## Una mirada retrospectiva a la historia de Node.js desde 2009 hasta la actualidad

Lo creas o no, Node.js tiene solo 9 años.

En comparación, JavaScript tiene 23 años y la web tal como la conocemos (después de la introducción de Mosaic) tiene 25 años.

9 años es muy poco tiempo para una tecnología, pero Node.js parece haber sido alrededor para siempre.

He tenido el placer de trabajar con Node desde los primeros días cuando solo tenía 2 años y, a pesar de la poca información disponible, ya se podía sentir que era algo enorme.

En esta publicación, quiero dibujar el panorama general de Node en su historia, para poner las cosas en perspectiva.

- [Un poquito de historia](#)
- [2009](#)
- [2010](#)
- [2011](#)
- [2012](#)
- [2013](#)
- [2014](#)
- [2015](#)
- [2016](#)
- [2017](#)
- [2018](#)

## Un poquito de historia

JavaScript es un lenguaje de programación que se creó en Netscape como una herramienta de secuencias de comandos para manipular páginas web dentro de su navegador, [Netscape Navigator](#).

Parte del modelo comercial de Netscape era vender servidores web, que incluían un entorno llamado *Netscape LiveWire*, que podía crear páginas dinámicas utilizando JavaScript del lado del servidor. Por lo tanto, Node.js no introdujo la idea de JavaScript del lado del servidor, pero es tan antiguo como JavaScript, pero en ese momento no tuvo éxito.

Un factor clave que condujo al surgimiento de Node.js fue el tiempo. JavaScript desde hace algunos años comenzó a ser considerado un lenguaje serio, gracias a las aplicaciones "Web 2.0" que le mostraron al mundo cómo podría ser una experiencia moderna en la web (piense en Google Maps o

Gmail).

La barra de rendimiento de los motores de JavaScript aumentó considerablemente gracias a la batalla de la competencia de los navegadores, que sigue siendo fuerte. Los equipos de desarrollo detrás de cada uno de los principales navegadores trabajan arduamente todos los días para brindarnos un mejor rendimiento, lo cual es una gran victoria para JavaScript como plataforma. V8, el motor que usa Node.js bajo el capó, es uno de ellos y, en particular, es el motor Chrome JS.

Pero, por supuesto, Node.js no es popular solo por pura suerte o tiempo. Introdujo muchas ideas innovadoras sobre cómo programar en JavaScript en el servidor.

## 2009

Nace Node.js La primera forma de [npm](#) es creado

## 2010

[Expresar](#) nace [Socket.io](#) ha nacido

## 2011

npm llega a 1.0 Las grandes empresas comienzan a adoptar Node: LinkedIn, Uber [Hapi](#) ha nacido

## 2012

La adopción continúa muy rápidamente

## 2013

Primera gran plataforma de blogs que usa Node: Ghost [Koa](#) ha nacido

## 2014

Gran drama: [IO.js](#) es una bifurcación importante de Node.js, con el objetivo de introducir compatibilidad con ES6 y mover más rápido

## 2015

La Fundación [Node.js](#) nace IO.js se fusiona nuevamente con Node.js npm presenta módulos privados Nodo 4 (no se lanzaron versiones 1, 2, 3 anteriormente)

## 2016

El [incidente de la almohadilla](#) izquierda nace Nodo 6

## 2017

npm se centra más en la seguridad Nodo 8 HTTP/2 [V8](#) presenta Node en su conjunto de pruebas, lo que convierte oficialmente a Node en un objetivo para el motor JS, además de las descargas de 3 mil millones de npm de Chrome cada semana

## 2018

[Módulos ES](#) del nodo 10 Soporte experimental .mjs

# Cómo instalar Nodo

## Cómo puede instalar Node.js en su sistema: un administrador de paquetes, el instalador del sitio web oficial o nvm

Node.js se puede instalar de diferentes maneras. Esta publicación destaca los más comunes y los convenientes.

Los paquetes oficiales para todas las plataformas principales están disponibles en <https://nodejs.org/en/download/>.

Una forma muy conveniente de instalar Node.js es a través de un administrador de paquetes. En este caso, cada sistema operativo tiene el suyo.

En macOS, [Homebrew](#) es el estándar de facto y, una vez instalado, permite instalar Node.js muy fácilmente ejecutando este comando en la CLI:

nodo de instalación de cerveza

Otros administradores de paquetes para Linux y Windows se enumeran en <https://nodejs.org/en/download/package-manager/>

nvm es una forma popular de ejecutar Node. Le permite cambiar fácilmente la versión de Node e instalar nuevas versiones para probar y revertir fácilmente si algo se rompe, por ejemplo.

También es muy útil probar tu código con versiones antiguas de Node.

Consulte <https://github.com/creationix/nvm> para obtener más información sobre esta opción.

Mi sugerencia es usar el instalador oficial si recién está comenzando y aún no usa Homebrew; de lo contrario, Homebrew es mi solución favorita.

En cualquier caso, cuando Node esté instalado, tendrá acceso al programa ejecutable del nodo en el línea de comando.

# ¿Cuánto JavaScript necesitas saber para usar Node?

## Si recién está comenzando con JavaScript, ¿qué tan profundo necesita saber el idioma?

Como principiante, es difícil llegar a un punto en el que tenga suficiente confianza en su programación. habilidades.

Mientras aprende a codificar, es posible que también se sienta confundido sobre dónde termina JavaScript y dónde comienza Node.js, y viceversa.

Le recomendaría tener una buena comprensión de los principales conceptos de JavaScript antes de sumergirse en Node.js:

- Estructura léxica
- Expresiones
- Tipos
- Variables
- Funciones
- este
- Funciones de flecha
- Bucles
- Bucles y alcance
- arreglos
- Literales de plantilla
- punto y coma
- Modo estricto
- ECMAScript 6, 2016, 2017

Con esos conceptos en mente, está bien encaminado para convertirse en un desarrollador de JavaScript competente, tanto en el navegador como en Node.js.

Los siguientes conceptos también son clave para entender la programación asíncrona, que es una parte fundamental de Node.js:

- Programación asíncrona y devoluciones de llamada
- Temporizadores
- promesas
- Asíncrono y en espera
- Cierres
- El bucle de eventos

Afortunadamente, escribí un libro electrónico gratuito que explica todos esos temas y se llama **Fundamentos de JavaScript**. Es el recurso más compacto que encontrarás para aprender todo esto.

Puede encontrar el libro electrónico en la parte inferior de esta página: <https://flaviocopes.com/javascript/>.

# Diferencias entre Node y el Navegador

## En qué se diferencia la escritura de una aplicación JavaScript en Node.js de la programación para la Web dentro del navegador

Tanto el navegador como Node utilizan JavaScript como lenguaje de programación.

La creación de aplicaciones que se ejecutan en el navegador es algo completamente diferente a la creación de una aplicación Node.js.

A pesar de que siempre es JavaScript, existen algunas diferencias clave que hacen que la experiencia sea radicalmente diferente.

Como desarrollador frontend que escribe aplicaciones Node, tiene una gran ventaja: el lenguaje sigue siendo lo mismo.

Tiene una gran oportunidad porque sabemos lo difícil que es aprender un lenguaje de programación de forma completa y profunda, y al usar el mismo lenguaje para realizar todo su trabajo en la web, tanto en el cliente como en el servidor, se encuentra en un posición única de ventaja.

Lo que cambia es el ecosistema.

En el navegador, la mayoría de las veces lo que estás haciendo es interactuar con el [DOM](#), u otras [API de plataforma web](#) como cookies. Esos no existen en Node, por supuesto. no tienes el `documento` , `ventana` y todos los demás objetos proporcionados por el navegador.

Y en el navegador, no tenemos todas las buenas API que proporciona Node.js a través de sus módulos, como la funcionalidad de acceso al sistema de archivos.

Otra gran diferencia es que en Node.js tú controlas el entorno. A menos que esté creando una aplicación de código abierto que cualquiera pueda implementar en cualquier lugar, sabe en qué versión de Node ejecutará la aplicación. En comparación con el entorno del navegador, donde no puede darse el lujo de elegir qué navegador usarán sus visitantes, esto es muy conveniente.

Esto significa que puede escribir todos los [ES6-7-8-9](#) modernos JavaScript compatible con su versión de Node.

Dado que JavaScript se mueve tan rápido, pero los navegadores pueden ser un poco lentos y los usuarios un poco lentos para actualizar, a veces en la web, se ve obligado a usar versiones anteriores de JavaScript/ECMAScript.

YPuede usar Babel para transformar su código para que sea compatible con ES5 antes de enviarlo al navegador, pero en Node, no necesitará eso.

Otra diferencia es que Node usa el [sistema de módulos CommonJS](#), mientras que en el navegador estamos empezando a ver los [Módulos ES](#) estándar que se está implementando.

En la práctica, esto significa que, por el momento, utiliza require() en Node e importa en el navegador.

## V8

**V8 es el nombre del motor de JavaScript que impulsa Google Chrome. Es lo que toma nuestro JavaScript y lo ejecuta mientras navegamos con Chrome. V8 proporciona el entorno de tiempo de ejecución en el que se ejecuta JavaScript. El navegador proporciona el DOM y las demás API de la plataforma web.**



V8 es el nombre del motor de JavaScript que impulsa Google Chrome. Es lo que toma nuestro JavaScript y lo ejecuta mientras navegamos con Chrome.

V8 proporciona el entorno de tiempo de ejecución en el que se ejecuta JavaScript. El navegador proporciona el [DOM](#) y las demás [API](#) de la plataforma web.

Lo bueno es que el motor de JavaScript es independiente del navegador en el que está alojado.

Esta característica clave permitió el surgimiento de [Node.js](#). Se eligió V8 por ser el motor elegido por Node.js en 2009 y, a medida que explotó la popularidad de Node.js, V8 se convirtió en el motor que ahora impulsa una cantidad increíble de código del lado del servidor escrito en JavaScript.

El ecosistema de Node.js es enorme y, gracias a él, V8 también potencia las aplicaciones de escritorio, con proyectos como [electrón](#).

## Otros motores JS

Otros navegadores tienen su propio motor de JavaScript:

- Firefox tiene [Mono Araña](#)
- Safari tiene [JavaScriptCore](#) (también llamado Nitro)
- Edge tiene [Chakra](#)

y muchos otros existen también.

Todos esos motores implementan el estándar ECMA ES-262, también llamado [ECMAScript](#), el estándar utilizado por JavaScript.

## La búsqueda del rendimiento

V8 está escrito en C++ y se mejora continuamente. Es portátil y se ejecuta en Mac, Windows, Linux y varios otros sistemas.

En esta introducción de V8, ignoraré los detalles de implementación de V8: se pueden encontrar en sitios más autorizados (por ejemplo, el sitio oficial de V8) y cambian con el tiempo, a menudo radicalmente.

V8 siempre está evolucionando, al igual que los otros motores de JavaScript, para acelerar la Web y el ecosistema Node.js.

En la web, existe una carrera por el rendimiento desde hace años, y nosotros (como usuarios y desarrolladores) nos beneficiamos mucho de esta competencia porque obtenemos máquinas más rápidas y optimizadas año tras año.

## Compilacion

JavaScript generalmente se considera un lenguaje interpretado, pero los motores de JavaScript modernos ya no solo interpretan JavaScript, sino que lo compilan.

Esto sucede desde 2009 cuando se agregó el compilador de JavaScript SpiderMonkey a Firefox 3.5, y todos siguieron esta idea.

JavaScript es compilado internamente por V8 con compilación **justo a tiempo** (JIT) para acelerar el ejecución.

Esto puede parecer contrario a la intuición, pero desde la introducción de Google Maps en 2004, JavaScript ha evolucionado de un lenguaje que generalmente ejecutaba unas pocas docenas de líneas de código a aplicaciones completas con miles a cientos de miles de líneas ejecutándose en el navegador.

Nuestras aplicaciones ahora pueden ejecutarse durante horas dentro de un navegador, en lugar de ser solo unas pocas reglas de validación de formularios o simples scripts.

En este *nuevo mundo*, compilar JavaScript tiene mucho sentido porque, si bien puede llevar un poco más de tiempo tener el JavaScript *listo*, una vez hecho, tendrá mucho más rendimiento que el código puramente interpretado.

# Ejecute los scripts de Node.js desde la línea de comandos

## Cómo ejecutar cualquier script de Node.js desde la CLI

La forma habitual de ejecutar un programa de Node es llamar al comando node disponible globalmente (una vez que instale Node) y pasar el nombre del archivo que desea ejecutar.

Si su archivo de aplicación de Nodo principal está en app.js , puede llamarlo escribiendo

```
aplicación de nodo.js
```

# Cómo salir de un programa Node.js

## Aprenda cómo terminar una aplicación Node.js de la mejor manera posible

Hay varias formas de finalizar una aplicación Node.js.

Al ejecutar un programa en la consola, puede cerrarlo con ctrl-C. La discusión aquí se está , pero lo que quiero cerrando mediante programación.

Comencemos con el más drástico y veamos por qué es mejor que *no* lo uses.

El módulo principal del proceso proporciona un método útil que le permite salir mediante programación de un programa Node.js: `process.exit()` .

Cuando Node.js ejecuta esta línea, el proceso se ve obligado a terminar inmediatamente.

Esto significa que cualquier devolución de llamada que esté pendiente, cualquier solicitud de red que aún se envíe, cualquier acceso al sistema de archivos o procesos que escriban en `stdout` o `stderr` , todo se terminará de manera irregular de inmediato.

Si esto está bien para usted, puede pasar un número entero que le indica al sistema operativo el código de salida:

```
proceso.salir(1)
```

De forma predeterminada, el código , lo que significa éxito. Diferentes códigos de salida tienen diferentes de salida tiene el significado de 0 , que es posible que desee utilizar en su propio sistema para que el programa se comunique con otros programas.

Puede leer más sobre los códigos de salida en [https://nodejs.org/api/process.html#process\\_exit\\_codes](https://nodejs.org/api/process.html#process_exit_codes)

También puede establecer la propiedad `process.exitCode` :

```
proceso.exitCode = 1
```

y cuando el programa finalice más tarde, Node devolverá ese código de salida.

Un programa saldrá correctamente cuando todo el procesamiento haya terminado.

Muchas veces con Node arrancamos servidores, como este servidor HTTP:

```
const express = require('express') const app
= express()

app.get('/', (requerido, res) => {
  res.send('¡Hola!')
})
```

```
app.listen(3000, () => console.log('Servidor listo'))
```

Este programa nunca va a terminar. Si llama a `process.exit()`, se cancelará cualquier solicitud actualmente pendiente o en ejecución. Esto *no es agradable*.

En este caso, debe enviar el comando una señal SIGTERM y manejar eso con el controlador de señal de proceso:

Nota: el proceso no requiere un "requerimiento", está disponible automáticamente.

```
const express = require('express')

const aplicación = express()

app.get('/', (requerido, res) => {
    res.send('¡Hola!')
})

const server = app.listen(3000, () => console.log('Servidor listo'))

proceso.on('SIGTERM', () =>
    { servidor.cerrar() =>
        { consola.log('Proceso terminado') } })
)
```

¿Qué son las señales? Las señales son un sistema de intercomunicación POSIX: una notificación enviada a un proceso para avisarle de un evento ocurrido.

SIGKILL son las señales que le dicen a un proceso que termine inmediatamente, e idealmente actuarían como `proceso.salir()`.

SIGTERM son las señales que le dicen a un proceso que termine correctamente. Es la señal que se envía de gerentes de procesos como upstart o supervisord y muchos otros.

Puedes enviar esta señal desde dentro del programa, en otra función:

```
proceso.matar(proceso.pid, 'SIGTERM')
```

O desde otro programa en ejecución de Node.js, o cualquier otra aplicación que se ejecute en su sistema que conozca el PID del proceso que desea finalizar.

# Cómo leer variables de entorno

## Aprenda a leer y hacer uso de variables de entorno en un programa Node.js

El módulo central del proceso de Node proporciona la propiedad env que alberga todas las variables de entorno que se establecieron en el momento en que se inició el proceso.

Aquí hay un ejemplo que accede a la variable de entorno NODE\_ENV, que se establece en desarrollo por defecto.

Nota: el proceso no requiere un "requerimiento", está disponible automáticamente.

```
proceso.env.NODE_ENV // "desarrollo"
```

Establecerlo en "producción" antes de que se ejecute el script le dirá a Node que se trata de una producción ambiente.

De la misma manera, puede acceder a cualquier variable de entorno personalizada que establezca.

## Opciones de alojamiento de nodos

**Una aplicación de Node.js se puede alojar en muchos lugares, según sus necesidades. Esta es una lista de todas las diversas opciones que tiene a su disposición**

Aquí hay una lista no exhaustiva de las opciones que puede explorar cuando desee implementar su aplicación y hacerla accesible públicamente.

Enumeraré las opciones desde las más simples y restringidas hasta las más complejas y poderosas.

- [La opción más simple: túnel local](#)
- [Despliegues de configuración cero](#)
  - [Falla](#)
  - [Código abierto](#)
- [sin servidor](#)
- [PAAS](#)
  - [Ahora](#)
  - [Nanocaja](#)
  - [Heroku](#)
  - [microsoft azure](#)
  - [Plataforma en la nube de Google](#)
- [Servidor Virtual Privado](#)
- [Metal básico](#)

## La opción más simple: túnel local

Incluso si tiene una IP dinámica o está bajo un NAT, puede implementar su aplicación y atender las solicitudes directamente desde su computadora usando un túnel local.

Esta opción es adecuada para algunas pruebas rápidas, demostraciones de un producto o compartir una aplicación con un grupo muy pequeño de personas.

Una muy buena herramienta para esto, disponible en todas las plataformas, es [ngrok](#).

Utilizándolo, puede simplemente escribir ngrok PORT y el PORT que desea está expuesto a Internet. Obtendrá un dominio ngrok.io, pero con una suscripción paga puede obtener una URL personalizada, así como más opciones de seguridad (recuerde que está abriendo su máquina a la Internet pública).

Otro servicio que puede usar es <https://github.com/localtunnel/localtunnel>

## Despliegues de configuración cero

## Falla

[Falla](#) es un patio de recreo y una forma de crear sus aplicaciones más rápido que nunca, y verlas en vivo en su propio subdominio `glitch.com`. Actualmente no puede tener un dominio personalizado y existen algunas [restricciones](#) en su lugar, pero es realmente genial hacer un prototipo. Parece divertido (y esto es una ventaja), y no es un entorno simplificado: obtienes todo el poder de Node.js, un CDN, almacenamiento seguro para credenciales, importación/exportación de GitHub y mucho más.

Proporcionado por la empresa detrás de FogBugz y Trello (y co-creadores de Stack Overflow).

Lo uso mucho para propósitos de demostración.

## Código abierto

[Código abierto](#) es una plataforma y una comunidad increíbles. Puede crear un proyecto con varios archivos e implementarlo con un dominio personalizado.

## sin servidor

Una forma de publicar sus aplicaciones y no tener ningún servidor que administrar es Serverless. Serverless es un paradigma en el que publica sus aplicaciones como **funciones** y responden en un punto final de red (también llamado FAAS - Funciones como servicio).

Soluciones muy populares son

- [Marco sin servidor](#)
- [Biblioteca estándar](#)

Ambos proporcionan una capa de abstracción para publicar en AWS Lambda y otras soluciones FAAS basadas en Azure o la oferta de Google Cloud.

## PAAS

PAAS significa Plataforma como servicio. Estas plataformas eliminan muchas cosas de las que debería preocuparse al implementar su aplicación.

## Ahora

Zeit es una opción interesante. Simplemente escriba ahora en su terminal y se encargará de implementar su aplicación. Hay una versión gratuita con limitaciones y la versión de pago es más potente. Simplemente olvida que hay un servidor, simplemente implementa la aplicación.

## Nanocaja

[Nanocaja](#)

## Heroku

Heroku es una plataforma increíble.

Este es un excelente artículo sobre [cómo comenzar con Node.js en Heroku](#).

## microsoft azure

Azure es la oferta de Microsoft Cloud.

Vea cómo [crear una aplicación web Node.js en Azure](#).

## Plataforma en la nube de Google

Google Cloud es una estructura increíble para sus aplicaciones.

Tienen una buena [sección de documentación de Node.js](#)

## Servidor Virtual Privado

En esta sección encontrará los sospechosos habituales, ordenados de más fácil de usar a menos fácil de usar:

- [oceano digital](#)
- [Linodo](#)
- [servicios web de amazon](#), en particular menciono Amazon Elastic Beanstalk ya que abstrae un poco la complejidad de AWS.

Dado que proporcionan una máquina Linux vacía en la que puede trabajar, no hay un tutorial específico para éstos.

Hay muchas más opciones en la categoría VPS, esas son solo las que usé y me gustaría recomendar.

## Metal básico

Otra solución es obtener un servidor bare metal, instalar una distribución de Linux, conectarlo a Internet (o alquilar uno mensualmente, como se puede hacer con [Vultr Bare Metal](#)). Servicio)



# Utilice el nodo REPL

**REPL significa Read-Evaluate-Print-Loop, y es una excelente manera de explorar las funciones de Node de manera rápida**

El comando de nodo es el que usamos para ejecutar nuestros scripts de Node.js:

```
script de nodo.js
```

Si omitimos el nombre del archivo, lo usamos en modo REPL:

```
nodo
```

Si lo pruebas ahora en tu terminal, esto es lo que sucede:

```
ÿ nodo  
>
```

el comando se queda en modo inactivo y espera a que ingresemos algo.

Sugerencia: si no está seguro de cómo abrir su terminal, busque en Google "Cómo abrir una terminal".

El REPL está esperando que ingresemos algún código JavaScript, para ser más precisos.

Comience simple e ingrese

```
> consola.log('prueba')  
prueba  
indefinido  
>
```

El primer valor, prueba , es el resultado que le dijimos a la consola que imprimiera, luego obtenemos undefined, que es el valor de retorno de ejecutar `console.log()` .

Ahora podemos ingresar una nueva línea de JavaScript.

## Usa la pestaña para autocompletar

Lo bueno de REPL es que es interactivo.

A medida que escribe su código, si presiona la tecla de tabulación , REPL intentará autocompletar lo que escribió para que coincida con una variable que ya definió o una predefinida.

## Explorando objetos de JavaScript

Intente ingresar el nombre de una clase de JavaScript, como Número , agrega un punto y presiona tabulador .

El REPL imprimirá todas las propiedades y métodos a los que puede acceder en esa clase:

```
node-handbook: node
> Number.
Number.__defineGetter__    Number.__defineSetter__    Number.__lookupGetter__
Number.__lookupSetter__   Number.__proto__          Number.constructor
Number.hasOwnProperty      Number.isPrototypeOf   Number.propertyIsEnumerable
Number.toLocaleString     Number.toString          Number.valueOf

Number.apply               Number.arguments        Number.bind
Number.call                Number.caller          Number.length

Number.EPSILON             Number.MAX_SAFE_INTEGER Number.MAX_VALUE
Number.MIN_SAFE_INTEGER    Number.MIN_VALUE       Number.NEGATIVE_INFINITY
Number.NaN                 Number.POSITIVE_INFINITY Number.isFinite
Number.isInteger            Number.isNaN           Number.isSafeInteger
Number.parseFloat          Number.parseInt        Number.prototype
```

## Explora objetos globales

Puede inspeccionar los globales a los que tiene acceso escribiendo global. y presionando tabulador :

```

node-handbook: node
|> global.
global.__defineGetter__          global.__defineSetter__
global.__lookupGetter__          global.__lookupSetter__
global.__proto__                 global.constructor
global.hasOwnProperty            global.isPrototypeOf
global.propertyIsEnumerable      global.toLocaleString
global.toString                  global.valueOf

global.Array                      global.ArrayBuffer
global.Boolean                     global.Buffer
global.DTRACE_HTTP_CLIENT_REQUEST global.DTRACE_HTTP_CLIENT_RESPONSE
global.DTRACE_HTTP_SERVER_REQUEST  global.DTRACE_HTTP_SERVER_RESPONSE
global.DTRACE_NET_SERVER_CONNECTION global.DTRACE_NET_STREAM_END
global.DataView                   global.Date
global.Error                      global.EvalError
global.Float32Array                global.Float64Array
global.Function                   global.GLOBAL
global.Infinity                   global.Int16Array
global.Int32Array                 global.Int8Array
global.Intl                       global.JSON
global.Map                        global.Math
global.NaN                         global.Number
global.Object                     global.Promise
global.Proxy                      global.RangeError
global.ReferenceError             global.Reflect
global.RegExp                     global.Set
global.String                     global.Symbol
global.SyntaxError                global.TypeError
global.URIError                   global.Uint16Array
global.Uint32Array                 global.Uint8Array
global.Uint8ClampedArray          global.WeakMap
global.WeakSet                    global.WebAssembly

```

## los \_ variable especial

Si después de algún código escribes `_`, eso imprimirá el resultado de la última operación.

## Comandos de puntos

El REPL tiene algunos comandos especiales, todos comenzando con un punto `.`

Están

- `.help` : muestra la ayuda de los comandos de punto
- `.editor` : habilita al editor más, para escribir código JavaScript multilínea con facilidad. una vez que estés en este modo, ingrese `ctrl-D` para ejecutar el código que escribió.
- `.break` : al ingresar una expresión de varias líneas, ingresar el comando `.break` abortará entrada adicional. Igual que presionar `ctrl-C`.
- `.clear` : restablece el contexto REPL a un objeto vacío y borra cualquier expresión de varias líneas

que se está ingresando actualmente.

- .load : carga un archivo JavaScript, relativo al directorio de trabajo actual .save : guarda todo lo que
- ingresó en la sesión REPL en un archivo (especifique el nombre del archivo) .exit : existe el repl (lo mismo que presionar
- ctrl-C dos veces)

El REPL sabe cuándo está escribiendo una declaración de varias líneas sin necesidad de invocar

.editor \_

Por ejemplo, si comienza a escribir una iteración como esta:

```
[1, 2, 3].forEach(num => {
```

y presionas enter , el REPL irá a una nueva línea que comienza con 3 puntos, lo que indica que puede ahora continúa trabajando en ese bloque.

```
... consola.log(num) ... })
```

Si escribe .break al final de una línea, el modo multilínea se detendrá y la declaración no ser ejecutado.

# Pasar argumentos desde la línea de comando

## Cómo aceptar argumentos en un programa Node.js pasado desde la línea de comando

Puede pasar cualquier cantidad de argumentos al invocar una aplicación Node.js usando

```
aplicación de nodo.js
```

Los argumentos pueden ser independientes o tener una clave y un valor.

Por ejemplo:

```
nodo app.js flavio
```

O

```
nodo app.js nombre=flavio
```

Esto cambia la forma en que recuperará este valor en el código del nodo.

La forma de recuperarlo es utilizando el objeto de proceso integrado en Node.

Expone una propiedad argv , que es una matriz que contiene todos los argumentos de invocación de la línea de comandos.

El primer argumento es la ruta completa del comando de nodo .

El segundo elemento es la ruta completa del archivo que se está ejecutando.

Todos los argumentos adicionales están presentes desde la tercera posición en adelante.

Puede iterar sobre todos los argumentos (incluida la ruta del nodo y la ruta del archivo) usando un bucle:

```
proceso.argv.forEach((valor, índice) => { consola.log(`$ {índice}: ${valor}`) })
```

Puede obtener solo los argumentos adicionales creando una nueva matriz que excluya los primeros 2 parámetros:

```
const argumentos = proceso.argv.slice(2)
```

Si tiene un argumento sin un nombre de índice, así:

```
nodo app.js flavio
```

puedes acceder usando

```
const argumentos = proceso.argv.slice(2)  
argumentos[0]
```

En este caso:

```
nodo app.js nombre=flavio
```

argumentos[0] es nombre=flavio , y necesitas analizarlo. La mejor manera de hacerlo es usando el biblioteca [minimalista](#) , que ayuda a lidiar con los argumentos:

```
const args = require('minimist')(process.argv.slice(2)) args['name'] //flavio
```

# Salida a la línea de comando

Cómo imprimir en la consola de línea de comandos usando Node, desde el archivo `console.log` básico hasta escenarios más complejos

- Salida básica usando el módulo de consola
- Borrar la consola
- Elementos de conteo
- Imprimir el seguimiento de la pila
- Calcular el tiempo dedicado
- salida estándar y estándar
- Colorea la salida
- Crear una barra de progreso

## Salida básica usando el módulo de consola

Node proporciona un [módulo de consola](#) que proporciona toneladas de formas muy útiles para interactuar con el línea de comando.

Es básicamente lo mismo que el objeto de la consola que encuentra en el navegador.

El método más básico y más utilizado es `console.log()`, que imprime la cadena que le pasas a la consola

Si pasa un objeto, lo representará como una cadena.

Puede pasar múltiples variables a `console.log`, por ejemplo:

```
constante x = 'X'  
const y = 'Y'  
consola.log(x, y)
```

y Node imprimirá ambos.

También podemos formatear frases bonitas pasando variables y un especificador de formato.

Por ejemplo:

```
console.log('Mi %s tiene %d años', 'gato', 2)
```

- `%s` da formato a una variable como una cadena
- `%d` o `%i` da formato a una variable como un número entero

- `%f` da formato a una variable como un número de coma flotante
- `%O` utilizado para imprimir una representación de objeto

Ejemplo:

```
consola.log("%O", Número)
```

## Borrar la consola

`console.clear()` borra la consola (el comportamiento puede depender de la consola utilizada)

## Elementos de conteo

`console.count()` es un método útil.

Toma este código:

```
constante x = 1
constante y = 2
constante z = 3
consola.cuenta(
  'El valor de x es ' + x + ' y ha sido revisado... ¿cuántas veces?'
)
consola.cuenta(
  'El valor de x es ' + x + ' y ha sido revisado... ¿cuántas veces?'
)
console.count( 'El
  valor de y es ' + y + ' y ha sido revisado... ¿cuántas veces?'
)
```

Lo que sucede es que `count` contará el número de veces que se imprime una cadena e imprimirá el cuenta al lado:

Solo puedes contar manzanas y naranjas:

```
const naranjas = ['naranja', 'naranja'] const
manzanas = ['solo una manzana']
naranjas.paraCada(fruta => { consola.cuenta(fruta) })
manzanas.paraCada(fruta => {
  consola.contar(fruta) })
```

## Imprimir el seguimiento de la pila

Puede haber casos en los que sea útil imprimir el seguimiento de la pila de llamadas de una función, tal vez para responder a la pregunta *¿cómo llegó a esa parte del código?*

Puedes hacerlo usando `console.trace()`:

```
const función2 = () => consola.trace() const función1
= () => función2() función1 ()
```

Esto imprimirá el seguimiento de la pila. Esto es lo que se imprime si pruebo esto en el Nodo REPL:

```
Rastro
en función2 (reemplazo:1:33)
en función1 (reemplazo:1:25)
en remplazo:1:1 en
ContextifyScript.Script.runInThisContext (vm.js:44:33) en REPLServer.defaultEval
(repl.js:239: 29) en el límite (domain.js:301:14) en REPLServer.runBound
[como evaluación] (domain.js:314:12) en REPLServer.onLine (repl.js:440:10)
en emitOne (events.js: 120:20) en REPLServer.emit (eventos.js:210:7)
```

## Calcular el tiempo dedicado

Puede calcular fácilmente cuánto tiempo tarda en ejecutarse una función, usando `time()` y `timeEnd()`

```
const hacerAlgo = () => console.log('test') const
medirHacerAlgo = () => { console.time('hacerAlgo()') //hacer
algo y medir el tiempo que lleva hacerAlgo() console.timeEnd
('hacer algo()')
}
} medirHaciendoAlgo()
```

## salida estándar y estándar

Como vimos, `console.log` es excelente para imprimir mensajes en la consola. Esto es lo que se llama la salida estándar o `stdout`.

`console.error` se imprime en la secuencia `stderr`.

No aparecerá en la consola, pero aparecerá en el registro de errores.

## Colorea la salida

Puede colorear la salida de su texto en la consola usando secuencias de escape. Una secuencia de escape es un conjunto de caracteres que identifica un color.

Ejemplo:

```
consola.log("\x1b[33m%$\\x1b[0m", '¡hola!')
```

Puede probar eso en el Nodo REPL, y se imprimirá ¡hola! en amarillo.

Sin embargo, esta es la forma de bajo nivel de hacer esto. La forma más sencilla de colorear la salida de la consola es mediante el uso de una biblioteca. [Tiza](#) es una biblioteca de este tipo, y además de colorear, también ayuda con otras funciones de estilo, como poner el texto en negrita, cursiva o subrayado.

Lo instalas con `npm install chalk`, entonces puedes usarlo:

```
const tiza = require('tiza')
console.log(tiza.amarillo('¡hola!'))
```

Usar `chalk.yellow` es mucho más conveniente que tratar de recordar los códigos de escape, y el código es mucho más legible.

Consulte el enlace del proyecto que publiqué anteriormente para obtener más ejemplos de uso.

## Crear una barra de progreso

[Progreso](#) es un paquete increíble para crear una barra de progreso en la consola. Instálelo usando el comando de instalación de npm:

Este fragmento crea una barra de progreso de 10 pasos y cada 100 ms se completa un paso. Cuando la barra se completa borramos el intervalo:

```
const ProgressBar = require('progreso')

const bar = new ProgressBar(':bar', { total: 10 }) const timer =
setInterval(() => {
  barra.tick() if
  (barra.completa) {
    clearInterval(temporizador)
  } }, 100)
```



# Aceptar entrada desde la línea de comando

## Cómo hacer que un programa CLI de Node.js sea interactivo usando el módulo de nodo de línea de lectura incorporado

¿Cómo hacer que un programa CLI de Node.js sea interactivo?

Desde la versión 7, Node proporciona el [módulo readline](#) para realizar exactamente esto: obtener información de un flujo legible como el flujo `process.stdin`, que durante la ejecución de un programa Node es la entrada del terminal, una línea a la vez.

```
const readline = require('readline').createInterface({  
  entrada: proceso.stdin,  
  salida: proceso.stdout })  
  
readline.question(' ¿Cuál es tu nombre? ', (nombre) => {  
  console.log('Hola ${nombre}!')  
  readline.close() })
```

Este fragmento de código solicita el nombre de usuario, y una vez que se ingresa el texto y el usuario presiona enter, enviamos un saludo.

El método `question()` muestra el primer parámetro (una pregunta) y espera la entrada del usuario. Llama a la función de devolución de llamada una vez que se presiona enter.

En esta función de devolución de llamada, cerramos la interfaz de línea de lectura.

readline ofrece varios otros métodos, y te dejaré comprobarlos en el paquete documentación que vinculé arriba.

Si necesita solicitar una contraseña, ahora es mejor repetirla, pero mostrando un símbolo.

La forma más sencilla es usar el [paquete readline-sync](#) que es muy similar en términos de API y maneja esto fuera de la caja.

El [paquete Inquirer.js](#) proporciona una solución más completa y abstracta .

Puede instalarlo usando `npm install inquirer`, y luego puede replicar el código anterior como este:

```
const indagador = require('indagador')  
  
var preguntas = [{ tipo:  
  'entrada',
```

```
nombre: 'nombre',
mensaje: "¿Cuál es tu nombre?", }]

inquirer.prompt(preguntas).then(respuestas =>
{ console.log(`Hola ${respuestas['nombre']}!`) })
```

Inquirer.js le permite hacer muchas cosas, como pedir opciones múltiples, tener botones de opción, confirmaciones y más.

Vale la pena conocer todas las alternativas, especialmente las integradas proporcionadas por Node, pero si planea llevar la entrada de CLI al siguiente nivel, Inquirer.js es una opción óptima.

# Exponga la funcionalidad de un archivo de nodo mediante exportaciones

## Cómo usar la API module.exports para exponer datos a otros archivos en su aplicación, o también a otras aplicaciones

Node tiene un sistema de módulos integrado.

Un archivo de Node.js puede importar la funcionalidad expuesta por otros archivos de Node.js.

Cuando quieras importar algo que usas

```
const biblioteca = require('./biblioteca')
```

para importar la funcionalidad expuesta en el archivo library.js que reside en la carpeta de archivos actual.

En este archivo, la funcionalidad debe exponerse antes de que otros archivos puedan importarla.

Cualquier otro objeto o variable definido en el archivo por defecto es privado y no está expuesto al mundo exterior.

Esto es lo que nos permite hacer la API module.exports que ofrece el [sistema de módulos](#).

Cuando asigna un objeto o una función como una nueva propiedad de exportación , eso es lo que se expone y, como tal, se puede importar en otras partes de su aplicación o en otras aplicaciones como bien.

Puedes hacerlo de 2 maneras.

El primero es asignar un objeto a module.exports , que es un objeto proporcionado de forma inmediata por el sistema de módulos, y esto hará que su archivo exporte solo ese objeto:

```
const coche =
  { marca: 'Ford',
    model: 'Fiesta'
  }

módulo.exportaciones = coche

//..en el otro archivo

const coche = require('./coche')
```

La segunda forma es agregar el objeto exportado como una propiedad de exportaciones . De esta manera le permite exportar múltiples objetos, funciones o datos:

```
coche constante = {  
    marca: 'Ford',  
    model: 'Fiesta'  
}  
  
exportaciones.coche = coche
```

o directamente

```
exportaciones.coche = {  
    marca: 'Ford',  
    model: 'Fiesta'  
}
```

Y en el otro archivo, lo usará haciendo referencia a una propiedad de su importación:

```
elementos const = requieren ('./elementos')  
artículos.coche
```

o

```
const coche = require('./items').coche
```

¿Cuál es la diferencia entre module.exports y export ?

El primero expone el objeto al que apunta. Este último expone *las propiedades* del objeto al que apunta.  
a.

# npm

Una guía rápida de npm, el poderoso administrador de paquetes clave para el éxito de Node.js. En enero de 2017, se informó que más de 350,000 paquetes se incluyeron en el registro de npm, lo que lo convierte en el repositorio de código de un solo idioma más grande del mundo, y puede estar seguro de que hay un paquete para (¡casi!) todo.



- [Introducción a npm](#)
- [Descargas](#)
  - [Instalando todas las dependencias](#)
  - [Instalación de un solo paquete](#)
  - [Actualización de paquetes](#)
- [Versionado](#)
- [Tareas en ejecución](#)

## Introducción a npm

npm es el administrador de paquetes estándar para Node.js.

En enero de 2017, se informó que más de 350,000 paquetes se incluyeron en el registro de npm, lo que hace es el repositorio de código de un solo idioma más grande del mundo, y puede estar seguro de que hay un paquete para (¡casi!) todo.

Comenzó como una forma de descargar y administrar dependencias de [Node.js](#) paquetes, pero desde entonces se ha convertido en una herramienta utilizada también en [JavaScript frontend](#).

Hay muchas cosas que hace npm .

[Hilo](#) es una alternativa a npm. Asegúrate de comprobarlo también.

## Descargas

npm gestiona las descargas de las dependencias de tu proyecto.

### Instalando todas las dependencias

Si un proyecto tiene un archivo `packages.json` , al ejecutar

```
instalar npm
```

instalará todo lo que necesita el proyecto, en la carpeta `node_modules` , creándolo si aún no existe.

### Instalación de un solo paquete

También puede instalar un paquete específico ejecutando

```
npm install <nombre del paquete>
```

A menudo verás más banderas agregadas a este comando:

- `--save` instala y agrega la entrada a las *dependencias* del archivo `package.json` `--save-dev` instala
- `y agrega la entrada al archivo package.json devDependencies`

La diferencia es principalmente que las `devDependencies` suelen ser herramientas de desarrollo, como una biblioteca de pruebas, mientras que las `dependencias` se incluyen con la aplicación en producción.

### Actualización de paquetes

La actualización también es fácil, al ejecutar

```
actualización de npm
```

```
npm verificará todos los paquetes en busca de una versión más nueva que satisfaga sus restricciones de control de versiones.
```

También puede especificar un solo paquete para actualizar:

```
actualización de npm <nombre del paquete>
```

## Versionado

Además de las descargas simples, npm también administra el **control de versiones**, por lo que puede especificar cualquier versión específica de un paquete o solicitar una versión superior o inferior a la que necesita.

Muchas veces encontrará que una biblioteca solo es compatible con una versión principal de otra biblioteca.

O un error en la última versión de una biblioteca, aún sin corregir, está causando un problema.

Especificar una versión explícita de una biblioteca también ayuda a mantener a todos en la misma versión exacta de un paquete, de modo que todo el equipo ejecute la misma versión hasta que se actualice el archivo package.json .

En todos esos casos, el control de versiones ayuda mucho, y npm sigue el control de versiones semántico (semver) estándar.

## Tareas en ejecución

El archivo package.json admite un formato para especificar tareas de línea de comandos que se pueden ejecutar mediante

```
npm ejecutar <nombre de la tarea>
```

Por ejemplo:

```
{
  "scripts": { "start-
    dev": "node lib/server-development", "start": "node lib/
    server-production"
  },
}
```

Es muy común usar esta función para ejecutar Webpack:

```
{
  "guiones": {
```

```
"reloj": "webpack --reloj --progress --colors --config webpack.conf.js", "dev": "webpack --progress --colors --config webpack.conf.js", "prod" : "NODE_ENV=paquete web de producción -p --config webpack.conf.js", },  
}
```

Entonces, en lugar de escribir esos comandos largos, que son fáciles de olvidar o escribir mal, puede ejecutar

```
$ npm ejecutar ver $  
npm ejecutar dev $  
npm ejecutar prod
```

# ¿Dónde instala npm los paquetes?

## Cómo saber dónde npm instala los paquetes

Leer la [guía npm](#) si está comenzando con npm, incluirá muchos de los aspectos básicos detalles de la misma.

Cuando instala un paquete usando npm (o [yarn](#)), se pueden realizar 2 tipos de instalación:

- una instalación local
- una instalación global

De forma predeterminada, cuando escribe un comando de instalación de npm , como:

```
npm instalar lodash
```

el paquete se instala en el árbol de archivos actual, en la subcarpeta node\_modules .

Mientras esto sucede, npm también agrega la entrada lodash en la propiedad de dependencias del [archivo package.json](#) presente en la carpeta actual.

Una instalación global se realiza usando el indicador -g :

```
npm install -g lodash
```

Cuando esto sucede, npm no instalará el paquete en la carpeta local, sino que utilizará una ubicación global.

¿Dónde exactamente?

El comando npm root -g le dirá dónde está esa ubicación exacta en su máquina.

En macOS o Linux, esta ubicación podría ser /usr/local/lib/node\_modules . En Windows podría ser C:\Users\USTED\AppData\Roaming\npm\node\_modules

Sin embargo, si usa nvm para administrar las versiones de Node.js, esa ubicación será diferente.

Yo, por ejemplo, uso nvm y la ubicación de mis paquetes se mostró como  
/Users/flavio/.nvm/versions/node/v8.9.0/lib/node\_modules .

# Cómo usar o ejecutar un paquete instalado usando npm

## Cómo incluir y usar en su código un paquete instalado en su carpeta node\_modules

Cuando instala usando npm un paquete en su carpeta node\_modules , o también globalmente, ¿cómo ¿Lo usas en tu código de nodo?

Digamos que instalas lodash , la popular biblioteca de utilidades de JavaScript, usando

```
npm instalar lodash
```

Esto instalará el paquete en la carpeta local node\_modules .

Para usarlo en su código, solo necesita importarlo a su programa usando require :

```
constante _ = requerir ('lodash')
```

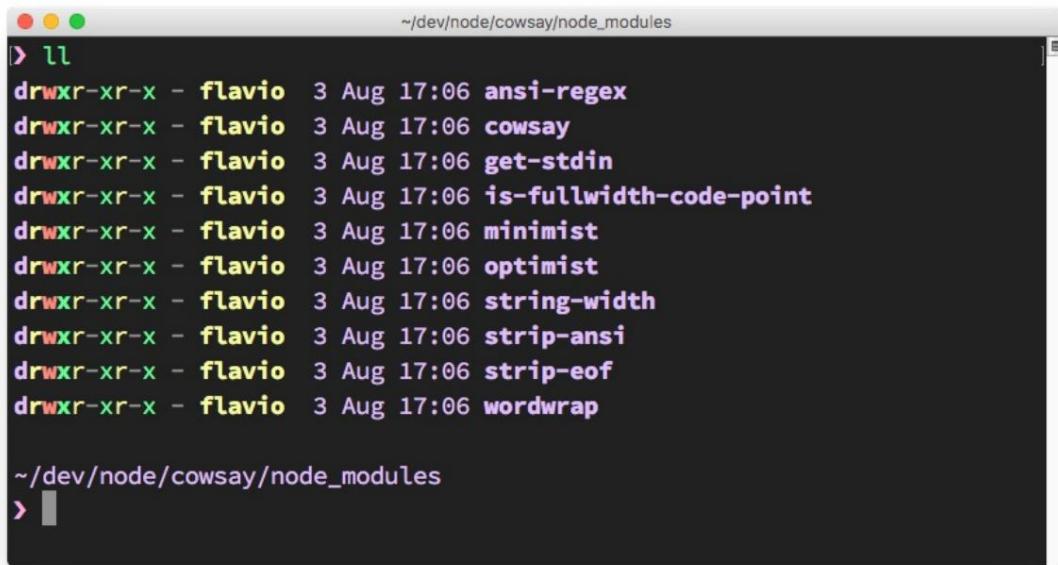
¿Qué sucede si su paquete es un ejecutable?

En este caso, colocará el archivo ejecutable en la carpeta node\_modules/.bin/ .

Una manera fácil de demostrar esto es [cowsay](#).

El paquete cowsay proporciona un programa de línea de comandos que se puede ejecutar para hacer que una vaca decir algo (y otros animales también ý.)

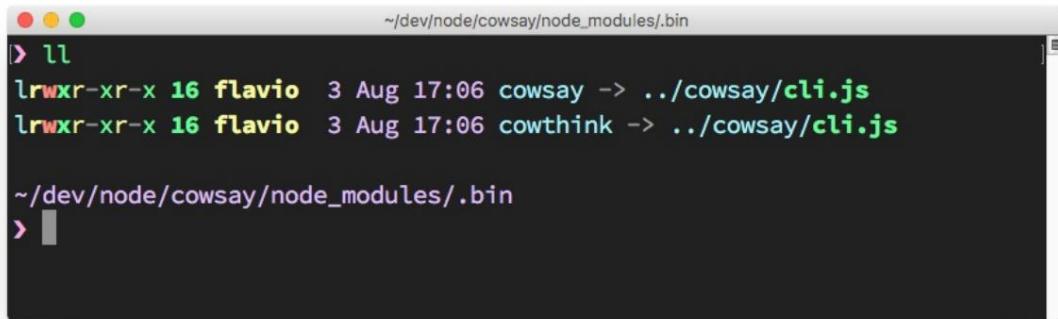
Cuando instala el paquete usando npm install cowsay , se instalará solo y algunos dependencias en la carpeta node\_modules:



```
~/dev/node/cowsay/node_modules
> ll
drwxr-xr-x - flavio 3 Aug 17:06 ansi-regex
drwxr-xr-x - flavio 3 Aug 17:06 cowsay
drwxr-xr-x - flavio 3 Aug 17:06 get-stdin
drwxr-xr-x - flavio 3 Aug 17:06 is-fullwidth-code-point
drwxr-xr-x - flavio 3 Aug 17:06 minimist
drwxr-xr-x - flavio 3 Aug 17:06 optimist
drwxr-xr-x - flavio 3 Aug 17:06 string-width
drwxr-xr-x - flavio 3 Aug 17:06 strip-ansi
drwxr-xr-x - flavio 3 Aug 17:06 strip-eof
drwxr-xr-x - flavio 3 Aug 17:06 wordwrap

~/dev/node/cowsay/node_modules
>
```

Hay una carpeta .bin oculta, que contiene enlaces simbólicos a los binarios de cowsay:



```
~/dev/node/cowsay/node_modules/.bin
> ll
lrwxr-xr-x 16 flavio 3 Aug 17:06 cowsay -> ../cowsay/cli.js
lrwxr-xr-x 16 flavio 3 Aug 17:06 cowthink -> ../cowsay/cli.js

~/dev/node/cowsay/node_modules/.bin
>
```

¿Cómo ejecutas esos?

Por supuesto, puede escribir ./node\_modules/.bin/cowsay para ejecutarlo y funciona, pero [npx](#), incluido en las versiones recientes de npm (desde 5.2), es una opción mucho mejor. solo ejecutas:

```
vaca npx
```

y npx encontrará la ubicación del paquete.



A screenshot of a terminal window titled 'npx cowsay take me out of here'. The terminal shows the command being run and the resulting ASCII art of a cow's head. The cow has a single horn, two eyes, and a mouth. The terminal is located at the path '~/dev/node/cowsay/node\_modules/.bin'.

```
~/.dev/node/cowsay/node_modules/.bin
> npx cowsay take me out of here
-----
< take me out of here >
-----
 \  ^__^
  \  (oo)\_____
   (__)\       )\/\
      ||----w |
      ||     ||
```

# El archivo paquete.json

**El archivo package.json es un elemento clave en muchas bases de código de aplicaciones basadas en el ecosistema Node.js.**

Si trabaja con JavaScript, o si alguna vez ha interactuado con un proyecto de JavaScript, Node.js o un proyecto de interfaz, seguramente conoció el archivo package.json .

¿Para qué es eso? ¿Qué debe saber al respecto y cuáles son algunas de las cosas geniales que puede hacer con él?

El archivo package.json es una especie de manifiesto para su proyecto. Puede hacer muchas cosas, completamente ajenas. Es un repositorio central de configuración de herramientas, por ejemplo. También es donde npm y yarn almacena los nombres y versiones del paquete que instaló.

- [La estructura del archivo](#)
- [Desglose de propiedades](#)
  - [nombre](#)
  - [autor](#)
  - [contribuyentes](#)
  - [insectos](#)
  - [página principal](#)
  - [versión](#)
  - [licencia](#)
  - [palabras clave](#)
  - [descripción](#)
  - [repositorio](#)
  - [principal](#)
  - [privado](#)
  - [guiones](#)
  - [dependencias](#)
  - [devDependencias](#)
  - [motores](#)
  - [lista de navegadores](#)
  - [Propiedades específicas del comando](#)
- [Versiones del paquete](#)

## La estructura del archivo

Aquí hay un archivo de ejemplo package.json:

```
{
}
```

¡Esta vacío! No hay requisitos fijos de lo que debería estar en un archivo package.json para una aplicación. El único requisito es que respete el formato JSON, de lo contrario no podrá ser leído por programas que intenten acceder a sus propiedades mediante programación.

Si está creando un paquete de Node.js que desea distribuir a través de npm , las cosas cambian radicalmente y debe tener un conjunto de propiedades que ayudarán a otras personas a usarlo. Ya veremos más sobre esto más adelante.

Este es otro paquete.json:

```
{
  "nombre": " proyecto de prueba"
}
```

Define una propiedad de nombre , que indica el nombre de la aplicación o paquete que se encuentra en la misma carpeta donde se encuentra este archivo.

Aquí hay un ejemplo mucho más complejo, que extrae de una aplicación Vue.js de muestra:

```
{
  "nombre": " proyecto de prueba",
  "versión": "1.0.0",
  "descripción": "Un proyecto Vue.js", "principal": "src/main.js", "privado": verdadero, "scripts": {
    "dev": "webpack-dev-server --inline --progress --config build/webpack.dev.conf.js", "start": "npm run dev", "unit": "jest --config test/unit/jest.conf.js --coverage", "test": "npm run unit", "lint": "eslint --ext .js,.vue src test/unit", "build": "construcción del nodo/construcción.js"
  },
  "dependencias": {
    "ver": "^2.5.2"
  },
  "devDependencies": {
    "autoprefixer": "^7.1.2", "babel-core": "^6.22.1", "babel-eslint": "^8.2.1", "babel-helper-vue-jsx-merge-props": "^2.0.3", "babel-jest": "^21.0.2", "babel-loader": "^7.1.1", "babel-plugin-dynamic-import-nodo": "^1.2.0", "babel-plugin-syntax-jsx": "^6.18.0",
  }
}
```

```
"babel-plugin-transform-es2015-modules-commonjs": "^6.26.0", "babel-plugin-transform-runtime": "^6.22.0", "babel-plugin-transform-vue-jsx": "^3.5.0", "babel-preset-env": "^1.3.2", "babel-preset-stage-2": "^6.22.0", "tiza": "^2.0.1", "copy-webpack-plugin": "4.0.1", "css-loader": "0.28.0", "eslint": "4.15.0", "eslint-config-airbnb-base": "11.3.0", "eslint-friendly-formatter": "3.0.0", "eslint-import-resolver-webpack": "0.8.3", "eslint-loader": "1.7.1", "eslint-plugin-import": "2.7.0", "eslint-plugin-vue": "4.0.0", "extract-text-webpack-plugin": "3.0.0", "archivo -loader": "1.1.4", "errores amistosos-webpack-plugin": "1.6.1", "html-webpack-plugin": "2.30.1", "broma": "22.0 .4", "jest-serializer-vue": "0.3.0", "node-notifier": "5.1.2", "optimize-css-assets-webpack-plugin": "3.2.0" , "ora": "1.2.0", "buscador de puertos": "1.0.13", "importación postcss": "11.0.0", "cargador postcss": "2.0.8", "postcss-url": "7.2.1", "rimraf": "2.6.0", "semver": "5.3.0", "shelljs": "0.7.6", "uglifyjs-webpack complemento k": "1.1.1", "cargador de URL": "0.5.8", "vue-jest": "1.0.2", "vue-loader": "13.3.0" , "vue-style-loader": "3.0.1", "vue-template-compiler": "2.5.2", "webpack": "3.6.0", "webpack-bundle-analyzer": "2.9.0", "webpack-dev-servidor": "2.9.1", "webpack-fusión": "4.1.0"
```

```
},
"motores":
{ "nodo": ">= 6.0.0",
  "npm": ">= 3.0.0"
},
"lista de navegadores":
[ "> 1%", "últimas 2
versiones",
  "no es decir <= 8"
]
}
```

aquí pasan *muchas* cosas:

- nombre establece el nombre de la aplicación/paquete
- versión indica la versión actual
- descripción es una breve descripción de la aplicación/paquete
- principal establece el punto de entrada para la aplicación privado si
- se establece en verdadero evita que la aplicación/paquete se publique accidentalmente en scripts npm define un
- conjunto de scripts de nodo que puede ejecutar dependencias establece una lista de npm paquetes instalados
- como dependencias devDependencies establece una lista de paquetes npm instalados como motores de
- dependencias de desarrollo establece en qué versiones de Node funciona este paquete/aplicación browserslist
- se usa para indicar qué navegadores (y sus versiones) desea admitir
- 

Todas esas propiedades son utilizadas por npm u otras herramientas que podamos usar.

## Desglose de propiedades

En esta sección se describen detalladamente las propiedades que puede utilizar. Me refiero a "paquete", pero lo mismo se aplica a las aplicaciones locales que no usa como paquetes.

La mayoría de esas propiedades solo se usan en <https://www.npmjs.com/>, otros por scripts que interactúan con su código, como npm u otros.

### nombre

Establece el nombre del paquete.

Ejemplo:

```
"nombre": " proyecto de prueba"
```

El nombre debe tener menos de 214 caracteres, no debe tener espacios, solo puede contener letras minúsculas, guiones ( - ) o guiones bajos ( \_ ).

Esto se debe a que cuando se publica un paquete en npm , obtiene su propia URL en función de esta propiedad.

Si publicó este paquete públicamente en GitHub, un buen valor para esta propiedad es el nombre del repositorio de GitHub.

### autor

Muestra el nombre del autor del paquete

Ejemplo:

```
{
  "autor": "Flavio Copes <flavio@flaviocopes.com> (https://flaviocopes.com)"
}
```

También se puede utilizar con este formato:

```
{
  "autor":
    { "nombre": "Flavio Copes",
      "email": "flavio@flaviocopes.com", "url":
      "https://flaviocopes.com"
    }
}
```

Además del autor, el proyecto puede tener uno o más colaboradores. Esta propiedad es una matriz que los enumera.

Ejemplo:

```
{
  "colaboradores": [
    "Flavio Copes <flavio@flaviocopes.com> (https://flaviocopes.com)"
  ]
}
```

También se puede utilizar con este formato:

```
{
  "colaboradores": [
    {
      "nombre": "Flavio Copes",
      "email": "flavio@flaviocopes.com", "url":
      "https://flaviocopes.com"
    }
  ]
}
```

Enlaces al rastreador de problemas del paquete, muy probablemente una página de problemas de GitHub

Ejemplo:

```
{
  "errores": "https://github.com/flaviocopes/package/issues"
```

```
}
```

## página principal

Establece la página de inicio del paquete

Ejemplo:

```
{
  "homepage": "https://flaviocopes.com/package"
}
```

## versión

Indica la versión actual del paquete.

Ejemplo:

```
"versión": "1.0.0"
```

Esta propiedad sigue la notación semántica de versiones (semver) para versiones, lo que significa que la versión siempre se expresa con 3 números: xxx .

El primer número es la versión principal, el segundo la versión secundaria y el tercero es el parche versión.

Hay un significado en estos números: una versión que solo corrige errores es una versión de parche, una versión que introduce cambios compatibles con versiones anteriores es una versión menor, una versión principal puede tener cambios importantes.

## licencia

Indica la licencia del paquete.

Ejemplo:

```
"licencia": "MIT"
```

## palabras clave

Esta propiedad contiene una matriz de palabras clave que se asocian con lo que hace su paquete.

Ejemplo:

```

"palabras clave": [
  "correo
  electrónico", "aprendizaje automático",
  "ai"
]

```

Esto ayuda a las personas a encontrar su paquete cuando navegan por paquetes similares o cuando navegan por <https://www.npmjs.com/> sitio web.

## descripción

Esta propiedad contiene una breve descripción del paquete.

Ejemplo:

```
"description": "Un paquete para trabajar con cadenas"
```

Esto es especialmente útil si decide publicar su paquete en npm para que las personas puedan averiguar de qué se trata el paquete.

## repositorio

Esta propiedad especifica dónde se encuentra este depósito de paquetes.

Ejemplo:

```
"repositorio": "github:flaviocopes/testing",
```

Observe el prefijo de github . Hay otros servicios populares horneados en:

```
"repositorio": "gitlab:flaviocopes/testing",
```

```
"repositorio": "bitbucket:flaviocopes/testing",
```

Puede establecer explícitamente el sistema de control de versiones:

```

"repositorio": {
  "tipo": "git", "url":
  "https://github.com/flaviocopes/testing.git"
}

```

Puede utilizar diferentes sistemas de control de versiones:

```
"repositorio": { "tipo":  
    "svn",  
    "url": "..."  
}
```

## principal

Establece el punto de entrada para el paquete.

Cuando importa este paquete en una aplicación, ahí es donde la aplicación buscará las exportaciones del módulo.

Ejemplo:

```
"principal": "src/principal.js"
```

## privado

si se establece en verdadero , evita que la aplicación/paquete se publique accidentalmente en npm

Ejemplo:

```
"privado": verdadero
```

## guiones

Define un conjunto de scripts de nodo que puede ejecutar

Ejemplo:

```
"guiones": {  
    "dev": "webpack-dev-server --inline --progress --config build/webpack.dev.conf.js", "start": "npm run dev", "unit":  
    "jest --config test /unit/jest.conf.js --coverage", "test": "npm run unit", "lint": "eslint --ext .js,.vue src test/unit",  
    "build": "node build /construir.js"  
  
}
```

Estos scripts son aplicaciones de línea de comandos. Puede ejecutarlos llamando a npm run XXXX o hilo XXXX , donde XXXX es el nombre del comando. Ejemplo: npm ejecutar dev .

Puede usar cualquier nombre que desee para un comando y los scripts pueden hacer literalmente cualquier cosa que desee. desear.

## dependencias

Establece una lista de paquetes npm instalados como dependencias.

Cuando instala un paquete usando npm o yarn:

```
npm install <NOMBRE DEL PAQUETE>
hilo agregar <NOMBRE DEL PAQUETE>
```

ese paquete se inserta automáticamente en esta lista.

Ejemplo:

```
"dependencias": {
  "ver": "^2.5.2"
}
```

## devDependencias

Establece una lista de paquetes npm instalados como dependencias de desarrollo.

Se diferencian de las dependencias porque están diseñadas para instalarse solo en un desarrollo . máquina, no es necesario para ejecutar el código en producción.

Cuando instala un paquete usando npm o yarn:

```
npm install --dev <NOMBRE DEL PAQUETE>
yarn add --dev <NOMBRE DEL PAQUETE>
```

ese paquete se inserta automáticamente en esta lista.

Ejemplo:

```
"devDependencies": {
  "autoprefixer": "^7.1.2",
  "babel-core": "^6.22.1"
}
```

## motores

Establece en qué versiones de Node.js y otros comandos funciona este paquete/aplicación

Ejemplo:

```
"motores": {
```

```
"nodo": ">= 6.0.0",
"npm": ">= 3.0.0", "hilo":
"~0.13.0"
}
```

## lista de navegadores

Se utiliza para indicar qué navegadores (y sus versiones) desea admitir. Es referenciado por Babel, Autoprefixer y otras herramientas, para agregar solo los polyfills y fallbacks necesarios para los navegadores a los que se dirige.

Ejemplo:

```
"lista de navegadores":
[ "> 1%", "últimas 2
versiones",
"no es decir <= 8"
]
```

Esta configuración significa que desea admitir las últimas 2 versiones principales de todos los navegadores con al menos el 1% de uso (de [CanIUse.com](#) stats), excepto IE8 y versiones inferiores.

[\(ver más\)](#)

## Propiedades específicas del comando

El archivo package.json también puede alojar configuraciones específicas de comandos, por ejemplo, para Babel, ESLint y más.

Cada uno tiene una propiedad específica, como `babel` y otros. Esos son comandos `eslintConfig`, específica, y puede encontrar cómo usarlas en la documentación respectiva del comando/proyecto.

## Versiones del paquete

Ha visto en la descripción anterior números de versión como estos: `~3.0.0` o `^0.13.0`.

¿Qué significan y qué otros especificadores de versión puede usar?

Ese símbolo especifica qué actualizaciones acepta su paquete, de esa dependencia.

Dado que al usar semver (versiones semánticas) todas las versiones tienen 3 dígitos, el primero es el lanzamiento principal, el segundo el lanzamiento menor y el tercero es el lanzamiento del parche, tiene estos normas:

- `~`: si escribe `~0.13.0`, solo desea actualizar las versiones de parches: `0.13.1` está bien, pero

0.14.0 no lo es.

- ^ : si escribe ^0.13.0 , desea actualizar el parche y las versiones menores: 0.13.1 , 0.14.0 y así.
- \* : si escribes \* , eso significa que acepta todas las actualizaciones, incluidas las actualizaciones de versiones principales.
- > : aceptas cualquier versión superior a la que especificas
- >= : aceptas cualquier versión igual o superior a la que especificas
- <= : aceptas cualquier versión igual o inferior a la que especificas
- < : aceptas cualquier versión inferior a la que especificas

También hay otras reglas:

- sin símbolo: acepta solo la versión específica que especifique
- Latest : desea utilizar la última versión disponible

y puede combinar la mayoría de los anteriores en rangos, así: 1.0.0 || >=1.1.0 <1.2.0 use , a cualquiera 1.0.0 o una versión de 1.1.0 en adelante, pero inferior a 1.2.0.

# El archivo package-lock.json

## El archivo package-lock.json se genera automáticamente al instalar paquetes de nodos. Aprende de qué se trata

En la versión 5, [npm](#) introdujo el archivo package-lock.json .

¿Qué es eso? Probablemente conozca el [archivo package.json](#) , que es mucho más común y existe desde hace mucho más tiempo.

El objetivo del archivo es realizar un seguimiento de la versión exacta de cada paquete que se instala para que un producto sea 100% reproducible de la misma manera, incluso si los paquetes se actualizan por sus mantenedores

Esto resuelve un problema muy específico que package.json dejó sin resolver. En package.json puede establecer a qué versiones desea actualizar ( parche o menor), utilizando la notación **semver** , por ejemplo:

- si escribe ~0.13.0 , solo desea actualizar las versiones de parches: 0.13.1 está bien, pero 0.14.0 no es.
- si escribe ^0.13.0 , desea actualizar el parche y las versiones menores: 0.13.1 , 0.14.0 y pronto.
- si escribes 0.13.0 , esa es la versión exacta que se utilizará, siempre

No se compromete con Git en su carpeta node\_modules , que generalmente es enorme, y cuando intenta replicar el proyecto en otra máquina usando el comando npm install , si especificó la sintaxis ~ y se lanzó un parche de un paquete , ese va

Para ser instalado. Lo mismo para ^ y lanzamientos menores.

Si especifica versiones exactas, como 0.13.0 en el ejemplo, no se verá afectado por este problema.

Podrías ser tú u otra persona intentando inicializar el proyecto en el otro lado del mundo.  
ejecutando npm install .

Entonces, su proyecto original y el proyecto recién inicializado son realmente diferentes. Incluso si un parche o una versión menor no debe introducir cambios importantes, todos sabemos que los errores pueden (y lo harán) deslizarse.

El package-lock.json establece su versión actualmente instalada de cada paquete en piedra , y npm usará esas versiones exactas cuando ejecute npm install .

Este concepto no es nuevo, y otros administradores de paquetes de lenguajes de programación (como Composer en PHP) usan un sistema similar desde hace años.

El archivo package-lock.json debe confirmarse en su repositorio de Git, de modo que otras personas puedan recuperarlo, si el proyecto es público o si tiene colaboradores, o si usa Git como fuente para las implementaciones.

Las versiones de las dependencias se actualizarán en el archivo package-lock.json cuando ejecute npm  
actualizar \_

## Un ejemplo

Esta es una estructura de ejemplo de un archivo package-lock.json que obtenemos cuando ejecutamos npm  
install cowsay en una carpeta vacía:

```
{
  "requiere": verdadero,
  "lockfileVersion": 1,
  "dependencias": { "ansi-
    regex": { "versión":
      "3.0.0", "resuelto": "https://
        registry.npmjs.org/ansi -regex/-ansi-regex-3.0.0.tgz", "integridad":
      "sha1-7QMXwyIGT3lGbAKWa922Bas32Zg=" }, "cowsay": { "versión": "1.3.1", "resuelto": "https://
        registry.npmjs.org/cowsay/-/cowsay-1.3.1.tgz"
      ,
      "integridad": "sha512-3PVFe6FePVtPj1HTeLin9v8WyLI+VmM1I1H/5P+BTTDkM
Ajufp+0F9eLjzRnOHzVAYelYFF5po5NjRrgefnRMQ==",
      "requiere": { "get-
        stdin": "^5.0.1", "optimist":
        "~0.6.1", "string-width":
        "~2.1.1", "strip-eof": "^ 1.0.0"
      }
    },
    "get-stdin":
      { "versión": "5.0.1",
        "resuelto": "https://registry.npmjs.org/get-stdin/-/get-stdin-5.0.1.tgz", "integridad": "sha1-
Ei4WFZHiH/TFJTAwVpPyDmOTo5g="
      },
      "es-punto-de-código-de-ancho-
        completo": { "versión": "2.0.0",
          "resuelto": "https://registry.npmjs.org/es-punto-de-código-de-ancho-completo/-/
            es-ancho-completo-punto-de-código-2.0.0.tgz",
          "integridad": "sha1-o7MKXE8ZkYMWeqq5O+764937ZU8=", },
          "minimista": {
            "versión": "0.0.10",
            "resuelto": "https://registry.npmjs.org/minimist/-/minimist-0.0.10 .tgz",
          }
        }
      }
```

## archivo package-lock.json

```
"integridad": "sha1-3j+YVD2/lgr5IrRoMfNqDYwHc8="
},
"optimista": {
  { "versión": "0.6.1",
    "resuelto": "https://registry.npmjs.org/optimist/-/optimist-0.6.1.tgz", "integridad": "
sha1-2j6nRob6laGaERwybpDrFaAZZoY=",
    "requiere": {
      "minimista": "~0.0.1", "ajuste
      de palabra": "~0.0.2"
    }
  },
  "ancho de cadena": {
    "versión": "2.1.1",
    "resuelto": "https://registry.npmjs.org/string-width/-/string-width-2.1.1.tgz", "integridad": "sha512-
nOqH59deCq9SRHlxq1Aw85Jnt4w6KvLkqWVik6oA9ZkIXLNIOlqg4F2yrT1MVaT jAqvVwdfeZ7w7aCvJD7ugkw==",
    "requiere": {
      "es-punto-de-código-de-ancho-completo": "
^2.0.0", "strip-ansi": "^4.0.0"
    }
  },
  "strip-ansi": {
    "versión": "4.0.0",
    "resuelto": "https://registry.npmjs.org/strip-ansi/-/strip-ansi-4.0.0.tgz ", "integridad": "sha1-
qEeQIusaw2iocTibY1JixQXuNo8=", "requiere": { "ansi-regex": "^3.0.0"
  }
},
"strip-eof": {
  "versión": "1.0.0",
  "resuelto": "https://registry.npmjs.org/strip-eof/-/strip-eof-1.0.0.tgz ", "integridad": "sha1-u0P/
VZim6wXYm1n80SnJgzE2Br8=", "wordwrap": { "versión": "0.0.3", "resuelto": "https://
registry.npmjs.org/wordwrap/-/ wordwrap-0.0.3.tgz", "integridad": "sha1-o9XabNXAvAAI03I0u68b7WMFkQc="
}
}
```

Instalamos cowsay , que depende de

- get-stdin
  - optimista
  - ancho de cadena
  - tira-eof

A su vez, esos paquetes requieren otros paquetes, como podemos ver en la propiedad require que algunos tienen:

- ansi-regex
- es-punto-de-código-de-ancho-completo
- mínimos
- ajuste de línea
- tira-eof

Se agregan en orden alfabético en el archivo, y cada uno tiene un campo de versión , un campo resuelto que apunta a la ubicación del paquete y una cadena de integridad que podemos usar para verificar el paquete.

# Encuentre la versión instalada de un paquete npm

## Cómo averiguar qué versión de un paquete en particular ha instalado en su aplicación

Para ver la última versión de todo el paquete npm instalado, incluidas sus dependencias:

```
lista npm
```

Ejemplo:

```
ÿ npm list /  
Users/flavio/dev/node/cowsay yy  
cowsay@1.3.1 yy get-stdin@5.0.1 yy  
optimista@0.6.1 yy minimist@0.0.10  
ÿ yy wordwrap@0.0.3 yy string-  
width@2.1.1 yy is-fullwidth-code-  
point@2.0.0 reject@3.0.0@4.0.0  
eof@1.0.0
```

```
yy  
yy
```

También puede simplemente abrir el archivo package-lock.json , pero esto implica un escaneo visual.

```
npm list -g es lo mismo, pero para paquetes instalados globalmente.
```

Para obtener solo sus paquetes de nivel superior (básicamente, los que le dijo a npm que instalara y que enumeró en el paquete.json ), ejecute npm list --depth=0 :

```
ÿ lista npm --profundidad=0  
/Usuarios/flavio/dev/nodo/cowsay  
yy vacasay@1.3.1
```

Puede obtener la versión de un paquete específico especificando el nombre:

```
ÿ lista npm cowsay /  
Users/flavio/dev/node/cowsay  
yy cowsay@1.3.1
```

Esto también funciona para las dependencias de los paquetes que instaló:

```
ÿ npm list minimalist /Users/  
flavio/dev/node/cowsay ÿÿÿ cowsay@1.3.1  
ÿÿÿ optimist@0.6.1  
  
ÿÿÿ minimalista@0.0.10
```

Si desea ver cuál es la última versión disponible del paquete en el repositorio npm, ejecute

npm ver la versión de [nombre\_del\_paquete] :

```
ÿ npm ver versión cowsay  
  
1.3.1
```

# Cómo instalar una versión anterior de un paquete npm

**Aprenda a instalar una versión anterior de un paquete npm, algo que podría ser útil para resolver un problema de compatibilidad**

Puede instalar una versión anterior de un paquete npm usando la sintaxis @ :

```
npm install <paquete>@<versión>
```

Ejemplo:

```
npm instalar cowsay
```

instala la versión 1.3.1 (en el momento de escribir este artículo).

Instale la versión 1.2.0 con:

```
npm instalar cowsay@1.2.0
```

Lo mismo se puede hacer con paquetes globales:

```
npm install -g webpack@4.16.4
```

También puede estar interesado en enumerar todas las versiones anteriores de un paquete. Puedes hacerlo con

```
npm ver versiones <paquete> :
```

```
ÿ npm ver versiones cowsay
```

```
[ '1.0.0',
  '1.0.1',
  '1.0.2',
  '1.0.3',
  '1.1.0',
  '1.1.1',
  '1.1.2',
  '1.1.3',
  '1.1.4',
  '1.1.5',
  '1.1.6',
  '1.1.7',
  '1.1.8',
  '1.1.9',
  '1.2.0',
  '1.2.1',
```

```
'1.3.0',  
'1.3.1']
```

# Cómo actualizar todas las dependencias de Node a su última versión

## ¿Cómo se actualiza todo el almacén de dependencias de npm en el archivo package.json a su última versión disponible?

Cuando instala un paquete usando `npm install <packagename>` , la última versión disponible del paquete se descarga y se coloca en la carpeta `node_modules` , y se agrega una entrada correspondiente a los archivos `package.json` y `package-lock.json` que están presentes en su carpeta actual.

`npm` calcula las dependencias e instala la última versión disponible de ellas también.

Supongamos que instala `cowsay` , una herramienta de línea de comandos genial que le permite hacer que una vaca diga cosas.

Cuando npm instala `cowsay` , esta entrada se agrega al archivo `package.json` :

```
{  
  "dependencias": {  
    "vaquera": "^ 1.3.1"  
  }  
}
```

y este es un extracto de `package-lock.json` , donde eliminé las dependencias anidadas para mayor claridad:

```
{  
  "requiere": verdadero,  
  "lockfileVersion": 1,  
  "dependencias": { "cowsay":  
    { "versión": "1.3.1",  
      "resuelto": "https://  
      registry.npmjs.org/cowsay/- /cowsay-1.3.1.tgz", "integridad":  
      "sha512-3PVFe6FePVtPj1HTeLin9v8WyLI+VmM1I1H/5P+BTTDkMAjufp+0F9eLjzRnOHZ  
      VAYelYFF5po5NjRrgefnRMQ==", "requiere": { "get-stdin": "^5.0.1", "optimista": "~0.6.1", "ancho de cadena": "~2.1.1", "strip-  
      eof": "1.0.0"  
    }  
  }  
}
```

Ahora esos 2 archivos nos dicen que instalamos la versión 1.3.1 de cowsay, y nuestra regla para las actualizaciones es ^ 1.3.1 , que para las reglas de control de versiones de npm significa que npm puede actualizar a parche y menor versiones: 0.13.1 , 0.14.0 y así sucesivamente.

Si hay una nueva versión menor o un parche y escribimos npm update , la versión instalada es actualizado, y el archivo package-lock.json se llenó diligentemente con la nueva versión.

paquete.json permanece sin cambios.

Para descubrir nuevas versiones de los paquetes, ejecute npm obsoleto .

Aquí está la lista de algunos paquetes obsoletos en un repositorio que no actualicé durante bastante tiempo:

Package	Current	Wanted	Latest	Location
autoprefixer	8.6.5	8.6.5	9.1.0	ghostwriter
css-loader	0.28.4	0.28.11	1.0.0	ghostwriter
cssnano	3.10.0	3.10.0	4.0.5	ghostwriter
extract-text-webpack-plugin	2.1.2	2.1.2	3.0.2	ghostwriter
node-sass	4.5.3	4.9.2	4.9.2	ghostwriter
normalize.css	7.0.0	7.0.0	8.0.0	ghostwriter
optimize-css-assets-webpack-plugin	2.0.0	2.0.0	5.0.0	ghostwriter
postcss-cli	5.0.1	5.0.1	6.0.0	ghostwriter
postcss-discard-comments	2.0.4	2.0.4	4.0.0	ghostwriter
sass-loader	6.0.6	6.0.7	7.1.0	ghostwriter
style-loader	0.18.2	0.18.2	0.21.0	ghostwriter
webpack	3.0.0	3.12.0	4.16.4	ghostwriter

Algunas de esas actualizaciones son lanzamientos importantes. Ejecutar la actualización de npm no actualizará la versión de aquellos. Las versiones principales nunca se actualizan de esta manera porque (por definición) introducen rompiendo cambios, y npm quiere ahorrarle problemas.

Para actualizar todos los paquetes a una nueva versión principal, instale el paquete npm-check-updates globalmente:

```
npm install -g npm-verificar actualizaciones
```

luego ejecútalo:

```
ncu -u
```

esto actualizará todas las sugerencias de versión en el archivo package.json , a dependencias y dependencias de desarrollo para que npm pueda instalar la nueva versión principal.

Ahora está listo para ejecutar la actualización:

actualización de npm

Si acaba de descargar el proyecto sin las dependencias de node\_modules y desea instale primero las nuevas versiones brillantes, simplemente ejecute

instalar npm

# Reglas de versionado semántico

**El control de versiones semántico es una convención utilizada para proporcionar un significado a las versiones.**

Si hay algo bueno en los paquetes de Node.js, es que todos acordaron usar el control de versiones semántico para la numeración de sus versiones.

El concepto de Versionado Semántico es simple: todas las versiones tienen 3 dígitos: xyz .

- el primer dígito es la versión principal el
- segundo dígito es la versión secundaria el
- tercer dígito es la versión del parche

Cuando haces un nuevo lanzamiento, no solo subes un número como quieras, sino que tienes reglas:

- actualiza la versión principal cuando crea una API incompatible cambia la versión
- secundaria cuando agrega funcionalidad de manera compatible con versiones anteriores actualiza la versión del
- parche cuando realiza correcciones de errores compatibles con versiones anteriores

La convención se adopta en todos los lenguajes de programación, y es muy importante que cada paquete npm se adhiera a ella, porque todo el sistema depende de eso.

¿Por qué es eso tan importante?

Debido a que npm estableció algunas reglas que podemos usar en el [archivo package.json](#) para elegir qué versiones podemos actualizar nuestros paquetes cuando ejecutamos `npm update` .

Las reglas usan esos símbolos:

- ^
- ~
- >
- >=
- <
- <=
- =
- -
- ||

Veamos esas reglas en detalle:

- ^ : si escribe `^0.13.0` cuando ejecuta `npm update` , puede actualizarse a parche y menor versiones: `0.13.1` , `0.14.0` y así sucesivamente.
- ~ : si escribes `~0.13.0` `0.13.1` , cuando se ejecuta la actualización de npm , puede actualizarse a versiones de parches: `0.13.1` está bien, pero `0.14.0` no.

- > : aceptas cualquier versión superior a la que especificas
- >= : aceptas cualquier versión igual o superior a la que especificas <= :
- aceptas cualquier versión igual o inferior a la que especificas < :
- aceptas cualquier versión inferior al que especifiques = : aceptas esa
- versión exacta - : aceptas un rango de versiones. Ejemplo: 2.1.0 - 2.6.2
- || : combinás conjuntos. Ejemplo: < 2.1 || > 2.6
- 

Puede combinar algunas de esas notaciones, por ejemplo, use 1.0.0 || >=1.1.0 <1.2.0 para usar 1.0.0 o una versión de 1.1.0 en adelante, pero anterior a 1.2.0.

También hay otras reglas:

- sin símbolo: acepta solo la versión específica que especifique ( 1.2.1 )
- Latest : desea utilizar la última versión disponible

# Desinstalación de paquetes npm

## Cómo desinstalar un paquete de npm Node, local o globalmente

Para desinstalar un paquete que instaló **localmente** previamente (usando `npm install <nombre del paquete>` en la carpeta `node_modules` , ejecute

```
npm desinstalar <nombre-paquete>
```

desde la carpeta raíz del proyecto (la carpeta que contiene la carpeta `node_modules`).

Usando el indicador `-S` , o `--save` , esta operación también eliminará la referencia en el [archivo `package.json`](#) .

Si el paquete era una dependencia de desarrollo, enumerada en `devDependencies` del archivo `package.json` , debe usar el indicador `-D` / `--save-dev` para eliminarlo del archivo:

```
npm uninstall -S <nombre del paquete>
npm uninstall -D <nombre del paquete>
```

Si el paquete se instala **globalmente**, debe agregar el indicador `-g` / `--global` :

```
npm uninstall -g <nombre-paquete>
```

por ejemplo:

```
npm desinstalar -g paquete web
```

y puede ejecutar este comando desde cualquier lugar que desee en su sistema porque la carpeta en la que se encuentra actualmente no importa.

# Paquetes globales o locales

## ¿Cuándo es mejor instalar un paquete globalmente? ¿Por qué?

La principal diferencia entre los paquetes locales y globales es la siguiente:

- **Los paquetes locales** se instalan en el directorio donde ejecuta `npm install <package-name>`, y se colocan en la carpeta `node_modules` en este directorio. Los **paquetes globales** se colocan en un solo lugar en su sistema (exactamente dónde depende de su configuración), independientemente de dónde ejecute `npm install -g <package-name>`

En su código, ambos son necesarios de la misma manera:

```
require('nombre-paquete')
```

entonces, ¿cuándo se debe instalar de una forma u otra?

En general, **todos los paquetes deben instalarse localmente**.

Esto asegura que pueda tener docenas de aplicaciones en su computadora, todas ejecutando una versión diferente de cada paquete si es necesario.

Actualizar un paquete global haría que todos sus proyectos usen la nueva versión y, como puede imaginar, esto podría causar pesadillas en términos de mantenimiento, ya que algunos paquetes podrían romper la compatibilidad con otras dependencias, etc.

Todos los proyectos tienen su propia versión local de un paquete, aunque esto pueda parecer un desperdicio de recursos, es mínimo en comparación con las posibles consecuencias negativas.

Un paquete **debe instalarse globalmente** cuando proporciona un comando ejecutable que ejecuta desde el shell (CLI) y se reutiliza en todos los proyectos.

También puede instalar comandos ejecutables localmente y ejecutarlos usando `npx`, pero algunos paquetes se instalan mejor globalmente.

Excelentes ejemplos de paquetes globales populares que quizás conozca son

- `npm`
- `crear-reaccionar-app`
- `ver-cli`
- `gruñido-cli`
- `moca`
- `reaccionar-nativo-cli`
- `gatsby-cli`

- Siempre
- nodemonio

Probablemente ya tenga algunos paquetes instalados globalmente en su sistema. Puedes verlos corriendo

```
npm lista -g --profundidad 0
```

en su línea de comando.

# dependencias npm y devDependencies

## ¿Cuándo es un paquete una dependencia y cuándo es una dependencia de desarrollo?

Cuando instala un paquete npm usando `npm install <package-name>` , lo está instalando como una **dependencia**.

El paquete se incluye automáticamente en el [archivo package.json](#), en la lista de dependencias (a partir de npm 5: antes tenía que especificar manualmente `--save` ).

Cuando agrega el indicador `-D` o `--save-dev` , lo está instalando como una dependencia de desarrollo, lo que lo agrega a la lista de `devDependencies` .

Las dependencias de desarrollo están pensadas como paquetes solo de desarrollo, que no son necesarios en producción. Por ejemplo, paquetes de prueba, paquete [web](#) o [Babel](#).

Cuando entra en producción, si escribe `npm install` y la carpeta contiene un archivo `package.json` , se instalan, ya que npm asume que se trata de una implementación de desarrollo.

Debe configurar el indicador `--production` (`npm install --production`) para evitar instalar esas dependencias de desarrollo.

# npx

**npx es una forma genial de ejecutar el código de Node y proporciona muchas características útiles**

En esta publicación, quiero presentar un comando muy poderoso que ha estado disponible en [npm](#) a partir de la versión 5.2, lanzada en julio de 2017: **npx**.

Si no desea instalar npm, puede [instalar npx como un paquete independiente](#)

npx le permite ejecutar código creado con Node y publicado a través del registro npm.

## Ejecute fácilmente comandos locales

Los desarrolladores de nodos solían publicar la mayoría de los comandos ejecutables como paquetes globales, para que estuvieran en la ruta y fueran ejecutables de inmediato.

Esto fue un dolor porque realmente no podías instalar diferentes versiones del mismo comando.

Ejecutar npx commandname encuentra automáticamente la referencia correcta del comando dentro de la carpeta node\_modules de un proyecto, sin necesidad de conocer la ruta exacta y sin requerir el paquete que se instalará globalmente y en la ruta del usuario.

## Ejecución de comandos sin instalación

Hay otra gran característica de npm , que permite ejecutar comandos sin instalarlos primero.

Esto es bastante útil, principalmente porque:

1) no necesita instalar nada 2) puede ejecutar diferentes versiones del mismo comando, usando la sintaxis @version

Una demostración típica del uso de npx es a través del comando cowsay . cowsay imprimirá una vaca diciendo lo que escribiste en el comando. Por ejemplo:

cowsay "Hola" se imprimirá

```
< Hola >
-----
 \^__^
  \(oo)\_____(--)
   \||----w |
```

|| ||

Ahora, esto si tiene el comando cowsay instalado globalmente desde npm anteriormente; de lo contrario, obtendrá un error cuando intente ejecutar el comando.

npx le permite ejecutar ese comando npm sin tenerlo instalado localmente:

```
npx cowsay "Hola"
```

hará el trabajo.

Ahora, este es un comando divertido e inútil. Otros escenarios incluyen:

- ejecutando la herramienta vue CLI para crear nuevas aplicaciones y ejecutarlas: npx vue create my aplicación vue
- creando una nueva aplicación React usando create-react-app : npx create-react-app my-react-app

y muchos más.

Una vez descargado, el código descargado se borrará.

## Ejecute un código usando una versión de Nodo diferente

Use @ para especificar la versión y combínela con el paquete node npm:

```
nodo npx@6 -v #v6.14.3 nodo  
npx@8 -v #v8.11.3
```

Esto ayuda a evitar herramientas como nvm u otras herramientas de administración de versiones de Node.

## Ejecute fragmentos de código arbitrarios directamente desde una URL

npx no lo limita a los paquetes publicados en el registro de npm.

Puede ejecutar código que se encuentra en un [GitHub](#) esencia, por ejemplo:

```
npx https://gist.github.com/zkat/4bc19503fe9e9309e2bfaa2c58074d32
```

Por supuesto, debe tener cuidado al ejecutar código que no controla, ya que un gran poder conlleva una gran responsabilidad.



# El bucle de eventos

**El Event Loop es uno de los aspectos más importantes para entender sobre JavaScript. Esta publicación lo explica en términos simples.**

- [Introducción](#)
- [Bloqueo del bucle de eventos](#)
- [La pila de llamadas](#)
- [Una explicación simple del bucle de eventos](#)
- [Ejecución de la función de cola](#)
- [La cola de mensajes](#)
- [Cola de trabajos ES6](#)

## Introducción

El Event Loop es uno de los aspectos más importantes para entender sobre JavaScript.

He programado durante años con JavaScript, pero nunca he entendido *completamente* cómo funcionan las cosas **bajo el capó**. Está completamente bien no conocer este concepto en detalle, pero como de costumbre, es útil saber cómo funciona, y también podrías tener un poco de curiosidad en este punto.

Esta publicación tiene como objetivo explicar los detalles internos de cómo funciona JavaScript con un solo hilo y cómo maneja las funciones asíncronas.

Su código JavaScript se ejecuta en un solo subproceso. Sólo está sucediendo una cosa a la vez.

Esta es una limitación que en realidad es muy útil, ya que simplifica mucho la forma de programar sin preocuparse por los problemas de concurrencia.

Solo debe prestar atención a cómo escribe su código y evitar cualquier cosa que pueda bloquear el hilo, como llamadas de red síncronas o [bucles infinitos](#).

En general, en la mayoría de los navegadores hay un bucle de eventos para cada pestaña del navegador, para aislar cada proceso y evitar una página web con bucles infinitos o procesamiento pesado para bloquear su todo el navegador.

El entorno administra múltiples bucles de eventos simultáneos, para manejar llamadas API, por ejemplo. [Trabajadores web](#) también se ejecutan en su propio bucle de eventos.

Principalmente debe preocuparse de que *su código* se ejecute en un solo ciclo de eventos y escribir código con esto en mente para evitar bloquearlo.

## Bloqueo del bucle de eventos

Cualquier código JavaScript que tarde demasiado en devolver el control al bucle de eventos bloqueará la ejecución de cualquier código JavaScript en la página, incluso bloqueará el subproceso de la interfaz de usuario, y el usuario no podrá hacer clic, desplazarse por la página, etc.

Casi todas las primitivas de E/S en JavaScript son de no bloqueo. Solicitudes de red, [Node.js](#) operaciones del sistema de archivos, etc. El bloqueo es la excepción, y es por eso que JavaScript se basa tanto en las devoluciones de llamadas y, más recientemente, en las [promesas](#) . y [asíncrono/espera](#).

## La pila de llamadas

La pila de llamadas es una cola LIFO (último en entrar, primero en salir).

El bucle de eventos comprueba continuamente la **pila de llamadas** para ver si hay alguna función que necesite correr.

Mientras lo hace, agrega cualquier llamada de función que encuentre a la pila de llamadas y ejecuta cada una en ordenar.

¿Conoce el seguimiento de la pila de errores con el que puede estar familiarizado, en el depurador o en la consola del navegador?

El navegador busca los nombres de las funciones en la pila de llamadas para informarle qué función origina la llamada actual:

```
> const bar = () => {
    throw new DOMException()
}

const baz = () => console.log('baz')

const foo = () => {
    console.log('foo')
    bar()
    baz()
}

foo()
```

foo

✖ ▼ Uncaught DOMException

bar @ VM570:2

foo @ VM570:9

(anonymous) @ VM570:13

&gt; |

## Una explicación simple del bucle de eventos

Escojamos un ejemplo:

```
const barra = () => consola.log('barra')

const baz = () => consola.log('baz')

constante foo = () => {
    consola.log('foo') bar()
    baz()

}

foo()
```

Este código imprime

```
Foo
bar
base
```

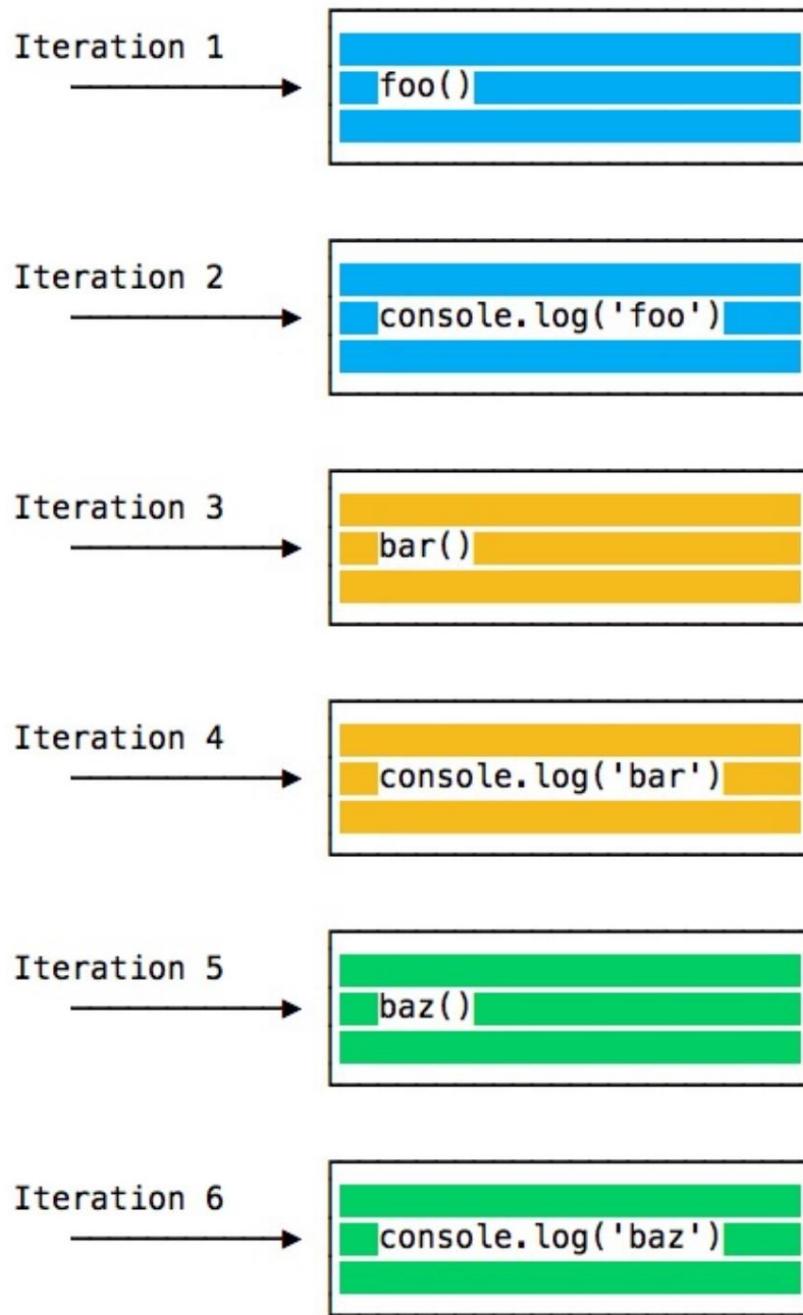
como se esperaba.

Cuando se ejecuta este código, primero se llama a `foo()`. Dentro de `foo()` primero llamamos a `bar()`, luego llamamos `base()`.

En este punto, la pila de llamadas se ve así:



El bucle de eventos en cada iteración busca si hay algo en la pila de llamadas y lo ejecuta:



hasta que la pila de llamadas esté vacía.

## Ejecución de la función de cola

El ejemplo anterior parece normal, no tiene nada de especial: JavaScript encuentra cosas para ejecutar, las ejecuta en orden.

Veamos cómo diferir una función hasta que la pila esté vacía.

El caso de uso de `setTimeout(() => {}), 0)` es llamar a una función, pero ejecutarla una vez cada dos segundos. La función en el código se ha ejecutado.

Toma este ejemplo:

```
const barra = () => consola.log('barra')

const baz = () => consola.log('baz')

const foo = () =>
  { console.log('foo')
    setTimeout(bar, 0) baz()

  }
  foo()
```

Este código se imprime, quizás sorprendentemente:

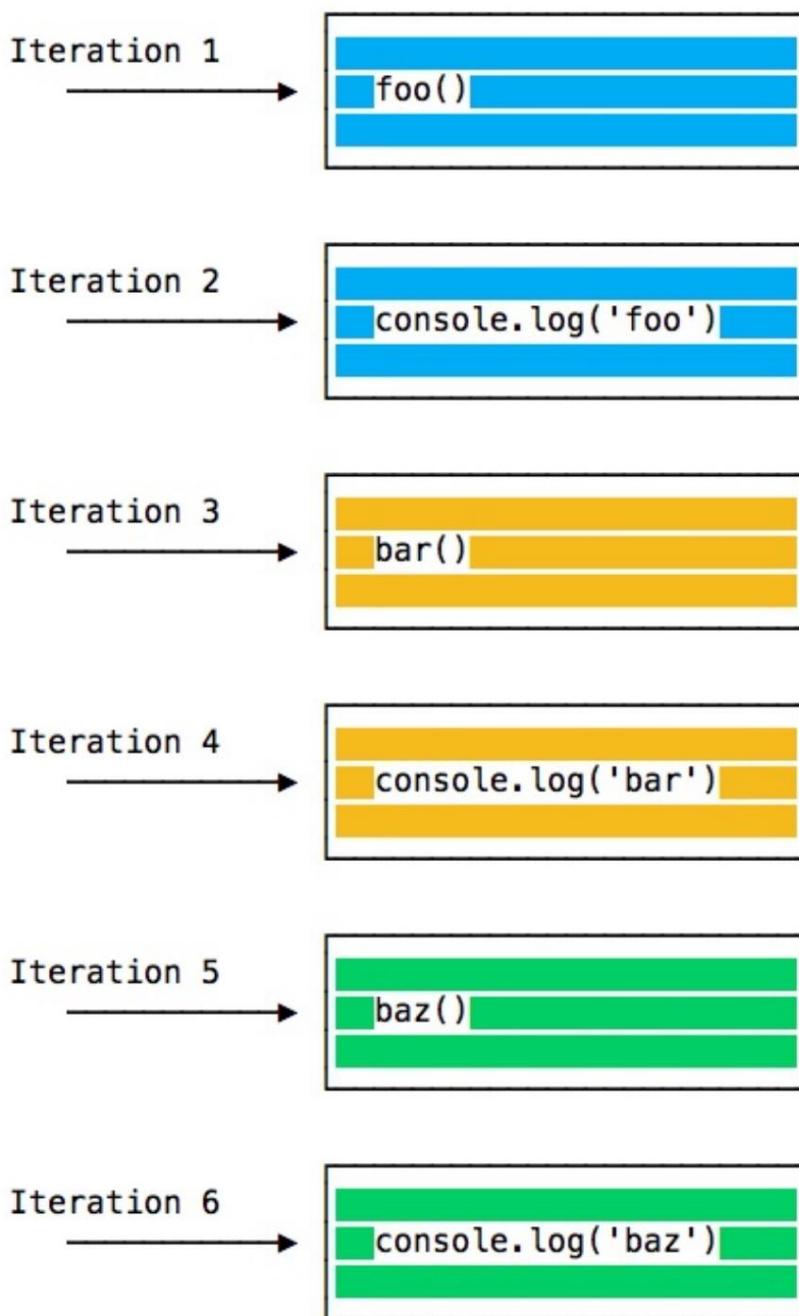
```
Foo
base
bar
```

Cuando se ejecuta este código, primero se llama a `foo()`. Dentro de `foo()` primero llamamos a `setTimeout`, pasando `bar` como argumento, y le indicamos que se ejecute inmediatamente lo más rápido posible, pasando `0` como temporizador. Luego llamamos a `baz()`.

En este punto, la pila de llamadas se ve así:



Este es el orden de ejecución de todas las funciones de nuestro programa:



¿Por qué está pasando esto?

## La cola de mensajes

Cuando se llama a `setTimeout()`, el navegador o Node.js inician el **temporizador**. Una vez que el temporizador expira, en este caso inmediatamente cuando ponemos 0 como tiempo de espera, la función de devolución de llamada se coloca en la **cola de mensajes**.

La Cola de mensajes también es donde los eventos iniciados por el usuario, como eventos de clic o teclado, o [buscar](#) las respuestas se ponen en cola antes de que su código tenga la oportunidad de reaccionar ante ellas. O también [DOM](#) eventos como `onLoad`.

**El ciclo da prioridad a la pila de llamadas, y primero procesa todo lo que encuentra en la pila de llamadas, y una vez que no hay nada ahí, va a recoger las cosas en el evento cola.**

No tenemos que esperar funciones como `setTimeout`, `fetch` u otras cosas para hacer su propio trabajo, porque los proporciona el navegador y viven en sus propios subprocesos. Por ejemplo, si establece el tiempo de espera de `setTimeout` en 2 segundos, no tiene que esperar 2 segundos, la espera ocurre en otro lugar.

## Cola de trabajos ES6

[ECMAScript 2015](#) introdujo el concepto de Job Queue, que es utilizado por Promises (también introducido en ES6/ES2015). Es una forma de ejecutar el resultado de una función asíncrona lo antes posible, en lugar de colocarla al final de la pila de llamadas.

Las promesas que se resuelven antes de que finalice la función actual se ejecutarán inmediatamente después de la actual función.

Me parece agradable la analogía de un paseo en montaña rusa en un parque de diversiones: la cola de mensajes te vuelve a poner en la cola después de todas las demás personas en la cola, mientras que la cola de trabajo es el boleto rápido que te permite tomar otro paseo justo después de que hayas terminado. El anterior.

Ejemplo:

```
const barra = () => consola.log('barra')

const baz = () => consola.log('baz')

const foo = () =>
  { console.log('foo')
    setTimeout(bar, 0)
    nueva Promesa((resolver, rechazar) =>
      resolve('debería estar justo después de baz, antes de bar')
    ).entonces(resolver => console.log(resolver)) baz()
  }
  foo()
```

esto imprime

```
Foo
```

```
base
```

```
debería ser justo después de baz, antes de bar  
bar
```

Esa es una gran diferencia entre Promises (y Async/await, que se basa en promesas) y las antiguas funciones asincrónicas a través de setTimeout() u otras API de plataforma.

## siguienteTick

### La función Node.js process.nextTick interactúa con el bucle de eventos de una manera especial

Mientras intenta comprender el [bucle de eventos de Node.js](#), una parte importante es `proceso.nextTick()`.

Cada vez que el ciclo de eventos realiza un viaje completo, lo llamamos un tic.

Cuando pasamos una función a `process.nextTick()`, le indicamos al motor que invoque esta función al final de la operación actual, antes de que comience el siguiente ciclo de eventos:

```
process.nextTick(() => { //hacer
  algo })
```

El bucle de eventos está ocupado procesando el código de función actual.

Cuando finaliza esta operación, el motor JS ejecuta todas las funciones pasadas a las llamadas `nextTick` durante esa operación.

Es la forma en que podemos decirle al motor JS que procese una función de forma asíncrona (después de la función actual), pero lo antes posible, sin ponerla en cola.

Llamar a `setTimeout(() => {}, 0)` ejecutará la función en el siguiente tick, mucho más tarde que cuando se usa `nextTick()`.

Usa `nextTick()` cuando quieras asegurarte de que en la próxima iteración del bucle de eventos ese código ya se haya ejecutado.

## establecerInmediato

### La función setImmediate de Node.js interactúa con el bucle de eventos de una manera especial

Cuando desee ejecutar algún fragmento de código de forma asíncrona, pero lo antes posible, una opción es utilizar la función `setImmediate()` proporcionada por Node.js:

```
setImmediate(() => { //  
  ejecutar algo })
```

Cualquier función pasada como argumento `setImmediate()` es una devolución de llamada que se ejecuta en la siguiente iteración del bucle de eventos.

¿En qué se diferencia `setImmediate()` de `setTimeout(() => {}, 0)` (pasando un tiempo de espera de 0 ms) y de `process.nextTick()` ?

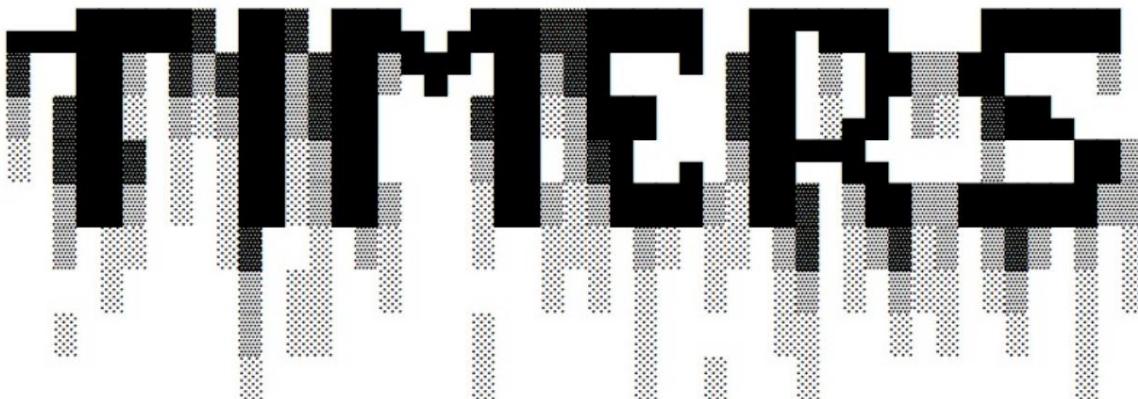
Una función pasada a `process.nextTick()` se ejecutará en la iteración actual del bucle de eventos, después de que finalice la operación actual. Esto significa que siempre se ejecutará antes

`setTimeout` y `setImmediate` .

Una devolución de llamada `setTimeout()` con un retraso de 0 ms es muy similar a `setImmediate()` . El orden de ejecución dependerá de varios factores, pero ambos se ejecutarán en la siguiente iteración del bucle de eventos.

## Temporizadores

Al escribir código JavaScript, es posible que desee retrasar la ejecución de una función. Aprenda a usar `setTimeout` y `setInterval` para programar funciones en el futuro



- `establecerTiempo de espera()`
  - Retraso cero
- `establecerIntervalo()`
- `setTimeout` recursivo

### `establecerTiempo de espera()`

Al escribir `JavaScript` código, es posible que desee retrasar la ejecución de una función.

Este es el trabajo de `setTimeout`. Usted especifica una función de devolución de llamada para ejecutar más tarde y un valor que expresa cuán tarde desea que se ejecute, en milisegundos:

```
establecerTiempo de espera(() => {
  // se ejecuta después de 2 segundos
}, 2000)

establecerTiempo de espera(() => {
  // se ejecuta después de 50 milisegundos
}, 50)
```

Esta sintaxis define una nueva función. Puede llamar a cualquier otra función que desee allí, o puede pasar un nombre de función existente y un conjunto de parámetros:

```
const myFunction = (firstParam, secondParam) => { // hacer algo
}
```

```
// se ejecuta después de 2 segundos
setTimeout(myFunction, 2000, firstParam, secondParam)
```

`setTimeout` devuelve la identificación del temporizador. Esto generalmente no se usa, pero puede almacenar esta identificación y borrar si desea eliminar la ejecución de esta función programada:

```
const id = establecerTiempo de espera () => {
  // debería ejecutarse después de 2 segundos
}, 2000)

// Cambié de opinión
clearTimeout(id)
```

## Retraso cero

Si especifica el retraso del tiempo de espera en 0 , la función de devolución de llamada se ejecutará tan pronto como es posible, pero después de la ejecución de la función actual:

```
setTimeout(() =>
  { console.log('después ') }, 0)

console.log(' antes de ')
```

se imprimirá antes después .

Esto es especialmente útil para evitar bloquear la CPU en tareas intensivas y permitir que se ejecuten otras funciones mientras se realiza un cálculo pesado, al poner en cola funciones en el programador.

Algunos navegadores (IE y Edge) implementan un **método** `setImmediate()` que hace exactamente la misma funcionalidad, pero no es estándar y [no está disponible en otros navegadores](#). Pero es una función estándar en Node.js.

## establecerIntervalo()

`setInterval` es una función similar a `setTimeout` , con una diferencia: en lugar de ejecutar el función de devolución de llamada una vez, la ejecutará para siempre, en el intervalo de tiempo específico que especifique (en milisegundos):

```
setInterval(() => { // se
  ejecuta cada 2 segundos }, 2000)
```

La función anterior se ejecuta cada 2 segundos a menos que le indique que se detenga, usando clearInterval , pasándole la identificación del intervalo que devolvió setInterval :

```
const id = setInterval(() => { // se ejecuta
    cada 2 segundos }, 2000)
```

```
clearInterval(id)
```

Es común llamar a clearInterval dentro de la función de devolución de llamada setInterval, para que se auto determinar si debe ejecutarse de nuevo o detenerse. Por ejemplo, este código ejecuta algo a menos que App.something|Wait haya llegado el valor :

```
intervalo constante = establecerIntervalo(() => {
    if (App.something|Wait === 'llegó')
        { clearInterval(intervalo)
        devolver

    } // de lo contrario haz las
    cosas }, 100)
```

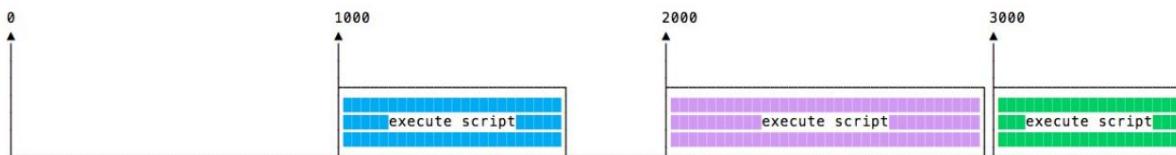
## setTimeout recursivo

setInterval inicia una función cada n milisegundos, sin tener en cuenta cuándo función terminó su ejecución.

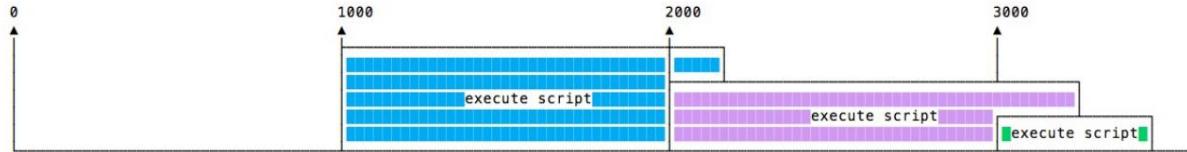
Si una función toma siempre la misma cantidad de tiempo, todo está bien:



Tal vez la función tome diferentes tiempos de ejecución, dependiendo de las condiciones de la red, por ejemplo:



Y tal vez una ejecución larga se superponga a la siguiente:



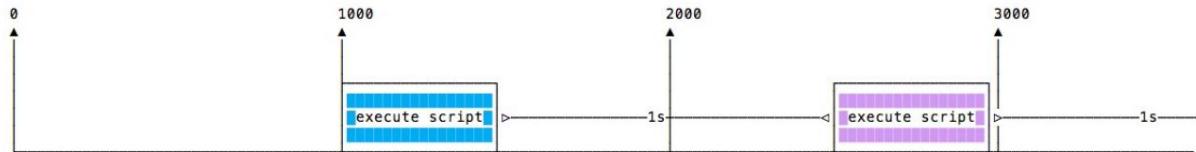
Para evitar esto, puede programar un setTimeout recursivo para que se llame cuando la función de devolución de llamada acabados:

```
const myFunction = () => { // hacer
    algo

    setTimeout(miFunción, 1000)
}

setTimeout( miFunción(), 1000)
```

para lograr este escenario:



`setTimeout` y `setInterval` están disponibles en [Node.js](#), a través del [módulo Temporizadores](#).

Node.js también proporciona `setImmediate()`, que es equivalente a usar `setTimeout(() => {}, 0)`, que se usa principalmente para trabajar con el bucle de eventos de Node.js.

## devoluciones de llamada

JavaScript es síncrono de forma predeterminada y tiene un solo subproceso. Esto significa que el código no puede crear nuevos subprocesos y ejecutarse en paralelo. Descubra qué significa el código asíncrono y cómo se ve



- Asincronía en lenguajes de programación
- JavaScript
- devoluciones de llamada
- Manejo de errores en devoluciones de llamada
- El problema con las devoluciones de llamada
- Alternativas a las devoluciones de llamada

## Asincronía en lenguajes de programación

Las computadoras son asíncronas por diseño.

Asíncrono significa que las cosas pueden suceder independientemente del flujo del programa principal.

En las computadoras de consumo actuales, cada programa se ejecuta durante un intervalo de tiempo específico, y luego detiene su ejecución para permitir que otro programa continúe su ejecución. Esta cosa se ejecuta en un ciclo tan rápido que es imposible notarlo, y creemos que nuestras computadoras ejecutan muchos programas simultáneamente, pero esto es una ilusión (excepto en las máquinas multiprocesador).

Los programas utilizan *interrupciones* internas , una señal que se emite al procesador para llamar la atención del sistema.

No entraré en detalles internos de esto, pero tenga en cuenta que es normal que los programas sean asíncronos y detener su ejecución hasta que necesiten atención, y la computadora puede ejecutar otras cosas mientras tanto. Cuando un programa está esperando una respuesta de la red, no puede detener el procesador hasta que finalice la solicitud.

Normalmente, los lenguajes de programación son síncronos y algunos proporcionan una forma de gestionar la asíncronía, en el lenguaje o mediante bibliotecas. C, Java, C#, PHP, Go, Ruby, Swift, Python, todos son síncronos de forma predeterminada. Algunos de ellos manejan asíncrono mediante el uso de subprocesos, lo que genera un nuevo proceso.

## JavaScript

JavaScript es **síncrono** por defecto y tiene un solo subproceso. Esto significa que el código no puede crear nuevos subprocesos y ejecutarse en paralelo.

Las líneas de código se ejecutan en serie, una tras otra, por ejemplo:

```
constante a = 1
constante b = 2
constante c = un * b
consola.log(c)
hacerAlgo()
```

Pero JavaScript nació dentro del navegador, su trabajo principal, al principio, era responder a las acciones del usuario, como onClick con un modelo de programación síncrona en Cambiar , onSubmit y así sucesivamente. ¿Cómo podría hacer esto?

La respuesta estaba en su entorno. El **navegador** proporciona una forma de hacerlo al proporcionar un conjunto de API que pueden manejar este tipo de funcionalidad.

Más recientemente, Node.js introdujo un entorno de E/S sin bloqueo para extender este concepto al acceso a archivos, llamadas de red, etc.

**devoluciones de llamada**

---

No puede saber cuándo un usuario va a hacer clic en un botón, por lo que lo que hace es **definir un controlador de eventos para el evento de clic**. Este controlador de eventos acepta una función, que se llamará cuando se active el evento:

```
document.getElementById('botón').addEventListener('clic', () => {
  //elemento en el que se hizo clic
})
```

Esta es la llamada **devolución de llamada**.

Una devolución de llamada es una función simple que se pasa como un valor a otra función y solo se ejecutará cuando ocurra el evento. Podemos hacer esto porque JavaScript tiene funciones de primera clase, que se pueden asignar a variables y pasar a otras funciones (llamadas **funciones de orden superior**)

Es común envolver todo su código de cliente en un detector de eventos de carga en el objeto de la ventana , que ejecuta la función de devolución de llamada solo cuando la página está lista:

```
ventana.addEventListener('cargar', () => {
  //ventana cargada
  //haz lo que quieras })
```

Las devoluciones de llamada se usan en todas partes, no solo en eventos DOM.

Un ejemplo común es mediante el uso de temporizadores:

```
establecerTiempo de espera() => {
  // se ejecuta después de 2 segundos
}, 2000)
```

Las solicitudes XHR también aceptan una devolución de llamada, en este ejemplo mediante la asignación de una función a una propiedad que se llamará cuando ocurra un evento en particular (en este caso, el estado de la solicitud cambia):

```
const xhr = new XMLHttpRequest()
xhr.onreadystatechange = () => {
  si (xhr. estado listo === 4) {
    xhr.status === 200 ? consola.log(xhr.responseText) : consola.error('error')
  }
}
xhr.open('GET', 'https://yoursite.com') xhr.send()
```

## Manejo de errores en devoluciones de llamada

¿Cómo maneja los errores con las devoluciones de llamada? Una estrategia muy común es usar lo que adoptó Node.js: el primer parámetro en cualquier función de devolución de llamada es el objeto de error: **error-primer devolución de llamada**

Si no hay ningún error, el objeto es nulo . Si hay un error, contiene alguna descripción del error y otra información.

```
fs.readFile('/file.json', (err, datos) => { if (err !== null) {

    // manejar el error
    consola.log(err)
    devolver
}

//sin errores, procesar datos
console.log(datos)})
```

## El problema con las devoluciones de llamada

¡Las devoluciones de llamada son excelentes para casos simples!

Sin embargo, cada devolución de llamada agrega un nivel de anidamiento, y cuando tiene muchas devoluciones de llamada, el código comienza a complicarse muy rápidamente:

```
ventana.addEventListener('cargar', () => {
    document.getElementById('botón').addEventListener('clic', () => {
        setTimeout(() =>
            { items.forEach(item => { //tu código
                aquí }) }, 2000) }))
```

Este es solo un código simple de 4 niveles, pero he visto muchos más niveles de anidamiento y no es divertido.

¿Cómo resolvemos esto?

## Alternativas a las devoluciones de llamada

A partir de ES6, JavaScript introdujo varias funciones que nos ayudan con el código asíncrono que no implica el uso de devoluciones de llamada:

- [Promesas \(ES6\)](#)
- [Asíncrono/ Espera \(ES8\)](#)



# promesas

**Las promesas son una forma de lidiar con el código asincrónico en JavaScript, sin escribir demasiadas devoluciones de llamada en su código.**

- [Introducción a las promesas](#)
  - [Cómo funcionan las promesas, en resumen](#)
  - [¿Qué promesas de uso de API JS?](#)
- [Crear una promesa](#)
- [Consumir una promesa](#)
- [Encadenamiento de](#)
  - [promesas Ejemplo de encadenamiento](#)
- [de promesas Manejo de errores Errores en](#)
  - [cascada Orquestación de promesas](#)
- [Promesa.todo\(\)](#)
  - [Promesa.carrera\(\)](#)
- [Errores comunes](#)
  - [TypeError no capturado: indefinido no es una promesa](#)

## Introducción a las promesas

Una promesa se define comúnmente como **un representante de un valor que eventualmente se convertirá disponible.**

Las promesas son una forma de lidiar con el código asincrónico, sin escribir demasiadas devoluciones de llamada en su código.

Aunque existen desde hace años, se han estandarizado e introducido en [ES2015](#), y ahora han sido reemplazados en [ES2017](#) por [funciones asíncronas](#).

**Las funciones asíncronas** usan la API de promesas como su bloque de construcción, por lo que comprenderlas es fundamental incluso si en el código más nuevo probablemente usará funciones asíncronas en lugar de promesas.

### Cómo funcionan las promesas, en resumen

Una vez que se ha llamado a una promesa, comenzará en **estado pendiente**. Esto significa que la función de la persona que llama continúa la ejecución, mientras espera que la promesa haga su propio procesamiento y le dé alguna retroalimentación a la función de la persona que llama.

En este punto, la función de la persona que llama espera que devuelva la promesa en un **estado resuelto** o en un **estado rechazado**, pero como sabe, [JavaScript](#) es asíncrono, por lo que *la función continúa su ejecución mientras la promesa lo hace*.

## ¿Qué promesas de uso de API JS?

Además de su propio código y el código de las bibliotecas, las promesas se utilizan en la Web moderna estándar. API como:

- API de batería
- la [API de obtención](#)
- [Trabajadores de servicios](#)

Es poco probable que en el JavaScript moderno *no* use promesas, así que comenzemos a profundizar en ellas.

---

## Creando una promesa

La API de Promise expone un constructor de Promise, que se inicializa con `new Promise()` :

```
dejar hecho = verdadero

const isItDoneYet = nueva promesa
( (resolver, rechazar) => { if (hecho) {

    const trabajoTerminado = 'Aquí está lo que construí'
    resolve(trabajoTerminado) } else { const por qué = 'Sigo
trabajando en otra cosa' rechazar(por qué)

    }
}
)
```

Como puede ver, la promesa verifica la constante global realizada y, si eso es cierto, devolvemos una promesa resuelta; de lo contrario, una promesa rechazada.

Usando `resolver` y `rechazar` podemos comunicar un valor, en el caso anterior solo devolvemos una cadena, pero también podría ser un objeto.

---

## Consumir una promesa

En la última sección, presentamos cómo se crea una promesa.

Ahora veamos cómo se puede *consumir* o usar la promesa.

```
const isItDoneYet = nueva promesa (
  //...
)

const comprobar si está hecho = () => {
  ya está hecho
  .entonces((bien) => {

    consola.log(ok) }) .catch((err) => {
      consola.error(err) })

}
```

Ejecutar `checkIfsDone()` ejecutará la promesa `isItDoneYet()` y esperará a que se resuelva, usando la devolución de llamada, y si hay un error, lo manejará en la captura llamar de vuelta.

## Encadenando promesas

Una promesa se puede devolver a otra promesa, creando una cadena de promesas.

La [API Fetch](#) proporciona un gran ejemplo de encadenamiento de promesas , una capa encima de la XMLHttpRequest API, que podemos usar para obtener un recurso y poner en cola una cadena de promesas para se ejecuta cuando se obtiene el recurso.

Fetch API es un mecanismo basado en promesas, y llamar a `fetch()` es equivalente a definir nuestra propia promesa usando `new Promise()` .

## Ejemplo de encadenamiento de promesas

```
const estado = (respuesta) => { if
  (respuesta.estado >= 200 && respuesta.estado < 300) {
    return Promise.resolve(respuesta)
  } return Promise.reject(nuevo Error(response.statusText))
}

const json = (respuesta) => respuesta.json()

buscar('/
  todos.json') .then(estado)
```

```
.then(json) .then((datos) => { console.log('Solicitud exitosa con respuesta JSON', datos) }) .catch((error) =>  
{ console.log('Solicitud fallida', error) })
```

En este ejemplo, llamamos a `fetch()` para obtener una lista de elementos TODO del archivo `todos.json` que se encuentra en la raíz del dominio, y creamos una cadena de promesas.

Ejecutar `fetch()` devuelve una `respuesta`, que tiene muchas propiedades, y dentro de ellas tenemos referencia:

- `estado` , un valor numérico que representa el código de estado HTTP
- `estadoTexto` , un mensaje de estado, que está bien si la solicitud tuvo éxito

`respuesta` también tiene un método `json()` , que devuelve una promesa que se resolverá con el contenido del cuerpo procesado y transformado en JSON.

Entonces dadas esas premisas, esto es lo que sucede: la primera promesa en la cadena es una función que definimos, llamada `status()` , que verifica el estado de la `respuesta` y si no es una respuesta exitosa (entre 200 y 299), rechaza la

Esta operación hará que la cadena de promesa omita todas las promesas encadenadas enumeradas y salte directamente a la instrucción `catch()` en la parte inferior, registrando el texto `Solicitud fallida` junto con el mensaje de error.

Si eso tiene éxito, llama a la función `json()` que definimos. Dado que la promesa anterior, cuando tuvo éxito, devolvió el objeto de respuesta , lo obtenemos como entrada para la segunda promesa.

En este caso, devolvemos los datos JSON procesados, por lo que la tercera promesa recibe el JSON directamente:

```
.then((datos) => {  
  console.log('Solicitud exitosa con respuesta JSON', datos) })
```

y simplemente lo registramos en la consola.

---

## Manejo de errores

En el ejemplo, en la sección anterior, teníamos una captura que se agregó a la cadena de promesas

Cuando algo en la cadena de promesas falla y genera un error o rechaza la promesa, el control va a la instrucción `catch()` más cercana en la cadena.

```
nueva Promesa((resolver, rechazar) =>
  { lanzar nuevo Error('Error')
})
  .catch((err) => { consola.error(err) })

// o

nueva Promesa((resolver, rechazar) =>
  { rechazar('Error')
})
  .catch((err) => { consola.error(err) })
```

## Errores en cascada

Si dentro de catch() genera un error, puede agregar un segundo catch() para manejarlo, y pronto.

```
nueva Promesa((resolver, rechazar) =>
  { lanzar nuevo Error('Error')
})
  .catch((err) => { lanzar un nuevo error('Error') }) .catch((err)
=> { console.error(err) })
```

## Orquestando promesas

### Promesa.todo()

Si necesita sincronizar diferentes promesas, Promise.all() lo ayuda a definir una lista de promesas y ejecutar algo cuando todas están resueltas.

Ejemplo:

```
const f1 = buscar('/algo.json') const f2 = buscar('/
algo2.json')

Promise.all([f1, f2]).then((res) => {
  console.log('Array de resultados', res)

}) .catch((error) => {
  consola.error(err) })
```

La tarea de desestructuración ES2015 la sintaxis también le permite hacer

```
Promise.all([f1, f2]).then(([res1, res2]) => {
```

```
    consola.log('Resultados', res1, res2)
})
```

Por supuesto, no está limitado a usar fetch , **cualquier promesa es buena.**

## Promesa.carrera()

Promise.race() se ejecuta cuando se resuelve la primera de las promesas que le pasas y ejecuta el devolución de llamada adjunta solo una vez, con el resultado de la primera promesa resuelta.

Ejemplo:

```
const primero = nueva Promesa((resolver, rechazar) =>
  { setTimeout(resolver, 500, 'primero')
})
const segundo = nueva Promesa((resolver, rechazar) =>
  { setTimeout(resolver, 100, 'segundo')
})

Promise.race([primero, segundo]).then((resultado) =>
  { console.log(resultado) // segundo })
```

## Errores comunes

### TypeError no capturado: indefinido no es una promesa

Si obtiene el TypeError no capturado: undefined no es un error de promesa en la consola, asegúrese de usar new Promise() en lugar de solo Promise()

## asíncrono/espera

**Descubra el enfoque moderno de las funciones asincrónicas en JavaScript. JavaScript evolucionó en muy poco tiempo desde las devoluciones de llamada a Promises y, desde ES2017, JavaScript asíncrono es aún más simple con la sintaxis `async/await`.**

- [Introducción](#)
- [¿Por qué se introdujeron `async/await`?](#)
- [Cómo funciona](#)
- [Un ejemplo rápido](#)
- [Promete todas las cosas](#)
- [El código es mucho más simple de leer.](#)
- [Múltiples funciones asíncronas en serie](#)
- [Depuración más fácil](#)

## Introducción

JavaScript evolucionó en muy poco tiempo de devoluciones de llamadas a [promesas](#) (ES2015), y desde [ES2017](#) JavaScript asíncrono es aún más simple con la sintaxis `async/await`.

Las funciones asíncronas son una combinación de promesas y [generadores](#), y básicamente, son una abstracción de mayor nivel sobre las promesas. Permítanme repetir: **`async/await` se basa en promesas**.

## ¿Por qué se introdujeron `async/await`?

Reducen el texto estándar en torno a las promesas y la limitación de "no romper la cadena" de encadenar promesas.

Cuando se introdujeron Promises en ES2015, estaban destinados a resolver un problema con el código asíncrono, y lo hicieron, pero durante los 2 años que separaron ES2015 y ES2017, quedó claro que las *promesas no podían ser la solución final*.

Las promesas se introdujeron para resolver el famoso problema del *infierno de devolución de llamada*, pero introdujeron complejidad por sí mismas y complejidad de sintaxis.

Eran buenas primitivas en torno a las cuales se podía exponer una mejor sintaxis a los desarrolladores, por lo que cuando llegó el momento adecuado obtuvimos **funciones asíncronas**.

Hacen que el código parezca síncrono, pero detrás es asíncrono y no bloquea las escenas.

## Cómo funciona

Una función asíncrona devuelve una promesa, como en este ejemplo:

```
const hacerAlgoAsync = () => {
  return new Promise((resolver) =>
    { setTimeout(() => resolver('Hice algo'), 3000)
  })
}
```

Cuando desee llamar a esta función, anteponga esperar a que la `Promise`, y el **código de llamada se detendrá hasta que la promesa se resuelva o rechace**. Una advertencia: la función de cliente debe definirse como

asíncrono \_ Aquí hay un ejemplo:

```
const hacerAlgo = asíncrono () => {
  console.log(espera hacerAlgoAsync())
}
```

## Un ejemplo rápido

Este es un ejemplo simple de `async/await` utilizado para ejecutar una función de forma asíncrona:

```
const hacerAlgoAsync = () => {
  return new Promise((resolver) =>
    { setTimeout(() => resolver('Hice algo'), 3000)
  })
}

const hacerAlgo = asíncrono () => {
  console.log(espera hacerAlgoAsync())
}

consola.log('Antes')
hacerAlgo()
consola.log('Después')
```

El código anterior imprimirá lo siguiente en la consola del navegador:

```
Antes
Después
Hice algo //después de 3s
```

## Promete todas las cosas

Anteponer la palabra clave `async` a cualquier función significa que la función devolverá una promesa.

Incluso si no lo hace explícitamente, internamente hará que devuelva una promesa.

Es por eso que este código es válido:

```
const aFunction = asíncrono () => {
  devolver 'prueba'
}

aFunction().then(alerta) // Esto alertará a 'prueba'
```

y es lo mismo que:

```
const aFunction = asíncrono () => {
  return Promesa.resolve('prueba')
}

aFunction().then(alerta) // Esto alertará a 'prueba'
```

## El código es mucho más simple de leer.

Como puede ver en el ejemplo anterior, nuestro código parece muy simple. Compárelo con el código usando promesas simples, con funciones de encadenamiento y devolución de llamada.

Y este es un ejemplo muy simple, los mayores beneficios surgirán cuando el código sea mucho más complejo.

Por ejemplo, así es como obtendría un recurso JSON y lo analizaría usando promesas:

```
const getFirstUserData = () => {
  return fetch('/users.json') // obtener la lista de usuarios
    .then(response => response.json()) // analiza JSON .then(users =>
    users[0]) // selecciona el primer usuario .then(user => fetch('/users/
    ${user.name}')) // obtener datos de usuario .then(userResponse => response.json()) //
    analizar JSON
}

getFirstUserData()
```

Y aquí está la misma funcionalidad provista usando `await/async`:

```
const getFirstUserData = asíncrono () => {
  const respuesta = await fetch('/users.json') // obtener la lista de usuarios const users
  = await respuesta.json() // analizar JSON const user = usuarios[0] // seleccionar el
  primer usuario const userResponse = await fetch(` /users/${user.name}`) // obtener
  datos de usuario
```

```

const userData = esperar usuario.json() // analizar JSON
return userData
}

getFirstUserData ()

```

## Múltiples funciones asíncronas en serie

Las funciones asíncronas se pueden encadenar muy fácilmente, y la sintaxis es mucho más legible que con simples promesas:

```

const prometeHacerAlgo = () => { return new
    Promise(resolve => {
        setTimeout(() => resolve('Hice algo'), 10000)
    })
}

const mirarSobreAlguienHaciendoAlgo = asíncrono () => { const algo
    = esperar prometerHacerAlgo() devolver algo +
        'y vi'
}

const vigilarSobreAlguienMirandoAlguienHaciendoAlgo = async () => {
    const algo = aguardar verOverAlguienHaciendoAlgo() devolver algo +
        'y yo también vi'
}

vigilar a alguien viendo a alguien haciendo algo().then((res) => { console.log(res)
})

```

Imprimirá:

```
Hice algo y observé y también observé
```

## Depuración más fácil

La depuración de promesas es difícil porque el depurador no pasará por encima del código asíncrono.

Async/await lo hace muy fácil porque para el compilador es como un código síncrono.

# El emisor de eventos del nodo

## Cómo trabajar con eventos personalizados en Node

Si trabajó con JavaScript en el navegador, sabe cuánto de la interacción del usuario se maneja a través de eventos: clics del mouse, pulsaciones de botones del teclado, reacciones a los movimientos del mouse, etc.

En el lado del backend, Node nos ofrece la opción de construir un sistema similar usando los [eventos módulo](#).

Este módulo, en particular, ofrece la clase `EventEmitter`, que usaremos para manejar nuestros eventos.

Inicializas eso usando

```
const eventEmitter = require('events').EventEmitter()
```

Este objeto expone, entre muchos otros, los métodos `on` y `emit`.

- `emit` se usa para desencadenar un evento
- `on` se usa para agregar una función de devolución de llamada que se ejecutará cuando el evento sea motivado

Por ejemplo, vamos a crear un evento de inicio, y como una cuestión de proporcionar una muestra, reaccionamos simplemente iniciando sesión en la consola:

```
eventEmitter.on('inicio', () =>
{ console.log('iniciado') })
```

cuando corremos

```
eventEmitter.emit('inicio')
```

se activa la función del controlador de eventos y obtenemos el registro de la consola.

Puede pasar argumentos al controlador de eventos pasándolos como argumentos adicionales a `emitir()`:

```
eventEmitter.on('inicio', (número) =>
{ console.log(`comenzó ${número}`) })
```

```
eventEmitter.emit('inicio', 23)
```

Múltiples argumentos:

```
eventEmitter.on('inicio', (inicio, fin) => {
  console.log(`comenzó desde ${start} hasta ${end}`))

eventEmitter.emit('inicio', 1, 100)
```

El objeto EventEmitter también expone varios otros métodos para interactuar con eventos, como

- `once()` : agrega un detector de una vez
- `sola vez removeListener() / off()` : elimina un detector de eventos de un evento
- `removeAllListeners()` : elimina todos los detectores de un evento

Puede leer todos sus detalles en la página del módulo de eventos en <https://nodejs.org/api/events.html>

# HTTP

## Una descripción detallada de cómo funcionan el protocolo HTTP y la Web.

HTTP (*Protocolo de transferencia de hipertexto*) es uno de los protocolos de aplicación de TCP/IP, el conjunto de protocolos que impulsa Internet.

Déjeme arreglar eso: no es *uno* de los protocolos, es el más exitoso y popular, por todos medio.

HTTP es lo que hace que la World Wide Web funcione, brindando a los navegadores un lenguaje para comunicarse con servidores remotos que alojan páginas web.

HTTP se estandarizó por primera vez en 1991, como resultado del trabajo que Tim Berners-Lee realizó en el CERN, el Centro Europeo de Investigación Nuclear, desde 1989.

El objetivo era permitir a los investigadores intercambiar e interconectar fácilmente sus artículos. Fue pensado como una forma para que la comunidad científica trabajara mejor.

En aquel entonces, las principales aplicaciones de Internet consistían básicamente en FTP (el Protocolo de transferencia de archivos), correo electrónico y Usenet (grupos de noticias, hoy casi abandonados).

En 1993, se lanzó Mosaic, el primer navegador web gráfico, y las cosas se dispararon desde allá.

La Web se convirtió en la aplicación asesina de Internet.

Con el tiempo, la Web y el ecosistema que la rodea han evolucionado drásticamente, pero los conceptos básicos aún permanecen. Un ejemplo de evolución: HTTP ahora impulsa, además de las páginas web, las API REST, una forma común de acceder mediante programación a un servicio a través de Internet.

HTTP recibió una revisión menor en 1997 con HTTP/1.1, y en 2015 su sucesor, HTTP/2, se estandarizó y ahora está siendo implementado por los principales servidores web utilizados en todo el mundo.

El protocolo HTTP se considera inseguro, al igual que cualquier otro protocolo (SMTP, FTP...) que no se sirva a través de una conexión cifrada. Esta es la razón por la cual hay un gran impulso hoy en día hacia el uso de HTTPS, que es HTTP servido sobre TLS.

Dicho esto, los componentes básicos de HTTP/2 y HTTPS tienen sus raíces en HTTP, y en este artículo Voy a presentar cómo funciona HTTP.

## documentos HTML

HTTP es la forma **en que los navegadores web** como Chrome, Firefox, Edge y muchos otros (también llamados *clientes* de ahora en adelante) se comunican con **los servidores web**.

El nombre Protocolo de transferencia de hipertexto deriva de la necesidad de transferir no solo archivos, como en FTP, el "Protocolo de transferencia de archivos", sino hipertextos, que se escribirían usando HTML y luego se representarían gráficamente por el navegador con una presentación agradable e interactiva. Enlaces.

Los enlaces fueron la fuerza motriz que impulsó la adopción, junto con la facilidad de creación de nuevos sitios web.  
paginas

HTTP es lo que transfiere esos archivos de hipertexto (y como veremos también imágenes y otros tipos de archivos) sobre la red.

## hipervínculos

Dentro de un navegador web, un documento puede apuntar a otro documento mediante enlaces.

Un enlace está compuesto por una primera parte que determina el protocolo y la dirección del servidor, ya sea a través de un nombre de dominio o una IP.

Esta parte no es exclusiva de HTTP, por supuesto.

Luego está la parte del documento. Todo lo que se agregue a la parte de la dirección representa la ruta del documento.

Por ejemplo, la dirección de este documento es <https://flaviocopes.com/http/> :

- https es el protocolo.
- flaviocopes.com es el nombre de dominio que apunta a mi
- servidor /http/ es la URL del documento relativa a la ruta raíz del servidor.

La ruta se puede anidar: <https://flaviocopes.com/page/privacy/> y en este caso la URL del documento es /page/privacy .

El servidor web es el encargado de interpretar la solicitud y, una vez analizada, servir la respuesta correcta.

## Una solicitud

¿Qué hay en una solicitud?

Lo primero es **la URL**, que ya hemos visto antes.

Cuando ingresamos una dirección y presionamos enter en nuestro navegador, bajo el capó, el servidor envía a la dirección IP correcta una solicitud como esta:

OBTENER /a-página

donde /a-page es la URL que solicitó.

Lo segundo es el **método HTTP** (también llamado verbo).

HTTP en los primeros días definía 3 de ellos:

- OBTENER
- CORREO
- CABEZA

y HTTP/1.1 introducido

- PONER
- ELIMINAR
- OPCIONES
- RASTRO

Los veremos en detalle en un minuto.

Lo tercero que compone una solicitud es un conjunto de **encabezados HTTP**.

Los encabezados son un conjunto de claves: pares de valores que se utilizan para comunicar al servidor información específica que está predefinida, para que el servidor pueda saber a qué nos referimos.

Los describí en detalle en [la lista de encabezados de solicitud HTTP](#).

Dale a esa lista un vistazo rápido. Todos esos encabezados son opcionales, excepto Host .

## Métodos HTTP

OBTENER

GET es el método más utilizado aquí. Es el que se usa cuando escribe una URL en la barra de direcciones del navegador o cuando hace clic en un enlace.

Le pide al servidor que envíe el recurso solicitado como respuesta.

CABEZA

HEAD es como GET, pero le dice al servidor que no devuelva el cuerpo de la respuesta. Solo el encabezados

## CORREO

El cliente utiliza el método POST para enviar datos al servidor. Por lo general, se usa en formularios, por ejemplo, pero también cuando interactúa con una API REST.

## PONER

El método PUT está destinado a crear un recurso en esa URL específica, con los parámetros pasados en el cuerpo de la solicitud. Utilizado principalmente en API REST

## ELIMINAR

Se llama al método DELETE contra una URL para solicitar la eliminación de ese recurso. Principalmente utilizado en las API REST

## OPCIONES

Cuando un servidor recibe una solicitud de OPCIONES, debe devolver la lista de métodos HTTP permitidos para esa URL específica.

## RASTRO

Devuelve al cliente la solicitud que ha recibido. Se utiliza con fines de depuración o diagnóstico.

# Comunicación cliente/servidor HTTP

HTTP, como la mayoría de los protocolos que pertenecen a la suite TCP/IP, es un protocolo *sin estado*.

Los servidores no tienen idea de cuál es el estado actual del cliente. Todo lo que les importa es que reciben solicitudes y deben cumplirlas.

Cualquier solicitud previa no tiene sentido en este contexto, y esto hace posible que un servidor web sea muy rápido, ya que hay menos para procesar, y también le da ancho de banda para manejar una gran cantidad de solicitudes concurrentes.

HTTP también es muy simple y la comunicación es muy rápida en términos de gastos generales. Esto contrasta con los protocolos que más se usaban cuando se introdujo HTTP: TCP y POP/SMTP, los protocolos de correo, que implican muchos protocolos de enlace y confirmaciones en los extremos receptores.

Los navegadores gráficos abstraen toda esta comunicación, pero la ilustraremos aquí para aprender propósitos

Un mensaje está compuesto por una primera línea, que comienza con el método HTTP, luego contiene la ruta relativa al recurso y la versión del protocolo:

```
OBTENER /a-página HTTP/1.1
```

Después de eso, debemos agregar los encabezados de solicitud HTTP. Como se mencionó anteriormente, hay muchos encabezados, pero el único obligatorio es Host :

```
OBTENER /a-page HTTP/  
1.1 Host: flaviocopes.com
```

¿Cómo puedes probar esto? Uso **de telnet**. Esta es una herramienta de línea de comandos que nos permite conectarnos a cualquier servidor y enviarle comandos.

Abre tu terminal y escribe telnet flaviocopes.com 80

Esto abrirá una terminal, que te dice

```
Probando 178.128.202.129...  
Conectado a flaviocopes.com.  
El carácter de escape es '^J'.
```

Estás conectado al servidor web Netlify que alimenta mi blog. Ahora puede escribir:

```
OBTENER /axios/HTTP/1.1  
Anfitrío: flaviocopes.com
```

y presione enter en una línea vacía para activar la solicitud.

La respuesta será:

```
HTTP/1.1 301 Movido permanentemente  
Control de caché: public, max-age=0, must-revalidate Content-  
Length: 46 Content-Type: text/plain Fecha: domingo, 29 de julio de  
2018 14:07:07 GMT Ubicación: https://flaviocopes.com/axios/  
Edad: 0 Conexión: keep-alive Servidor: Netlify
```

```
Redirigiendo a https://flaviocopes.com/axios/
```

Mira, esta es una respuesta HTTP que obtuvimos del servidor. Es una solicitud 301 de mudanza permanente. Consulte la [lista de códigos de estado HTTP](#) para obtener más información sobre los códigos de estado.

Básicamente nos dice que el recurso se ha movido permanentemente a otra ubicación.

¿Por qué? Porque nos conectamos al puerto 80, que es el predeterminado para HTTP, pero en mi servidor configuré una redirección automática a HTTPS.

La nueva ubicación se especifica en el encabezado de respuesta HTTP de ubicación .

Hay otros encabezados, todos descritos en [la lista de encabezados de respuesta HTTP](#).

Tanto en la solicitud como en la respuesta, una línea vacía separa el encabezado de la solicitud del cuerpo de la solicitud. El cuerpo de la solicitud en este caso contiene la cadena

```
Redirigiendo a https://flaviocopes.com/axios/
```

que tiene una longitud de 46 bytes, como se especifica en el encabezado Content-Length . Se muestra en el navegador cuando abres la página, mientras te redirige automáticamente a la ubicación correcta.

En este caso estamos usando telnet, la herramienta de bajo nivel que podemos usar para conectarnos a cualquier servidor, por lo que no podemos tener ningún tipo de redireccionamiento automático.

Hagamos este proceso nuevamente, ahora conectándonos al puerto 443, que es el puerto predeterminado del protocolo HTTPS. No podemos usar telnet debido al protocolo de enlace SSL que debe ocurrir.

Simplifiquemos las cosas y usemos curl en la solicitud , otra herramienta de línea de comandos. No podemos escribir directamente HTTP, pero veremos la respuesta:

```
curl -i https://flaviocopes.com/axios/
```

esto es lo que obtendremos a cambio:

```
HTTP/1.1 200 Aceptar
Cache-Control: public, max-age=0, must-revalidate
Tipo de contenido: texto/html; conjunto de caracteres = UTF-8
Fecha: domingo, 29 de julio de 2018 14:20:45 GMT
Etag: "de3153d6eacef2299964de09db154b32-ssl"
Seguridad estricta en el transporte: max-age=31536000
Edad: 152
Longitud del contenido: 9797
Conexión: mantener vivo
Servidor: Netlify

<!DOCTYPE html>
<prefijo html = "og: http://ogp.me/ns#" lang = "en">
<cabeza>
<juego de caracteres meta="utf-8">
```

```
<meta http-equiv="X-UA-Compatible" content="IE=edge">
<title>Solicitudes HTTP usando Axios</title>
....
```

Corté la respuesta, pero puede ver que ahora se devuelve el HTML de la página.

## Otros recursos

Un servidor HTTP no solo transferirá archivos HTML, sino que normalmente también servirá otros archivos: CSS, JS, SVG, PNG, JPG, muchos tipos de archivos diferentes.

Esto depende de la configuración.

HTTP también es perfectamente capaz de transferir esos archivos, y el cliente conocerá el tipo de archivo, por lo que los interpretará de la manera correcta.

Así es como funciona la web: cuando el navegador recupera una página HTML, se interpreta y cualquier otro recurso que necesite para mostrar la propiedad (CSS, JavaScript, imágenes...) se recupera a través de solicitudes HTTP adicionales al mismo servidor.

# Cómo funcionan las solicitudes HTTP

## Qué sucede cuando escribes una URL en el navegador, de principio a fin

- [El protocolo HTTP](#)
- [Solo analizo solicitudes de URL](#)
- Las cosas se relacionan con macOS / Linux
- Fase de búsqueda de DNS
  - [gethostbyname](#)
- Apretón de manos de solicitud de TCP
- [Enviando la solicitud](#)
  - [La línea de solicitud](#)
  - [El encabezado de la solicitud](#)
  - [El cuerpo de la solicitud](#)
- [La respuesta](#)
- [Analizar el HTML](#)

Este artículo describe cómo los navegadores realizan solicitudes de página utilizando el protocolo HTTP/1.1

Si alguna vez hiciste una entrevista, es posible que te hayan preguntado: "¿Qué sucede cuando escribes algo en el cuadro de búsqueda de Google y presionas Intro".

Es una de las preguntas más populares que te hacen. La gente solo quiere ver si puedes explicar algunos conceptos bastante básicos y si tienes alguna idea de cómo funciona realmente Internet.

En esta publicación, analizaré lo que sucede cuando escribes una URL en la barra de direcciones de tu navegador y presionas enter.

Es un tema muy interesante para diseccionar en una publicación de blog, ya que toca muchas tecnologías en las que puedo sumergirme. en publicaciones separadas.

Esta es una tecnología que rara vez cambia y alimenta uno de los ecosistemas más complejos y amplios jamás construidos por humanos.

## El protocolo HTTP

Primero, menciono HTTPS en particular porque las cosas son diferentes de una conexión HTTPS.

## Solo analizo solicitudes de URL

Los navegadores modernos tienen la capacidad de saber si lo que escribió en la barra de direcciones es una URL real o un término de búsqueda, y usarán el motor de búsqueda predeterminado si no es una URL válida.

Supongo que escribe una URL real.

Cuando ingresa la URL y presiona enter, el navegador primero crea la URL completa.

Si acaba de ingresar un dominio, como flaviocopes.com , el navegador por defecto antepondrá HTTP:// a él, de forma predeterminada en el protocolo HTTP.

## Las cosas se relacionan con macOS / Linux

Solo para tu información. Windows puede hacer algunas cosas de manera ligeramente diferente.

### Fase de búsqueda de DNS

El navegador inicia la búsqueda de DNS para obtener la dirección IP del servidor.

El nombre de dominio es un atajo útil para nosotros los humanos, pero Internet está organizado de tal manera que las computadoras pueden buscar la ubicación exacta de un servidor a través de su dirección IP, que es un conjunto de números como 222.324.3.1 (IPv4).

Primero, verifica el caché local de DNS para ver si el dominio ya se resolvió recientemente.

Chrome tiene un práctico visualizador de caché de DNS que puede ver en <chrome://net-internals/#dns>

Si no se encuentra nada allí, el navegador utiliza la resolución de DNS, utilizando la llamada al sistema POSIX gethostbyname para recuperar la información del host.

### gethostbyname

gethostbyname primero busca en el archivo de hosts locales, que en macOS o Linux se encuentra en /etc/hosts , para ver si el sistema proporciona la información localmente.

Si esto no da ninguna información sobre el dominio, el sistema realiza una solicitud al Servidor DNS.

La dirección del servidor DNS se almacena en las preferencias del sistema.

Esos son 2 servidores DNS populares:

- 8.8.8.8 : el servidor DNS público de Google
- 1.1.1.1 : el servidor DNS de CloudFlare

La mayoría de las personas utilizan el servidor DNS proporcionado por su proveedor de Internet.

El navegador realiza la solicitud de DNS utilizando el protocolo UDP.

TCP y UDP son dos de los protocolos fundamentales de las redes informáticas. Se ubican en el mismo nivel conceptual, pero TCP está orientado a la conexión, mientras que UDP es un protocolo sin conexión, más liviano, que se usa para enviar mensajes con poca sobrecarga.

La forma en que se realiza la solicitud UDP no está dentro del alcance de este tutorial.

El servidor DNS podría tener la IP del dominio en la memoria caché. Si no, le preguntará al **servidor DNS raíz**.

Es un sistema (compuesto por 13 servidores reales, distribuidos por todo el planeta) que impulsa el Internet completo.

El servidor DNS *no* conoce la dirección de todos y cada uno de los nombres de dominio del planeta.

Lo que sabe es dónde están **los resolutores de DNS de nivel superior**.

Un dominio de nivel superior es la extensión de dominio: .com , .es , .pizza y así sucesivamente.

Una vez que el servidor DNS raíz recibe la solicitud, la reenvía a ese servidor DNS de dominio de nivel superior (TLD).

Digamos que está buscando flaviocopes.com . El servidor DNS del dominio raíz devuelve la IP del servidor TLD .com.

Ahora nuestro sistema de resolución de DNS almacenará en caché la IP de ese servidor TLD, por lo que no tendrá que volver a solicitarla al servidor DNS raíz.

El servidor DNS de TLD tendrá las direcciones IP de los servidores de nombres autorizados para el dominio que estamos buscando.

¿Cómo? Cuando compra un dominio, el registrador de dominios envía el TDL correspondiente a los servidores de nombres. Cuando actualice los servidores de nombres (por ejemplo, cuando cambie el proveedor de alojamiento), su registrador de dominio actualizará automáticamente esta información.

Esos son los servidores DNS del proveedor de alojamiento. Suelen ser más de 1, para que sirvan de respaldo.

Por ejemplo:

- ns1.dreamhost.com
- ns2.dreamhost.com
- ns3.dreamhost.com

El solucionador de DNS comienza con el primero e intenta solicitar la IP del dominio (con el subdominio también) que está buscando.

Esa es la última fuente de verdad para la dirección IP.

Ahora que tenemos la dirección IP, podemos continuar con nuestro viaje.

## Apretón de manos de solicitud de TCP

Con la dirección IP del servidor disponible, ahora el navegador puede iniciar una conexión TCP con eso.

Una conexión TCP requiere un poco de negociación antes de que pueda inicializarse por completo y pueda comenzar a enviar datos.

Una vez establecida la conexión, podemos enviar la solicitud

## Enviando la solicitud

La solicitud es un documento de texto plano estructurado de forma precisa determinada por el protocolo de comunicación.

Está compuesto por 3 partes:

- la línea de la solicitud
- el encabezado de la solicitud
- el cuerpo de la solicitud

### La línea de solicitud

La línea de solicitud establece, en una sola línea:

- el método HTTP
- la ubicación del recurso
- la versión del protocolo

Ejemplo:

OBTENER/HTTP/1.1

### El encabezado de la solicitud

El encabezado de la solicitud es un conjunto de campos: pares de valores que establecen ciertos valores.

Hay 2 campos obligatorios, uno de los cuales es Host , otros [redacted] , y el otro es Conexión [redacted] , mientras todos los campos son opcionales:

Anfitrión: flaviocopes.com

Conexión: cerrar

Host indica el nombre de dominio al que queremos apuntar, mientras que Connection siempre está configurado para cerrarse a menos que la conexión deba mantenerse abierta.

Algunos de los campos de encabezado más utilizados son:

- Origen
- Aceptar
- Aceptar-Codificación
- Galleta
- Control de caché
- Dnt

pero existen muchos más.

La parte del encabezado termina con una línea en blanco.

## El cuerpo de la solicitud

El cuerpo de la solicitud es opcional, no se usa en las solicitudes GET, pero se usa mucho en las solicitudes POST y, a veces, también en otros verbos, y puede contener datos en formato JSON.

Dado que ahora estamos analizando una solicitud GET, el cuerpo está en blanco y no lo analizaremos más.

## La respuesta

Una vez que se envía la solicitud, el servidor la procesa y devuelve una respuesta.

La respuesta comienza con el código de estado y el mensaje de estado. Si la solicitud es exitosa y devuelve un 200, comenzará con:

200 bien

La solicitud puede devolver un código de estado y un mensaje diferentes, como uno de estos:

404 No encontrado  
403 Prohibido  
301 Movido Permanentemente  
Error interno de servidor 500  
304 No modificado  
401 no autorizado

La respuesta luego contiene una lista de encabezados HTTP y el cuerpo de la respuesta (que, dado que estamos realizando la solicitud en el navegador, será HTML)

## Anализar el HTML

El navegador ahora ha recibido el HTML y comienza a analizarlo, y repetirá exactamente el mismo proceso que no hicimos para todos los recursos requeridos por la página:

- archivos CSS
- imágenes
- el favicon
- Archivos JavaScript
- ...

La forma en que los navegadores representan la página está fuera del alcance, pero es importante comprender que el proceso que describí no es solo para las páginas HTML, sino para cualquier elemento que se sirva. HTTP.

# Construir un servidor HTTP

## Cómo construir un servidor HTTP con Node.js

Aquí está el servidor web HTTP que usamos como la aplicación Node Hello World en [Node.js](#)

### Introducción

```
constante http = require('http')

puerto constante = 3000

const servidor = http.createServer((req, res) => {
  res.statusCode = 200
  res.setHeader('Content-Type', 'text/plain') res.end('Hello
  World\n')
})

servidor.escucha(puerto, () => {
  console.log(`Servidor ejecutándose en http://${hostname}:${port}/` ) })
```

Analicémoslo brevemente. Incluimos el módulo [http](#) .

Usamos el módulo para crear un servidor HTTP.

El servidor está configurado para escuchar en el puerto especificado, 3000 . Cuando el servidor está listo, la escucha se llama a la función de devolución de llamada.

La función de devolución de llamada que pasamos es la que se ejecutará con cada solicitud que ingrese. Cada vez que se recibe una nueva solicitud, se llama al [evento de solicitud](#) , proporcionando dos objetos: una solicitud (un objeto [http.IncomingMessage](#) ) y una respuesta ( un objeto [http.ServerResponse](#) ).

[request](#) proporciona los detalles de la solicitud. A través de ella accedemos a las cabeceras de las solicitudes y solicitamos datos.

[La respuesta](#) se utiliza para completar los datos que vamos a devolver al cliente.

En este caso con

```
res.statusCode = 200
```

establecemos la propiedad statusCode en 200, para indicar una respuesta exitosa.

También configuramos el encabezado Content-Type:

```
res.setHeader('Tipo de contenido', 'texto/simple')
```

y terminamos de cerrar la respuesta, agregando el contenido como argumento a end() :

```
res.end('Hola Mundo\n')
```

## Realización de solicitudes HTTP

### Cómo realizar solicitudes HTTP con Node.js usando GET, POST, PUT y DELETE

Uso el término HTTP, pero HTTPS es lo que debe usarse en todas partes, por lo tanto, estos ejemplos usan HTTPS en lugar de HTTP.

## Realizar una solicitud GET

```
const https = require('https') const opciones
= {
  nombre de host: 'flaviocopes.com',
  puerto: 443, ruta: '/todos', método: 'GET'

}

const req = https.request(opciones, (res) =>
{ console.log(`statusCode: ${res.statusCode}`)

  res.on('data', (d) => {
    proceso.stdout.escribir(d) }}))

req.on('error', (error) =>
{ consola.error(error) })

req.end()
```

## Realizar una solicitud POST

```
constante https = require('https')

const data = JSON.stringify({ todo:
  'Comprar la leche'
})

const options = { nombre
  de host: 'flaviocopes.com', puerto: 443,
  ruta: '/todos', método: 'POST',
  encabezados: {
```

```
'Tipo de contenido': 'aplicación/json', 'Longitud  
del contenido': data.length  
}  
}  
  
const req = https.request(opciones, (res) => {  
    console.log(`statusCode: ${res.statusCode}`)  
  
    res.on('data', (d) => {  
        proceso.stdout.escribir(d) }) })  
  
  
req.on('error', (error) =>  
{ consola.error(error) })  
  
req.write(datos)  
req.end()
```

## PONER Y ELIMINAR

Las solicitudes PUT y DELETE utilizan el mismo formato de solicitud POST y solo cambian el valor de options.method .

# Axios

**Axios es una biblioteca de JavaScript muy conveniente para realizar solicitudes HTTP en Node.js**

- [Introducción](#)
- [Instalación](#)
- [La API de Axios](#)
- [OBTENER solicitudes](#)
- [Aregar parámetros a las solicitudes GET](#)
- [Solicitudes POST](#)

## Introducción

Axios es una biblioteca de JavaScript muy popular que puede usar para realizar solicitudes HTTP, que funciona tanto en el navegador como en [Node.js](#) plataformas



Es compatible con todos los navegadores modernos, incluido el soporte para IE8 y superior.

Está basado en promesas y esto nos permite escribir código asíncrono/en espera para realizar [XHR](#) solicitudes con mucha facilidad.

El uso de Axios tiene bastantes ventajas sobre la [API Fetch nativa](#):

- admite navegadores más antiguos (Fetch necesita un polyfill)
- tiene una forma de abortar una
- solicitud tiene una forma de establecer un tiempo
- de espera de respuesta tiene protección CSRF
- incorporada admite el progreso de carga realiza
- la transformación automática de datos JSON funciona en
- Node.js

## Instalación

Axios se puede instalar usando [npm](#):

```
npm instalar axios
```

o [hilo](#):

```
hilo agregar axios
```

o simplemente incluyelo en tu página usando [unpkg.com](#):

```
<secuencia de comandos src="https://unpkg.com/axios/dist/axios.min.js"></secuencia de comandos>
```

## La API de Axios

Puede iniciar una solicitud HTTP desde el objeto axios :

```
axios({url:  
  'https://dog.ceo/api/breeds/list/all', método: 'obtener',  
  datos: {  
    foo: 'barra'  
  } })
```

pero por conveniencia, generalmente usará

- `axios.get()`
- `axios.post()`

(como en jQuery, usaría `$.get()` y `$.post()` en lugar de `$.ajax()` )

Axios ofrece métodos para todos los verbos HTTP, que son menos populares pero aún se usan:

- `axios.delete()`
- `axios.put()`
- `axios.parche()`
- `axios.opciones()`

y un método para obtener los encabezados HTTP de una solicitud, descartando el cuerpo:

- `axios.cabeza()`

## OBTENER solicitudes

Una forma conveniente de usar Axios es usar la sintaxis moderna (ES2017) `async/await`.

Este ejemplo de Node.js consulta la [API Dog](#) para recuperar una lista de todas las razas de perros, usando `await` y los cuenta: `axios.get()`,

```
constante axios = require('axios')

const getBreeds = asíncrono () => {
  intenta
    { volver esperar axios.get('https://dog.ceo/api/breeds/list/all') } catch (error)
    { console.error(error)

  }
}

const countBreeds = asíncrono () => {
  const razas = esperar getBreeds()

  if (razas.datos.mensaje) { console.log(
    `Obtuve ${Objeto.entradas(razas.datos.mensaje).longitud} razas`)
  }
}

contarRazas()
```

Si no desea usar `async/await`, puede usar [Promises](#) sintaxis:

```
constante axios = require('axios')

const obtener Razas = () => {
  prueba
    { return axios.get('https://dog.ceo/api/breeds/list/all')
  } captura (error)
    { consola.error(error)
  }
}

const countBreeds = asíncrono () => { const
  razas = getBreeds()
```

```

    .entonces(respuesta => {
      if (response.data.message)
        { console.log(`Obtuve ${Object.entries(response.data.message).length} razas`)
        }

    }) .catch(error => {
  consola.log(error) })

}

contarRazas()

```

## Agregar parámetros a las solicitudes GET

Una respuesta GET puede contener parámetros en la URL, como este: `https://site.com/?foo=bar`.

Con Axios puede realizar esto simplemente usando esa URL:

```
axios.get('https://site.com/?foo=bar')
```

o puede usar una propiedad `params` en las opciones:

```

axios.get('https://sitio.com/', {
  parámetros: {
    foo: 'barra'

  }})

```

## Solicitudes POST

Realizar una solicitud POST es como realizar una solicitud GET, pero en lugar de `axios.get`, use `axios.post`:

```
axios.post('https://sitio.com/')
```

Un objeto que contiene los parámetros POST es el segundo argumento:

```

axios.post('https://sitio.com/', {
  foo: 'barra'

})

```



# Websockets

**Los WebSockets son una alternativa a la comunicación HTTP en Aplicaciones Web. Ofrecen un canal de comunicación bidireccional de larga duración entre el cliente y el servidor.**

Los WebSockets son una alternativa a la comunicación HTTP en Aplicaciones Web.

Ofrecen un canal de comunicación bidireccional de larga duración entre el cliente y el servidor.

Una vez establecido, el canal se mantiene abierto, ofreciendo una conexión muy rápida con baja latencia y sobrecarga.

## Soporte de navegador para WebSockets

Los WebSockets son compatibles con todos los navegadores modernos.

IE	Edge *	Firefox	Chrome	Safari	Opera	iOS Safari *
6		53	59	8	44	8.4
7	12	54	60	9	45	9.2
8	13	55	61	9.1	46	9.3
9	14	56	62	10	47	10.2
10	15	57	63	10.1	48	10.3
11	16	58	64	11	49	11.2
17	59	65	11.1	50	51	11.3
18	60	66	TP	52		
		67		68		

## En qué se diferencian WebSockets de HTTP

HTTP es un protocolo muy diferente y también una forma diferente de comunicarse.

HTTP es un protocolo de solicitud/respuesta: el servidor devuelve algunos datos cuando el cliente los solicita.

Con WebSockets:

- el **servidor** puede enviar un **mensaje al cliente** sin que el cliente solicite explícitamente algo el cliente y el servidor pueden **comunicarse entre sí simultáneamente** se necesita intercambiar muy poca sobrecarga de
- datos para enviar mensajes. Esto significa una **comunicación de baja latencia**.
- 

Los **WebSockets** son excelentes para comunicaciones **en tiempo real** y **de larga duración** .

HTTP es excelente para **el intercambio de datos ocasionales** y las interacciones iniciadas por el cliente.

HTTP es **mucho más simple** de implementar, mientras que WebSockets requiere un poco más de sobrecarga.

## WebSockets seguros

Utilice siempre el protocolo seguro y encriptado para WebSockets, `wss://` .

`ws://` se refiere a la versión no segura de WebSockets (el `http://` de WebSockets), y debe ser evitado por razones obvias.

## Crear una nueva conexión WebSockets

```
const url = 'wss://myserver.com/algo' const conexión =
nuevo WebSocket(url)
```

la conexión es un [WebSocket](#) objeto.

Cuando la conexión se establece con éxito, se dispara el evento abierto .

Escúchelo asignando una función de devolución de llamada a la propiedad `onopen` del objeto de conexión :

```
conexión.onopen = () => { //...
}
```

Si hay algún error, se activa la devolución de llamada de la función `onerror` :

```
conexión.onerror = error => { console.log(`  
    Error de WebSocket: ${error}`)  
}
```

## Envío de datos al servidor usando WebSockets

Una vez que la conexión está abierta, puede enviar datos al servidor.

Puede hacerlo convenientemente dentro de la función de devolución de llamada onopen :

```
conexión.onopen = () => {  
    conexión.send('hola')  
}
```

## Recibiendo datos del servidor usando WebSockets

Escuche con una función de devolución de llamada en onmessage , que se llama cuando el evento del mensaje es recibido:

```
conexión.onmessage = e => {  
    consola.log(e.datos)  
}
```

## Implementar un servidor WebSockets en Node.js

[ws](#) es una biblioteca popular de WebSockets para [Node.js](#).

Lo usaremos para construir un servidor WebSockets. También se puede usar para implementar un cliente, y usar WebSockets para comunicarse entre dos servicios de backend.

Instálalo fácilmente usando

```
hilo inicial  
hilo agregar ws
```

El código que necesitas escribir es muy pequeño:

```
const WebSocket = require('ws')  
  
const wss = nuevo WebSocket.Server({ puerto: 8080 })
```

```
wss.on('conexión', ws => {
  ws.on('mensaje', mensaje => {
    console.log(` Mensaje recibido => ${mensaje}`)) ws.send('ho!') })
```

Este código crea un nuevo servidor en el puerto 8080 (el puerto predeterminado para WebSockets) y agrega una función de devolución de llamada cuando se establece una conexión, enviando ho! al cliente y registrar los mensajes que recibe.

## Ver un ejemplo en vivo en Glitch

Aquí hay un ejemplo en vivo de un servidor WebSockets: <https://glitch.com/edit/#!/flavio-websockets-server-example>

Aquí hay un cliente de WebSockets que interactúa con el servidor: [https://glitch.com/edit/#!/flavio\\_websockets-client-example](https://glitch.com/edit/#!/flavio_websockets-client-example)

# HTTPS, conexiones seguras

**El protocolo HTTPS es una extensión de HTTP, la transferencia de hipertexto Protocolo, que proporcionan una comunicación segura**

HTTP en inseguro por diseño.

Cuando abre su navegador y le pide a un servidor web que le envíe una página web, sus datos realizan 2 viajes: 1 del navegador al servidor web y 1 del servidor web al navegador.

Luego, según el contenido de la página web, es posible que necesite más conexiones para obtener los archivos CSS, los archivos JavaScript, las imágenes, etc.

Durante cualquiera de esas conexiones, cualquier red que vayan a cruzar sus datos puede ser **inspeccionada y manipulada**.

Las consecuencias pueden ser graves: es posible que tenga toda la actividad de su red monitoreada y registrada, por parte de una tercera parte que ni siquiera sabe que existe, algunas redes [pueden injectar anuncios](#), y podría estar sujeto a un ataque de intermediario, una amenaza de seguridad en la que el atacante puede manipular sus datos e incluso hacerse pasar por su computadora a través de la red. Es muy fácil para alguien simplemente escuchar los paquetes HTTP que se transmiten a través de una red Wi-Fi pública y sin cifrar. la red.

HTTPS tiene como objetivo resolver el problema de raíz: toda la comunicación entre su navegador y el servidor web está encriptada.

La privacidad y la seguridad son una preocupación importante en la Internet actual. Hace unos años, podía salirse con la suya simplemente usando una conexión encriptada en páginas protegidas con inicio de sesión o durante un pago de comercio electrónico. También debido a los precios y las complicaciones de los certificados SSL, la mayoría de los sitios web solo usaban HTTP.

Hoy HTTPS es un requisito en cualquier sitio. Más del 50% de toda la Web lo usa ahora.

Google Chrome recientemente comenzó a marcar los sitios HTTP como inseguros, solo para darle una razón válida para tener HTTPS obligatorio (y forzado) en todos sus sitios web.

Cuando se usa HTTP, el puerto del servidor predeterminado es 80, y en HTTPS es 443. Por supuesto, no es necesario agregarlo explícitamente si el servidor usa el puerto predeterminado.

HTTPS también se denomina a veces *HTTP sobre SSL* o *HTTP sobre TLS*.

La diferencia entre los dos es simple: TLS es el sucesor de SSL.

Cuando se usa HTTPS, lo único que no está encriptado es el dominio del servidor web y el puerto del servidor.

Cualquier otra información, incluida la ruta del recurso, los encabezados, las cookies y los parámetros de consulta, están encriptados.

No entraré en los detalles del análisis de cómo funciona el protocolo TLS bajo el capó, pero podría pensar que está agregando una buena cantidad de **gastos generales**, y tendría razón.

Cualquier cálculo que se agregue al procesamiento de los recursos de la red provoca una sobrecarga tanto en el cliente, el servidor y el tamaño de los paquetes transmitidos.

Sin embargo, HTTPS permite el uso del protocolo más nuevo **HTTP/2**, que tiene una gran ventaja sobre HTTP/1.1: es mucho más rápido.

¿Por qué? Hay muchas razones, una es la compresión de encabezados, otra es la multiplexación de recursos. Una es la inserción del servidor: el servidor puede impulsar más recursos cuando se solicita un recurso. Entonces, si el navegador solicita una página, también recibirá todos los recursos necesarios (imágenes, CSS, JS).

Aparte de los detalles, HTTP/2 es una gran mejora con respecto a HTTP/1.1 y **requiere HTTPS**. Esto significa que HTTPS, a pesar de tener la sobrecarga de cifrado, es mucho más rápido que HTTP, si las cosas están configuradas correctamente con una configuración moderna.

# Descriptores de archivo

## Cómo interactuar con descriptoros de archivos usando Node

Antes de que pueda interactuar con un archivo que se encuentra en su sistema de archivos, debe obtener un descriptor de archivo.

Un descriptor de archivo es lo que se devuelve al abrir el archivo usando el método `open()` ofrecido por el módulo `fs` :

```
const fs = require('fs')

fs.open('/ Usuarios/flavio/test.txt', 'r', (err, fd) => {
  //fd es nuestro descriptor de archivo })
```

Observe la `r` que usamos como segundo parámetro para la llamada `fs.open()` .

Esa bandera significa que abrimos el archivo para lectura.

Otras banderas que usará comúnmente son

- `r+` abre el archivo para lectura y escritura `w+` abre el archivo para lectura y escritura, colocando el flujo al principio del expediente. El archivo se crea si no existe
- `a` abre el archivo para escritura, colocando la secuencia al final del archivo. El archivo es creado si no existe
- `a+` abre el archivo para lectura y escritura, colocando la secuencia al final del archivo. El archivo se crea si no existe

También puede abrir el archivo mediante el método `fs.openSync` , que en lugar de proporcionar el objeto descriptor de archivo en una devolución de llamada, lo devuelve:

```
const fs = require('fs')

intente
{ const fd = fs.openSync('/Users/flavio/test.txt', 'r') } catch (err)
{ console.error(err)

}
```

Una vez que obtenga el descriptor de archivo, de la forma que elija, puede realizar todas las operaciones que lo requieran, como llamar a `fs.open()` y muchas otras operaciones que interactúan con el sistema de archivos.



## Estadísticas de archivo

### Cómo obtener los detalles de un archivo usando Node

Cada archivo viene con un conjunto de detalles que podemos inspeccionar usando Node.

En particular, usando el método stat() provisto por el módulo fs .

Lo llamas pasando una ruta de archivo, y una vez que Node obtiene los detalles del archivo, llamará a la función de devolución de llamada que pasas, con 2 parámetros: un mensaje de error y las estadísticas del archivo:

```
const fs = require('fs') fs.stat('/
Users/flavio/test.txt', (err, stats) => {
  si (err)
    { consola.error(err)
      devolver
    }
  //tenemos acceso al archivo stats en `stats`
})
```

Node también proporciona un método de sincronización, que bloquea el hilo hasta que las estadísticas del archivo estén listas:

```
const fs = require('fs') try { const
stats = fs.stat('/Users/flavio/
test.txt')
} catch (err)
{ consola.error(err)
}
```

La información del archivo se incluye en la variable stats. ¿Qué tipo de información podemos extraer?

usando las estadísticas?

Mucho, incluyendo:

- si el archivo es un directorio o un archivo, usando stats.isFile() y stats.isDirectory() si el archivo es un enlace
- simbólico usando stats.isSymbolicLink() el tamaño del archivo en bytes usando stats.size .
- 

Hay otros métodos avanzados, pero la mayor parte de lo que usará en su programación diaria es esto.

```
const fs = require('fs') fs.stat('/
Users/flavio/test.txt', (err, stats) => {
  si (err)
    { consola.error(err)
      devolver
    }
})
```

```
stats.isFile() //verdadero  
stats.isDirectory() //falso  
stats.isSymbolicLink() //falso  
estadísticas.tamaño //1024000 //= 1MB  
})
```

## Rutas de archivo

### Cómo interactuar con rutas de archivo y manipularlas en Node

- [Obtener información de un camino](#)
- [Trabajando con caminos](#)

Cada archivo en el sistema tiene una ruta.

En Linux y macOS, una ruta podría verse así:

```
/usuarios/flavio/archivo.txt
```

mientras que las computadoras con Windows son diferentes y tienen una estructura como:

```
C:\usuarios\flavio\archivo.txt
```

Debe prestar atención al usar rutas en sus aplicaciones, ya que esta diferencia debe ser tenido en cuenta.

Usted incluye este módulo en sus archivos usando

```
const ruta = require('ruta')
```

y puedes empezar a usar sus métodos.

## Obtener información de un camino

Dada una ruta, puede extraer información de ella usando esos métodos:

- `dirname` : obtiene la carpeta principal de un archivo
- `basename` : obtiene la parte del nombre del archivo
- `extname` : obtiene la extensión del archivo

Ejemplo:

```
const notas = '/usuarios/flavio/notas.txt'

ruta.dirname(notas) // /usuarios/flavio
ruta.basename(notas) // notas.txt
ruta.extname(notas) // .txt
```

Puede obtener el nombre del archivo sin la extensión especificando un segundo argumento para

```
nombre base ::
```

```
ruta.nombrebase(notas, ruta.extname(notas)) //notas
```

## Trabajando con caminos

Puedes unir dos o más partes de una ruta usando path.join() :

```
const nombre = 'flavio'  
ruta.join('/', 'usuarios', nombre, 'notas.txt') //'/usuarios/flavio/notas.txt'
```

Puede obtener el cálculo de la ruta absoluta de una ruta relativa usando path.resolve() :

```
path.resolve('flavio.txt') //'/Users/flavio/flavio.txt' si se ejecuta desde mi carpeta de inicio
```

En este caso, Node simplemente agregará /flavio.txt al directorio de trabajo actual. Si especifica una segunda carpeta de parámetros, resolve usará la primera como base para la segunda:

```
path.resolve('tmp', 'flavio.txt')//'/Users/flavio/tmp/flavio.txt' si se ejecuta desde mi carpeta de inicio  
es
```

Si el primer parámetro comienza con una barra inclinada, eso significa que es una ruta absoluta:

```
ruta.resolve('/etc', 'flavio.txt')//'/etc/flavio.txt'
```

path.normalize() es otra función útil, que intentará calcular la ruta real, cuando contiene especificadores relativos como . o .. , o doble barra:

```
ruta.normalize('/usuarios/flavio/../prueba.txt') //'/usuarios/prueba.txt'
```

**Tanto resolver como normalizar no comprobarán si existe la ruta.** Simplemente calculan una ruta en función de la información que obtuvieron.

# Lectura de archivos

## Cómo leer archivos usando Node

La forma más sencilla de leer un archivo en Node es usar el método `fs.readFile()`, pasándole la ruta del archivo y una función de devolución de llamada que se llamará con los datos del archivo (y el error):

```
const fs = require('fs')

fs.readFile('/ Usuarios/flavio/prueba.txt', (err, datos) => {
  si (err)
    { consola.error(err)
      devolver

    } consola.log(datos) })
```

Alternativamente, puede usar la versión síncrona `fs.readFileSync()`:

```
const fs = require('fs')

intente
{ const data = fs.readFileSync('/Users/flavio/test.txt', 'utf8') console.log(data) } catch
(err) { console.error(err)

}
```

La codificación predeterminada es `utf8`, pero puede especificar una codificación personalizada utilizando un segundo parámetro.

Tanto `fs.readFile()` como `fs.readFileSync()` leen el contenido completo del archivo en la memoria antes de devolver los datos.

Esto significa que los archivos grandes tendrán un gran impacto en el consumo de memoria y la velocidad de ejecución del programa.

En este caso, una mejor opción es leer el contenido del archivo mediante secuencias.

# Escribir archivos

## Cómo escribir archivos usando Node

La forma más fácil de escribir en archivos en Node.js es usar la API `fs.writeFile()`.

Ejemplo:

```
const fs = require('fs')

const content = 'Algo de contenido'

fs.writeFile('/Users/flavio/test.txt', contenido, (err) => {
  si (err)
    { consola.error(err)
      devolver

  } //archivo escrito con éxito })
```

Alternativamente, puede usar la versión síncrona `fs.writeFileSync()`:

```
const fs = require('fs')

const content = 'Algo de contenido'

try
{ const data = fs.writeFileSync('/Users/flavio/test.txt', content) //archivo escrito con éxito } catch (err)
{ console.error(err)

}
```

De forma predeterminada, esta API **reemplazará el contenido del archivo** si ya existe.

Puede modificar el valor predeterminado especificando un indicador:

```
fs.writeFile('/Users/flavio/test.txt', contenido, { flag: 'a+' }, (err) => {})
```

Las banderas que probablemente usará son

- `r+` abre el archivo para lectura y escritura `w+`
- abre el archivo para lectura y escritura, colocando el flujo al principio del expediente. El archivo se crea si no existe
- `a` abra el archivo para escritura, colocando la secuencia al final del archivo. El archivo es creado si no existe

- `a+` abre el archivo para lectura y escritura, colocando la secuencia al final del archivo. los el archivo se crea si no existe

(puede encontrar más indicadores en [https://nodejs.org/api/fs.html#fs\\_file\\_system\\_flags](https://nodejs.org/api/fs.html#fs_file_system_flags))

## Agregar a un archivo

Un método útil para agregar contenido al final de un archivo es `fs.appendFile()` (y su

contraparte de `fs.appendFileSync()`):

```
const content = 'Algo de contenido!'

fs.appendFile('archivo.log', contenido, (err) => {
  si (err)
    { consola.error(err)
      devolver
    }
    //hecho!
  })
})
```

## Usando flujos

Todos esos métodos escriben el contenido completo en el archivo antes de devolver el control a su programa (en la versión asíncrona, esto significa ejecutar la devolución de llamada)

En este caso, una mejor opción es escribir el contenido del archivo usando flujos.

# Trabajando con carpetas

## Cómo interactuar con carpetas usando Node

El módulo principal de Node.js fs proporciona muchos métodos útiles que puede usar para trabajar con carpetas

## Comprobar si existe una carpeta

Use `fs.access()` para verificar si la carpeta existe y Node puede acceder a ella con sus permisos.

## Crear una nueva carpeta

Utilice `fs.mkdir()` o `fs.mkdirSync()` para crear una nueva carpeta.

```
const fs = require('fs')

const folderName = '/Usuarios/flavio/prueba'

prueba
{ si (!fs.existsSync(dir)){ fs.mkdirSync(dir)

}

} catch (err)
{ consola.error(err)
}
```

## Leer el contenido de un directorio

Use `fs.readdir()` o `fs.readdirSync` para leer el contenido de un directorio.

Este fragmento de código lee el contenido de una carpeta, tanto archivos como subcarpetas, y devuelve su ruta relativa:

```
const fs = require('fs') const ruta =
require('ruta')

const folderPath = '/Usuarios/flavio'

fs.readdirSync(carpetaPath)
```

Puede obtener la ruta completa:

```
fs.readdirSync(carpetaPath).map(fileName => {
  return
    path.join(carpetaPath, fileName)
})
```

También puede filtrar los resultados para devolver solo los archivos y excluir las carpetas:

```
const isFile = fileName => { return
  fs.lstatSync(fileName).isFile()
}

fs.readdirSync(carpetaPath).map(fileName => {
  return path.join(folderPath, fileName)).filter(isFile)
})
```

## Cambiar el nombre de una carpeta

Utilice `fs.rename()` o `fs.renameSync()` para cambiar el nombre de la carpeta. El primer parámetro es la ruta actual, el segundo la nueva ruta:

```
const fs = require('fs')

fs.rename('/ Usuarios/flavio', '/ Usuarios/roger', (err) => {
  si (err)
    { consola.error(err)
      devolver
    }
    //hecho
  })
})
```

`fs.renameSync()` es la versión síncrona:

```
const fs = require('fs')

prueba
  { fs.renameSync('/ Usuarios/flavio', '/ Usuarios/roger')
} catch (err)
  { consola.error(err)
  }
```

## Quitar una carpeta

Utilice `fs.rmdir()` o `fs.rmdirSync()` para eliminar una carpeta.

Eliminar una carpeta que tiene contenido puede ser más complicado de lo que necesita.

En este caso, recomiendo instalar el módulo `fs-extra`, que es muy popular y está bien mantenido, y es un reemplazo directo del módulo `fs`, que brinda más funciones en la parte superior.

de eso

En este caso, el método `remove()` es lo que desea.

Instálalo usando

```
npm instalar fs-extra
```

y usarlo así:

```
const fs = require('fs-extra')

carpeta const = '/Usuarios/flavio'

fs.remove(carpeta, err => { console.error(err) })
```

También se puede usar con promesas:

```
fs.remove(carpeta).then(() => {
  //hecho
}).catch(error => {
  consola.error(err) })
```

o con `async/await`:

```
función asincrona removeFolder(carpeta) {
  intente
    { esperar fs.remove(carpeta) //
      hecho
    } catch (err)
      { consola.error(err)
      }
  }

  carpeta const = '/Usuarios/flavio'
  removeFolder(carpeta)
```

# El módulo fs

**El módulo fs de Node.js proporciona funciones útiles para interactuar con el sistema de archivos**

El módulo fs proporciona una gran cantidad de funciones muy útiles para acceder e interactuar con el archivo. sistema.

No hay necesidad de instalarlo. Al ser parte del núcleo de Node, se puede usar simplemente requiriendo:

```
const fs = require('fs')
```

Una vez que lo haga, tendrá acceso a todos sus métodos, que incluyen:

- `fs.access()` : comprueba si el archivo existe y Node puede acceder a él con sus permisos
- `fs.appendFile()` : agrega datos a un archivo. Si el archivo no existe, se crea
- `fs.chmod()` : cambie los permisos de un archivo especificado por el nombre de archivo pasado. Relacionado:  
`fs.lchmod()` , `fs.fchmod()`
- `fs.chown()` : cambia el propietario y el grupo de un archivo especificado por el nombre de archivo pasado.  
Relacionado: `fs.fchown()` , `fs.lchown()`
- `fs.close()` : cierra un descriptor de
- `fs.copyFile()` : copia un archivo
- `fs.createReadStream()` : crea un flujo de archivo legible
- `fs.createWriteStream()` : crea un flujo de archivos grabable
- `fs.link()` : crea un nuevo enlace duro a un archivo
- `fs.mkdir()` : crea una nueva carpeta
- `fs.mkdtemp()` : crea un directorio temporal
- `fs.open()` : establece el modo de archivo
- `fs.readdir()` : lee el contenido de un directorio
- `fs.readFile()` : lee el contenido de un archivo. Relacionado: `fs.read()`
- `fs.readlink()` : leer el valor de un enlace simbólico
- `fs.realpath()` : resolver punteros de ruta de
- `fs.rename()` : cambiar el nombre de un archivo o carpeta ruta completa
- `fs.rmdir()` : eliminar una carpeta
- `fs.stat()` : devuelve el estado del archivo identificado por el nombre de archivo pasado.
- `fs.fstat()` , `fs.lstat()`
- `fs.symlink()` : crea un nuevo enlace simbólico a un archivo
- `fs.truncate()` : trunca a la longitud especificada el archivo identificado por el nombre de archivo pasado.  
Relacionado: `fs.ftruncate()`
- `fs.unlink()` : elimina un archivo o un enlace simbólico

- `fs.unwatchFile()` : deja de buscar cambios en un archivo
- `fs.utimes()` : cambia la marca de tiempo del archivo identificado por el nombre de archivo pasado. Relacionado:
- `fs.futimes()`
- `fs.watchFile()` : comience a buscar cambios en un archivo. Relacionado: `fs.watch()` `fs.writeFile()` :
- `fs.writeFileSync()` : escribe datos en un archivo. Relacionado: `fs.write()`

Una cosa peculiar sobre el módulo `fs` es que todos los métodos son asíncronos de forma predeterminada, pero también pueden funcionar de forma síncrona agregando `Sync`.

Por ejemplo:

- `fs.rename()`
- `fs.renameSync()`
- `fs.writeFile()`
- `fs.writeFileSync()`

Esto hace una gran diferencia en el flujo de su aplicación.

Nodo 10 incluye [soporte experimental](#) por una [promesa](#) API basada

Por ejemplo, examinemos el método `fs.rename()`. La API asíncrona se utiliza con un llamar de vuelta:

```
const fs = require('fs')

fs.rename('antes.json', 'después.json', (err) => { if (err) { return
  console.error(err)
}

//hecho
})
```

Se puede usar una API síncrona de esta manera, con un bloque `try/catch` para manejar los errores:

```
const fs = require('fs')

prueba
{
  fs.renameSync('antes.json', 'después.json')
  //hecho
} catch (err)
{
  consola.error(err)
}
```

La diferencia clave aquí es que la ejecución de su secuencia de comandos se bloqueará en el segundo ejemplo, hasta que la operación del archivo se realice correctamente.



# El módulo de ruta

**El módulo de ruta de Node.js proporciona funciones útiles para interactuar con las rutas de archivo**

El módulo de ruta proporciona una gran cantidad de funciones muy útiles para acceder e interactuar con el archivo sistema.

No hay necesidad de instalarlo. Al ser parte del núcleo de Node, se puede usar simplemente requiriendo:

```
const ruta = require('ruta')
```

Este módulo proporciona path.sep que proporciona el separador de segmento de ruta ( \ en Windows y / en Linux / macOS) y path.delimiter que proporciona el delimitador de ruta ( ; en Windows y : en Linux / macOS).

Estos son los métodos de ruta :

- `ruta.nombrebase()`
- `ruta.dirname()`
- `ruta.extname()`
- `ruta.isAbsolute()`
- `ruta.join()`
- `ruta.normalizar()`
- `ruta.analizar()`
- `ruta.relativa()`
- `ruta.resolve()`

## ruta.nombrebase()

Devuelve la última parte de una ruta. Un segundo parámetro puede filtrar la extensión del archivo:

```
require('ruta').nombrebase('/prueba/algo') //algo require('ruta').nombrebase('/prueba/algo.txt') //algo.txt require('ruta').nombrebase ('/prueba/algo.txt', '.txt') //algo
```

## ruta.dirname()

Devuelve la parte del directorio de una ruta:

```
require('ruta').dirname('/prueba/algo') // /prueba require('ruta').dirname('/prueba/algo/archivo.txt') // /prueba/algo
```

## ruta.extname()

Devuelve la parte de extensión de una ruta

```
require('ruta').dirname('/prueba/algo') // " require('ruta').dirname('/prueba/algo/archivo.txt') // '.txt'
```

## ruta.isAbsolute()

Devuelve verdadero si es una ruta absoluta

```
require('ruta').isAbsolute('/prueba/algo') // verdadero  
require('ruta').isAbsolute('./prueba/algo') // falso
```

## ruta.join()

Une dos o más partes de un camino:

```
const nombre = 'flavio'  
require('ruta').join('/', 'usuarios', nombre, 'notas.txt') // usuarios/flavio/notas.txt'
```

## ruta.normalizar()

Intenta calcular la ruta real cuando contiene especificadores relativos como . o .. , o doble barras:

```
require('ruta').normalize('/usuarios/flavio/../prueba.txt') // usuarios/prueba.txt
```

## ruta.analizar()

Analiza una ruta a un objeto con los segmentos que lo componen:

- raíz : la raíz
- dir : la ruta de la carpeta que comienza desde la raíz
- base : el nombre del archivo + extensión
- nombre : el nombre del archivo
- ext : la extensión del archivo

Ejemplo:

```
require('ruta').parse('/usuarios/prueba.txt')
```

da como resultado

```
{  
    raíz: '/',
    directorio: '/  
usuarios', base:  
'prueba.txt', ext: '.txt',  
nombre: 'prueba'  
}
```

## ruta.relativa()

Acepta 2 caminos como argumentos. Devuelve la ruta relativa de la primera ruta a la segunda, según el directorio de trabajo actual.

Ejemplo:

```
require('ruta').relative('/Usuarios/flavio', '/Usuarios/flavio/test.txt') //prueba.txt require('ruta').relative('/Usuarios/  
flavio', '/Usuarios/flavio/algo/prueba.txt') //algo /prueba.txt'
```

## ruta.resolve()

Puede obtener el cálculo de la ruta absoluta de una ruta relativa usando path.resolve() :

```
path.resolve('flavio.txt') //Users/flavio/flavio.txt si se ejecuta desde mi carpeta de inicio
```

Al especificar un segundo parámetro, resolve usará el primero como base para el segundo:

```
path.resolve('tmp', 'flavio.txt')//Users/flavio/tmp/flavio.txt si se ejecuta desde mi carpeta de inicio  
es
```

Si el primer parámetro comienza con una barra inclinada, eso significa que es una ruta absoluta:

```
ruta.resolve('/etc', 'flavio.txt')//etc/flavio.txt'
```

## El módulo del sistema operativo

### El módulo os de Node.js proporciona funciones útiles para interactuar con el sistema subyacente

Este módulo proporciona muchas funciones que puede usar para recuperar información del sistema operativo subyacente y la computadora en la que se ejecuta el programa, e interactuar con él.

```
const os = require('os')
```

Hay algunas propiedades útiles que nos dicen algunas cosas clave relacionadas con el manejo de archivos:

os.EOL proporciona la secuencia del delimitador de línea. Está \n en Linux y macOS, y \r\n en Ventanas.

Cuando digo Linux y macOS me refiero a plataformas POSIX. Para simplificar, excluyo otros sistemas operativos menos populares en los que se puede ejecutar Node.

os.constants.signals nos dice todas las constantes relacionadas con el manejo de señales de proceso, como SIGHUP, SIGKILL, etc.

os.constants.errno establece las constantes para el informe de errores, como EADDRINUSE, EOVERRLOW y más.

Puede leerlos todos en [https://nodejs.org/api/os.html#os\\_signal\\_constants](https://nodejs.org/api/os.html#os_signal_constants).

Veamos ahora los principales métodos que proporciona os :

- [os.arch\(\)](#)
- [os.cpus\(\)](#)
- [os.endianess\(\)](#)
- [os.freemem\(\)](#)
- [os.homedir\(\)](#)
- [os.nombre de host\(\)](#)
- [os.cargarvg\(\)](#)
- [os.interfaces de red\(\)](#)
- [os.plataforma\(\)](#)
- [os.release\(\)](#)
- [os.tmpdir\(\)](#)
- [os.totalmem\(\)](#)
- [os.tipo\(\)](#)
- [os.uptime\(\)](#)
- [os.userInfo\(\)](#)

## os.arch()

Devuelve la cadena que identifica la arquitectura subyacente, como arm , x64 , brazo64 .

## os.cpus()

Devolver información sobre las CPU disponibles en su sistema.

Ejemplo:

```
[ { modelo: 'Intel(R) Core(TM)2 Duo CPU velocidad: P8600 a 2,40 GHz',
  2400, veces:
    { usuario: 281685380,
      agradable: 0,
      sistema: 187986530,
      inactivo: 685833750,
      irq: 0 },
    { modelo: 'CPU Intel(R) Core(TM)2 Duo velocidad: P8600 a 2,40 GHz',
      2400,
      veces:
        { usuario: 282348700,
          agradable: 0,
          sistema: 161800480,
          inactivo: 703509470,
          irq: 0 } } ]
```

## os.endianness()

Devuelve BE o LE dependiendo de si Node se compiló con Big Endian o Little Endian.

## os.freemem()

Devuelve el número de bytes que representan la memoria libre en el sistema.

## os.homedir()

Devuelve la ruta al directorio de inicio del usuario actual.

Ejemplo:

```
'/Usuarios/flavio'
```

## os.nombre de host()

Devuelve el nombre de host.

## os.cargarvg()

Retorna el cálculo realizado por el sistema operativo sobre el promedio de carga.

Solo devuelve un valor significativo en Linux y macOS.

Ejemplo:

```
[ 3.68798828125, 4.00244140625, 11.1181640625 ]
```

## os.interfaces de red()

Devuelve los detalles de las interfaces de red disponibles en su sistema.

Ejemplo:

```
{ lo0:  
  [ { dirección: '127.0.0.1', máscara  
    de red: '255.0.0.0', familia:  
    'IPv4', mac: 'fe:82:00:00:00:00',  
    interno: verdadero },  
  
  { dirección: '::1',  
    máscara de red: 'ffff:ffff:ffff:ffff:ffff:ffff', familia: 'IPv6', mac:  
    'fe:82:00:00:00:00', ID de alcance: 0, interno: verdadero }, { dirección:  
    'fe80::1', máscara de red: 'ffff:ffff:ffff:ffff::', familia: 'IPv6', mac:  
    'fe:82:00:00:00:00', scopeid: 1, interno: verdadero } ],
```

```
en1:  
  [ { dirección: 'fe82::9b:8282:d7e6:496e',  
    máscara de red: 'ffff:ffff:ffff:ffff::', familia: 'IPv6',  
    mac: '06:00:00:02:0e:00', scopeid: 5, interno:  
    falso }, { dirección: '192.168 .1.38',
```

```
    máscara de red: '255.255.255.0',  
    familia: 'IPv4', mac:  
    '06:00:00:02:0e:00',
```

```
    interno: falso },  
utun0:  
[ { dirección: 'fe80::2513:72bc:f405:61d0', máscara de  
red: 'ffff:ffff:ffff::', familia: 'IPv6', mac:  
'fe:80:00:20:00: 00', ID de alcance: 8, interno:  
falso } ] }
```

## os.plataforma()

Devuelve la plataforma para la que se compiló Node:

- darwin
- freebsd
- linux
- openbsd
- ganar32
- ...más

## os.release()

Devuelve una cadena que identifica el número de versión del sistema operativo

## os.tmpdir()

Devuelve la ruta a la carpeta temporal asignada.

## os.totalmem()

Devuelve el número de bytes que representan la memoria total disponible en el sistema.

## os.tipo()

Identifica el sistema operativo:

- linux
- Darwin en mac OS
- Windows\_NT en Windows

## os.uptime()

Devuelve la cantidad de segundos que la computadora ha estado funcionando desde que se reinició por última vez.

## **os.userInfo()**

# El módulo de eventos

## El módulo de eventos de Node.js proporciona la clase EventEmitter

El módulo de eventos nos proporciona la clase EventEmitter, que es clave para trabajar con eventos en Nodo.

Publiqué un artículo completo sobre [eso](#), así que aquí solo describiré la API sin más ejemplos sobre cómo usarlo.

```
const EventEmitter = require('events') const puerta
= new EventEmitter()
```

El detector de eventos come su propia comida para perros y usa estos eventos:

- `newListener` cuando se agrega un oyente
- `removeListener` cuando se elimina un oyente

Aquí hay una descripción detallada de los métodos más útiles:

- `emisor.addListener()`
- `emisor.emit()`
- `emitter.eventNames()`
- `emitter.getMaxListeners()`
- `emitter.listenerCount()`
- `emisor.oyentes()`
- `emisor.off()`
- `emisor.on()`
- `emisor.una vez()`
- `emisor.prependListener ()`
- `emitter.prependOnceListener()`
- `emitter.removeAllListeners()`
- `emisor.removeListener()`
- `emisor.setMaxListeners()`

### emisor.addListener()

Alias para `emitter.on()` .

### emisor.emit()

Emite un evento. Llama sincrónicamente a cada detector de eventos en el orden en que se registraron.

## emitter.eventNames()

Devuelve una matriz de cadenas que representan los eventos registrados en el EventListener actual:

```
puerta.eventNames()
```

## emitter.getMaxListeners()

Obtenga la cantidad máxima de oyentes que se pueden agregar a un objeto EventListener, que tiene un valor predeterminado de 10, pero se puede aumentar o disminuir usando `setMaxListeners()`

```
puerta.getMaxListeners()
```

## emitter.listenerCount()

Obtenga el recuento de oyentes del evento pasado como parámetro:

```
puerta.listenerCount('abrir')
```

## emisor.oyentes()

Obtiene una matriz de oyentes del evento pasado como parámetro:

```
puerta.oyentes('abierto')
```

## emisor.off()

Alias para `emitter.removeListener()` agregado en el nodo 10

## emisor.on()

Agrega una función de devolución de llamada que se llama cuando se emite un evento.

Uso:

```
puerta.on('abrir', () => {
```

```
console.log(' Se abrió la puerta') })
```

## emisor.una vez()

Agrega una función de devolución de llamada que se llama cuando se emite un evento por primera vez después registrando esto. Esta devolución de llamada solo se llamará una vez, nunca más.

```
const EventEmitter = require('events') const ee =  
new EventEmitter()  
  
ee.once('mi-evento', () => {  
  // Llamar a la función de devolución de llamada una vez  
})
```

## emisor.prependListener ()

Cuando agrega un oyente usando on o addListener y llama al , se agrega el último en la cola de oyentes, último. Usando prependListener , se agrega y se llama antes que otros oyentes.

## emitter.prependOnceListener()

Cuando agrega un oyente usando una vez , se agrega el último en la cola de oyentes y se llama el último. usando prependOnceListener , se agrega y se llama antes que otros oyentes.

## emitter.removeAllListeners()

Elimina todos los oyentes de un objeto emisor de eventos que escuchan un evento específico:

```
puerta.removeAllListeners('abrir')
```

## emisor.removeListener()

Eliminar un oyente específico. Puede hacer esto guardando la función de devolución de llamada en una variable, cuando se agrega, para que pueda consultarla más tarde:

```
const hacerAlgo = () => {}  
puerta.on('abrir', hacerAlgo)  
puerta.removeListener('abrir', hacerAlgo)
```

## emisor.setMaxListeners()

Establece la cantidad máxima de oyentes que se pueden agregar a un objeto EventListener, que por defecto a 10 pero se puede aumentar o disminuir.

```
puerta.setMaxListeners(50)
```

# El módulo http

**El módulo http de Node.js proporciona funciones y clases útiles para construir un servidor HTTP**

El módulo principal de HTTP es un módulo clave para las redes de nodos.

- [Propiedades](#)

- [http.MÉTODOS](#)
- [http.CÓDIGOS\\_DE\\_ESTADO](#)
- [http.agenteglobal](#)

- [Métodos](#)

- [http.createServer\(\)](#)
- [http.solicitud\(\)](#)
- [http.get\(\)](#)

- [Clases](#)

- [http.Agente](#)
- [http.ClientRequest](#)
- [http.Servidor](#)
- [http.ServerResponse](#)
- [http.MensajeEntrante](#)

Se puede incluir usando

```
constante http = require('http')
```

El módulo proporciona algunas propiedades y métodos, y algunas clases.

## Propiedades

### http.MÉTODOS

Esta propiedad enumera todos los métodos HTTP admitidos:

```
> require('http').METHODS [ 'ACL',
  'BIND', 'CHECKOUT', 'CONNECT',
  'COPY', 'DELETE', 'GET',
```

```
'CABEZA',
'ENLACE',
'CERRAR',
'BÚSQUEDA M',
'UNIR',
'MKACTIVIDAD',
'MKCALENDARIO',
'MK COL',
'MUEVETE',
'NOTIFICAR',
'OPCIONES',
'PARCHE',
'CORREO',
'PROFINIR',
'PROPPATCH',
'PURGA',
'PONER',
'REENCUADERNAR',
'REPORTE',
'BÚSQUEDA',
'SUSCRIBIR',
'RASTRO',
'DESATAR',
'DESCONECTAR',
'DESBLOQUEAR',
'CANCELAR SUSCRIPCIÓN' ]
```

## **http.CÓDIGOS\_DE\_ESTADO**

Esta propiedad enumera todos los códigos de estado HTTP y su descripción:

```
> require('http').STATUS_CODES { '100': 'Continuar',
'101': 'Protocolos de conmutación', '102': 'Procesando',
'200': 'OK', '201': 'Creado', '202': 'Aceptado', '203':
'Información no autorizada', '204': 'Sin contenido',
'205': 'Restablecer contenido', '206': 'Contenido
parcial', '207': 'Estado múltiple', '208': 'Ya
informado', '226': 'IM usado', '300': 'Opciones
múltiples', '301': 'Movido permanentemente', '302': 'Encontrado',
'303': 'Ver otro', '304': 'No modificado', '305': 'Usar proxy', '307':
'Redirecciónamiento temporal', '308': 'Redirecciónamiento
permanente', '400': 'Solicitud incorrecta',
```

```
'401': 'No autorizado',
'402': 'Pago requerido',
'403': 'Prohibido',
'404': 'No encontrado',
'405': 'Método no permitido',
'406': 'No aceptable',
'407': 'Se requiere autenticación de proxy',
'408': 'Tiempo de espera de solicitud',
'409': 'Conflicto',
'410': 'Ido',
'411': 'Longitud requerida',
'412': 'Falló la condición previa',
'413': 'Carga útil demasiado grande',
'414': 'URI demasiado largo',
'415': 'Tipo de medio no admitido',
'416': 'Rango no satisfactorio',
'417': 'Expectativa fallida',
'418': 'Soy una tetera',
'421': 'Solicitud mal dirigida',
'422': 'Entidad no procesable',
'423': 'Bloqueado',
'424': 'Dependencia fallida',
'425': 'Colección desordenada',
'426': 'Actualización requerida',
'428': 'Condición previa requerida',
'429': 'Demasiadas solicitudes',
'431': 'Campos de encabezado de solicitud demasiado grandes',
'451': 'No disponible por motivos legales',
'500': 'Error interno del servidor',
'501': 'No implementado',
'502': 'Puerta de enlace incorrecta',
'503': 'Servicio no disponible',
'504': 'Tiempo de espera de la puerta de enlace',
'505': 'Versión HTTP no compatible',
'506': 'Variante también negocia',
'507': 'Almacenamiento insuficiente',
'508': 'Bucle detectado',
'509': 'Límite de ancho de banda excedido',
'510': 'No extendido',
'511': 'Se requiere autenticación de red' }
```

## http.agenteglobal

Apunta a la instancia global del objeto Agente, que es una instancia de la clase http.Agent .

Se utiliza para administrar la persistencia y reutilización de conexiones para clientes HTTP, y es un componente clave de la red Node HTTP.

Más en la descripción de la clase http.Agent más adelante.

## Métodos

## http.createServer()

Devuelve una nueva instancia de la clase http.Server .

Uso:

```
const servidor = http.createServer((req, res) => {
  //manejar cada solicitud con esta devolución de llamada })
```

## http.solicitud()

Realiza una solicitud HTTP a un servidor, creando una instancia de la clase http.ClientRequest .

## http.get()

Similar a http.request() , pero establece automáticamente el método HTTP en GET y llama a req.end() automáticamente.

# Clases

El módulo HTTP proporciona 5 clases:

- http.Agent
- http.ClientRequest
- http.Servidor
- http.ServerResponse
- http.MensajeEntrante

## http.Agent

Node crea una instancia global de la clase http.Agent para administrar la persistencia y reutilización de conexiones para clientes HTTP, un componente clave de la red HTTP de Node.

Este objeto se asegura de que todas las solicitudes realizadas a un servidor se pongan en cola y que se utilice un solo socket. reutilizado

También mantiene un grupo de enchufes. Esto es clave por razones de rendimiento.

## http.ClientRequest

Se crea un objeto http.ClientRequest cuando se llama a http.request() o http.get() .

Cuando se recibe una respuesta, se llama al evento de respuesta con la respuesta, con un  
Instancia de http.IncomingMessage como argumento.

Los datos devueltos de una respuesta se pueden leer de 2 maneras:

- puede llamar al método response.read() en el
- controlador de eventos de respuesta , puede configurar un detector de eventos para el evento de datos , por lo que  
puede escuchar los datos transmitidos.

## http.Servidor

Esta clase es comúnmente instanciada y devuelta cuando se crea un nuevo servidor usando  
http.createServer() .

Una vez que tiene un objeto de servidor, tiene acceso a sus métodos:

- close() evita que el servidor acepte nuevas conexiones listen() inicia el
- servidor HTTP y escucha las conexiones

## http.ServerResponse

Creado por un http.Server y pasado como segundo parámetro al evento de solicitud  
incendios

Comúnmente conocido y utilizado en código como res :

```
servidor const = http.createServer((req, res) => { //res es un  
    objeto http.ServerResponse })
```

El método al que siempre llamará en el controlador es end() , que cierra la respuesta, el mensaje está  
completo y el servidor puede enviarlo al cliente. Debe llamarse a cada  
respuesta.

Estos métodos se utilizan para interactuar con encabezados HTTP:

- getHeaderNames() obtiene la lista de los nombres de los encabezados HTTP ya establecidos
- getHeaders() obtiene una copia de los encabezados HTTP ya establecidos
- setHeader('headername', value) establece un valor de encabezado HTTP
- getHeader('headername') obtiene un encabezado HTTP ya establecido
- removeHeader('headername') elimina un encabezado HTTP ya establecido
- hasHeader('headername') devuelve verdadero si la respuesta tiene ese encabezado
- establecido headersSent() devuelve verdadero si los encabezados ya han sido enviado al cliente

Después de procesar los encabezados, puede enviarlos al cliente llamando a `response.writeHead()` , que acepta el código de estado como primer parámetro, el mensaje de estado opcional y el objeto de encabezados.

Para enviar datos al cliente en el cuerpo de la respuesta, usa `write()` . Enviará datos almacenados en búfer al flujo de respuesta HTTP.

Si los encabezados aún no se enviaron usando `response.writeHead()` , enviará los encabezados primero, con el código de estado y el mensaje que se establece en la solicitud, que puede editar configurando el valores de las propiedades `statusCode` y `statusMessage` :

```
response.statusCode = 500  
response.statusMessage = 'Error interno del servidor'
```

## http.MensajeEntrante

Un objeto `http.IncomingMessage` es creado por:

- `http.Server` al escuchar el evento de solicitud
- `http.ClientRequest` al escuchar el evento de respuesta

Se puede utilizar para acceder a la respuesta:

- estado usando sus métodos `statusCode` y `statusMessage` encabezados
- usando su método de encabezados o `rawHeaders`
- Método HTTP usando su método `metho`
- Versión HTTP usando el método `httpVersion`
- URL usando el método de `URL`
- socket subyacente usando el método de `socket`

Se accede a los datos mediante flujos, ya que `http.IncomingMessage` implementa el `Readable` Interfaz de transmisión.

# Corrientes

**Aprenda para qué sirven las transmisiones, por qué son tan importantes y cómo usarlas.**

- [que son las corrientes](#)
- [¿Por qué corrientes?](#)
- [Un ejemplo de una tubería de flujo \(\)](#)
- [API de nodo impulsadas por Streams](#)
- [Diferentes tipos de corrientes](#)
- [Cómo crear una transmisión legible](#)
- [Cómo crear un flujo de escritura](#)
- [Cómo obtener datos de un flujo legible](#)
- [Cómo enviar datos a un flujo de escritura](#)
- [Señalar una secuencia grabable que terminaste de escribir](#)
- [Conclusión](#)

## que son las corrientes

Los flujos son uno de los conceptos fundamentales que impulsan las aplicaciones de Node.js.

Son una forma de manejar archivos de lectura/escritura, comunicaciones de red o cualquier tipo de intercambio de información de extremo a extremo de manera eficiente.

Las transmisiones no son un concepto exclusivo de Node.js. Se introdujeron en el sistema operativo Unix hace décadas y los programas pueden interactuar entre sí pasando flujos a través del operador de tubería ( | ).

Por ejemplo, en la forma tradicional, cuando le dices al programa que lea un archivo, el archivo se lee en la memoria, de principio a fin, y luego lo procesas.

Usando secuencias, lo lee pieza por pieza, procesando su contenido sin guardar todo en memoria.

El [módulo de transmisión](#) de Node.js proporciona la base sobre la cual se construyen todas las API de transmisión.

## ¿Por qué corrientes?

Los flujos básicamente brindan dos ventajas principales al usar otros métodos de manejo de datos:

- **Eficiencia de la memoria:** no necesita cargar grandes cantidades de datos en la memoria antes de

son capaces de procesarlo

- **Eficiencia de tiempo:** toma mucho menos tiempo comenzar a procesar datos tan pronto como los tenga, en lugar de esperar hasta que toda la carga útil de datos esté disponible para comenzar

## Un ejemplo de un flujo

Un ejemplo típico es el de la lectura de archivos de un disco.

Con el módulo Node fs , puede leer un archivo y servirlo a través de HTTP cuando se establece una nueva conexión con su servidor http:

```
const http = require('http') const fs =
require('fs')

const servidor = http.createServer(función (req, res) {
  fs.readFile(__dirname + '/datos.txt', (err, datos) => {
    res.end(datos) })
  server.listen(3000)
```

readFile() lee el contenido completo del archivo e invoca la función de devolución de llamada cuando termina.

res.end(data) en la devolución de llamada devolverá el contenido del archivo al cliente HTTP.

Si el archivo es grande, la operación llevará bastante tiempo. Aquí está lo mismo escrito usando corrientes:

```
const http = require('http') const fs =
require('fs')

const servidor = http.createServer((req, res) => {
  flujo const = fs.createReadStream(__dirname + '/data.txt') stream.pipe(res) })
  server.listen(3000)
```

En lugar de esperar hasta que el archivo se lea por completo, comenzamos a transmitirlo al cliente HTTP tan pronto como tengamos una parte de los datos listos para enviar.

## tubo()

El ejemplo anterior usa la línea stream.pipe(res) : se llama al método pipe() en el archivo corriente.

¿Qué hace este código? Toma la fuente y la canaliza a un destino.

Lo llama en el flujo de origen, por lo que en este caso, el flujo de archivos se canaliza a la respuesta HTTP.

El valor de retorno del método pipe() es el flujo de destino, que es algo muy conveniente que nos permite encadenar varias llamadas de pipe() , como esta:

```
src.pipe(destino1).pipe(destino2)
```

Esta construcción es lo mismo que hacer

```
src.tubería(destino1)  
destino1.tubería(destino2)
```

## API de nodo impulsadas por Streams

Debido a sus ventajas, muchos módulos centrales de Node.js brindan capacidades nativas de manejo de secuencias, en particular:

- `process.stdin` devuelve una secuencia conectada a `stdin`
- `process.stdout` devuelve una secuencia conectada a `stdout`
- `process.stderr` devuelve una secuencia conectada a `stderr`
- `fs.createReadStream()` crea una secuencia legible en un archivo
- `fs.createWriteStream()` crea una secuencia grabable en un archivo
- `net.connect()` inicia una conexión basada en flujo
- `http.request()` devuelve una instancia de la clase `http.ClientRequest`, que se puede escribir corriente
- `zlib.createGzip()` comprime datos utilizando gzip (un algoritmo de compresión) en un flujo `zlib.createGunzip()`
- descomprime un flujo gzip. `zlib.createDeflate()` comprime datos usando deflate (un algoritmo de compresión)
- en un corriente
- `zlib.createInflate()` descomprime un flujo desinflado

## Diferentes tipos de corrientes

Hay cuatro clases de flujos:

- Legible : un flujo desde el que puede canalizarse, pero no canalizarse (puede recibir datos, pero no enviarle datos). Cuando inserta datos en un flujo legible, se almacenan en búfer, hasta que un el consumidor comienza a leer los datos.
- Escrutable : un flujo en el que puede canalizarse, pero no canalizarse (puede enviar datos, pero no

recibir de él)

- Dúplex : un flujo en el que puede canalizarse y desde el que puede canalizarse, básicamente una combinación de un Flujo de lectura y escritura
- Transformar : un flujo de transformación es similar a un dúplex, pero la salida es una transformación de su entrada

## Cómo crear una transmisión legible

Obtenemos el flujo legible del [módulo de flujo](#) y lo inicializamos

```
const Stream = require('stream') const
readableStream = new Stream.Readable()
```

Ahora que la transmisión está inicializada, podemos enviarle datos:

```
readableStream.push('¡hola!')
readableStream.push('¡Hola!')
```

## Cómo crear un flujo de escritura

Para crear un flujo grabable, extendemos el objeto base grabable e implementamos su método `_write()` .

Primero crea un objeto de flujo:

```
const Stream = require('stream') const
writableStream = new Stream.Writable()
```

Luego implementar `_write` :

```
writableStream._write = (fragmento, codificación, siguiente) =>
{ console.log(fragmento.toString()) siguiente()
}
```

Ahora puede canalizar un flujo legible en:

```
proceso.stdin.pipe(writableStream)
```

## Cómo obtener datos de un flujo legible

¿Cómo leemos los datos de un flujo legible? Usando una secuencia de escritura:

```
const Flujo = require('flujo')

const readableStream = new Stream.Readable() const
writableStream = new Stream.Writable()

writableStream._write = (fragmento, codificación, siguiente) =>
{ console.log(fragmento.toString()) siguiente()

}

readableStream.pipe(writableStream)

readableStream.push('¡hola!')
readableStream.push('¡Hola!')
```

También puede consumir una transmisión legible directamente, utilizando el evento legible :

```
readableStream.on('legible', () =>
{ console.log(readableStream.read()) })
```

## Cómo enviar datos a un flujo de escritura

Usando el método stream write() :

```
writableStream.write('¡oye!\n')
```

## Señalar una secuencia grabable que terminaste de escribir

Usa el método end() :

```
const Flujo = require('flujo')

const readableStream = new Stream.Readable() const
writableStream = new Stream.Writable()

writableStream._write = (fragmento, codificación, siguiente) =>
{ console.log(fragmento.toString()) siguiente()

}

readableStream.pipe(writableStream)
```

```
readableStream.push('¡hola!')
readableStream.push('¡Hola!')

WritableStream.end()
```

## Conclusión

Esta es una introducción a las secuencias. Hay aspectos mucho más complicados de analizar, y espero cubrirlos pronto.

# Trabajando con MySQL

**MySQL es una de las bases de datos relacionales más populares del mundo.**  
**Descubra cómo hacerlo funcionar con Node.js**

MySQL es una de las bases de datos relacionales más populares del mundo.

El ecosistema Node, por supuesto, tiene varios paquetes diferentes que le permiten interactuar con MySQL, almacenar datos, recuperar datos, etc.

Usaremos [mysqljs/mysql](#), un paquete que tiene más de 12.000 estrellas de GitHub y existe desde hace años.

## Instalación del paquete Node mysql

Lo instalas usando

```
npm instalar mysql
```

## Iniciar la conexión a la base de datos

Primero incluyes el paquete:

```
const mysql = require('mysql')
```

y creas una conexión:

```
opciones constantes = {
    usuario: 'the_mysql_user_name',
    contraseña: 'the_mysql_user_password', base
    de datos: 'the_mysql_database_name'

} const conexión = mysql.createConnection(opciones)
```

Inicia una nueva conexión llamando:

```
connection.connect(err => { if (err)
    { console.error(' Ocurrió un error
        al conectarse a la base de datos')
        tirar errar

    } })
```

## Las opciones de conexión

En el ejemplo anterior, el objeto de opciones contenía 3 opciones:

```
opciones constantes = {
    usuario: 'the_mysql_user_name',
    contraseña: 'the_mysql_user_password',
    base de datos: 'the_mysql_database_name'
}
```

Hay muchos más que puedes usar, incluyendo:

- `anfitrón` , el nombre de host de la base de datos, por defecto es localhost
- `puerto` , el número de puerto del servidor MySQL, por defecto es 3306
- `socketPath` , se utiliza para especificar un socket de Unix en lugar de host y puerto
- `debug` , deshabilitado de manera predeterminada, se puede usar para depurar
- `rastro` , activado de forma predeterminada, imprime seguimientos de pila cuando se producen errores
- `SSL` , utilizado para configurar una conexión SSL al servidor (fuera del alcance de este tutorial)

## Realizar una consulta SELECT

Ahora está listo para realizar una consulta SQL en la base de datos. La consulta una vez ejecutada invoca una función de devolución de llamada que contiene un error eventual, los resultados y los campos.

```
connection.query('SELECT * FROM todos', (error, todos, campos) => {
    si (error) {
        console.error(' Ocurrió un error al ejecutar la consulta')
        lanzar error
    }
    consola.log(todos)
})
```

Puede pasar valores que se escaparán automáticamente:

```
id constante = 223
 conexión.consulta('SELECCIONAR * DESDE todos DONDE id = ?', [id], (error, todos, campos) => {
    si (error) {
        console.error(' Ocurrió un error al ejecutar la consulta')
        lanzar error
    }
    consola.log(todos)
})
```

Para pasar varios valores, simplemente coloque más elementos en la matriz que pasa como segundo parámetro:

```
id constante = 223
const autor = 'Flavio'

connection.query('SELECT * todos, DESDE todos DONDE id = ? AND autor = ?', [id, autor], (error,
campos) => { if (error) {

    console.error(' Ocurrió un error al ejecutar la consulta')
    lanzar error

} consola.log(todos) })
```

## Realizar una consulta INSERTAR

Puedes pasar un objeto

```
const todo =
{ cosa: 'Comprar la leche'
autor: 'Flavio'

} connection.query('INSERT INTO todos SET ?', todo, (error, resultados, campos) => {
si (error) {
    console.error(' Ocurrió un error al ejecutar la consulta')
    lanzar error

} })
```

Si la tabla tiene una clave principal con auto\_increment , el valor de eso será devuelto en el resultados.insertId valor:

```
const todo =
{ cosa: 'Comprar la leche'
autor: 'Flavio'

} connection.query('INSERT INTO todos SET ?', todo, (error, resultados, campos) => {
si (error) {
    console.error('Se produjo un error al ejecutar la consulta') arroja un error

}}
const id = resultados.insertId
consola.log(id)
})
```

## Cerrar la conexión

Cuando necesite terminar la conexión a la base de datos, puede llamar al método end() :

```
conexión.end()
```

Esto asegura que se envíe cualquier consulta pendiente y que la conexión finalice correctamente.

# Diferencia entre desarrollo y producción.

## Aprenda a configurar diferentes configuraciones para entornos de producción y desarrollo

Puede tener diferentes configuraciones para entornos de producción y desarrollo.

Node asume que siempre se ejecuta en un entorno de desarrollo. Puede señalar a Node.js que se está ejecutando en producción configurando la variable de entorno de producción NODE\_ENV= .

Esto generalmente se hace ejecutando el comando

```
exportar NODE_ENV=producción
```

en el shell, pero es mejor ponerlo en el archivo de configuración de su shell (por ejemplo, .bash\_profile con el shell Bash) porque, de lo contrario, la configuración no persistirá en caso de que se reinicie el sistema.

También puede aplicar la variable de entorno anteponiéndola a la inicialización de su aplicación dominio:

```
NODE_ENV=nodo de producción app.js
```

Esta variable de entorno es una convención que también se usa ampliamente en bibliotecas externas.

Establecer el entorno en producción generalmente asegura que

- el registro se mantiene al mínimo, nivel esencial Se
- realizan más niveles de almacenamiento en caché para optimizar el rendimiento

Por ejemplo, Pug, la biblioteca de plantillas utilizada por Express, compila en modo de depuración si NODE\_ENV no está configurado en producción . Las vistas expresas se compilan en cada solicitud en modo de desarrollo, mientras que en producción se almacenan en caché. Hay muchos más ejemplos.

Express proporciona ganchos de configuración específicos para el entorno, que se llaman automáticamente en función del valor de la variable NODE\_ENV:

```
app.configure('desarrollo', () => {
  //...
})
app.configure('producción', () => {
  //...
})
app.configure('producción', 'puesta en escena', () => {
  //...
```

```
}
```

Por ejemplo, puede usar esto para configurar diferentes controladores de errores para diferentes modos:

```
app.configure('desarrollo', () => {
  app.use(express.errorHandler({ dumpExceptions: true, showStack: true }));
}

app.configure('producción', () =>
  { app.use(express.errorHandler()) })
```