

LUCAS EGA ACADEMY

Node.js



BACKEND • SYSTEMS • PERFORMANCE

Curso Completo de Node.js

© 2026 Esteban Gutiérrez · Colibrí Editorial EGA · Hecho en México

lucasegaacademy.com

ÍNDICE

PRÓLOGO.....	4
UNIDAD 1: Fundamentos de Node.js: del navegador al servidor.....	7
UNIDAD 2: El sistema de módulos: require, import y globales.....	42
UNIDAD 3: Archivos y rutas: fs y path.....	85
UNIDAD 4: El sistema operativo y tu primer servidor HTTP.....	125
UNIDAD 5: NPM y herramientas de desarrollo.....	170
UNIDAD 6: El corazón asíncrono: eventos, streams y buffers.....	206
UNIDAD 7: Dominando la asincronía: callbacks, promesas y async/await.....	241
UNIDAD 8: Express desde cero: tu primer framework.....	283
UNIDAD 9: Rutas avanzadas, middleware y archivos estáticos.....	320
UNIDAD 10: APIs REST y proyecto final: tu backend completo.....	359
EXAMEN FINAL.....	398

PRÓLOGO

Bienvenido a "Curso Completo de Node.js", el libro oficial de Lucas EGA Academy (lucasegaacademy.com) para convertirte en programador backend con JavaScript. Si ya sabes algo de JavaScript del navegador, estás a un paso de algo grande: usar ese mismo lenguaje para construir servidores, APIs y aplicaciones completas del lado del servidor. Eso es Node.js, y este libro te lleva de cero hasta tu primer backend funcional, paso a paso y sin saltos.

Por qué este libro existe

Hay miles de tutoriales de Node.js en internet, pero la mayoría comete los mismos errores: o te avientan código sin explicar qué está pasando, o se quedan en la teoría y nunca construyes nada real. Este libro nació del curso en video de 30 clases de Lucas EGA Academy, y conserva su filosofía: cada concepto se explica con palabras sencillas, se muestra con código que sí corre, se practica con ejercicios y se repasa al final. Nada de "cópialo y confía". Aquí vas a entender el porqué de cada línea.

Cómo está organizado

El libro tiene 10 unidades, y cada unidad agrupa 3 clases del curso original:

- UNIDAD 1: Fundamentos de Node.js: del navegador al servidor
- UNIDAD 2: El sistema de módulos: require, import y globales
- UNIDAD 3: Archivos y rutas: fs y path
- UNIDAD 4: El sistema operativo y tu primer servidor HTTP
- UNIDAD 5: NPM y herramientas de desarrollo
- UNIDAD 6: El corazón asíncrono: eventos, streams y buffers
- UNIDAD 7: Dominando la asincronía: callbacks, promesas y async/await
- UNIDAD 8: Express desde cero: tu primer framework
- UNIDAD 9: Rutas avanzadas, middleware y archivos estáticos
- UNIDAD 10: APIs REST y proyecto final: tu backend completo

Cada unidad sigue la misma estructura para que siempre sepas qué esperar: una introducción que te ubica, una explicación detallada con analogías de la vida real y errores comunes, ejemplos paso a paso con código completo, una sección de práctica con ejercicios y sus soluciones explicadas, y un resumen final con los puntos clave.

Al final del libro encontrarás el EXAMEN FINAL: 50 preguntas que cubren todo el curso, con sus respuestas y soluciones explicadas. Úsalo como tu prueba de graduación.

Cómo estudiar este libro

No lo leas como novela. La programación se aprende con las manos en el teclado, así que te recomiendo este método:

1. Lee la explicación detallada de la unidad con calma. Si algo no queda claro a la primera, relee: los conceptos de asincronía, por ejemplo, les cuestan a todos al inicio.
2. Escribe TÚ MISMO cada ejemplo de código, no lo copies y pegues. Tus dedos también tienen que aprender.
3. Corre cada ejemplo en tu terminal y rompe cosas a propósito: cambia valores, quita líneas, provoca errores. Ver el error y entenderlo vale más que diez lecturas.
4. Haz los ejercicios de práctica sin mirar la solución primero. Si te atorras más de 20 minutos, mira la solución, entiéndela y luego ciérrala y resuélvelo solo.
5. Al terminar cada unidad, lee el resumen y pregúntate: ¿podría explicarle esto a alguien más? Si la respuesta es sí, vas bien.

Necesitas una computadora con Node.js instalado (versión LTS, la unidad 1 te guía en la instalación) y un editor de código. No necesitas saber nada de servidores: empezamos desde qué es un servidor.

Una nota sobre los errores

Vas a ver muchos errores en este camino: "cannot find module", "port already in use", "req.body is undefined". Es normal. Los errores no son fracasos, son el lenguaje con el que la computadora te dice qué necesita. Este libro dedica espacio a los errores más comunes de cada tema y cómo resolverlos, porque un programador no es alguien que no se equivoca: es alguien que sabe leer los errores y seguir adelante.

Échale ganas, ve a tu ritmo y no te saltes las prácticas. Al final de este libro no solo vas a "saber Node.js": vas a tener un proyecto real funcionando, una API REST de tareas conectada a un frontend, hecha con tus propias manos. Eso ya es ser programador backend.

Nos vemos en la unidad 1. A programar.

Esteban Lucas EGA Academy - lucasegaacademy.com

UNIDAD 1: Fundamentos de Node.js: del navegador al servidor

Introducción

Bienvenido a la primera unidad del Curso Completo de Node.js de Lucas EGA Academy. Si llegaste hasta aquí, lo más probable es que ya sepas algo de JavaScript: variables, funciones, arreglos, objetos y cómo hacer que una página web reaccione cuando el usuario hace clic en un botón. Todo eso que aprendiste vive dentro del navegador. Pues bien, esta unidad marca el momento en que JavaScript sale del navegador y se va a trabajar al servidor, a la terminal y a lugares donde antes ni siquiera lo dejaban entrar. Ese salto, de la página web al servidor, es exactamente lo que Node.js hace posible, y entenderlo a fondo es la base de todo lo que construiremos en este libro.

Empecemos con la pregunta que todo el mundo se hace al principio: ¿por qué debería importarme Node.js? La respuesta corta es que Node.js te convierte en un programador completo con un solo lenguaje. Antes de que Node.js existiera, el mundo del desarrollo web estaba dividido en dos tribus que casi no se hablaban: los programadores de frontend, que escribían JavaScript para el navegador, y los programadores de backend, que escribían PHP, Java, Python o Ruby para el servidor. Si querías construir una aplicación completa, necesitabas aprender dos lenguajes, dos ecosistemas y dos formas de pensar. Node.js rompió esa pared. Con Node.js, el mismo JavaScript que usas para validar un formulario en el navegador sirve para recibir peticiones en un servidor, guardar datos en una base de datos, leer archivos del disco y construir la lógica completa de una aplicación. Un solo lenguaje de punta a punta. Eso no es un detalle menor: significa que lo que ya sabes vale el doble, que aprendes más rápido porque no cambias de chip mental, y que puedes construir proyectos completos tú solo, sin depender de nadie más.

La respuesta larga tiene que ver con el mundo real y con el trabajo. Node.js no es un juguete ni una moda pasajera: es la tecnología con la que están contruidos los servidores de Netflix, Uber, LinkedIn, PayPal, Walmart y hasta sistemas de la NASA. Cuando pides un Uber, hay Node.js decidiendo qué conductor está más cerca de ti. Cuando ves una serie en Netflix, hay Node.js sirviendo parte de esa experiencia. Y detrás de todo eso hay miles de empresas en México y América Latina contratando programadores que sepan Node.js, muchas veces permitiendo trabajo remoto y pagando en dólares. Aprender Node.js hoy es una de las apuestas más seguras que puedes hacer con tu tiempo si quieres vivir del desarrollo de software. No es magia ni es suerte: es oferta y demanda. Hay más puestos que programadores, y eso no va a cambiar pronto.

En esta unidad vamos a cubrir las primeras tres clases del curso, que son los cimientos de todo el edificio. La clase 1 responde qué es Node.js: qué significa que sea un "entorno de ejecución", por qué nació en 2009 de la mano de Ryan Dahl, qué papel juega el motor V8 de Chrome y cómo funciona por dentro ese mecanismo llamado event loop que le permite atender miles de peticiones con un solo hilo. La clase 2 es práctica desde el primer minuto: vamos a instalar Node.js y npm en Windows, Mac y Linux, paso a paso, sin dar nada por sentado. Verás la diferencia entre la versión LTS y la versión Current, aprenderás a verificar tu instalación con los comandos `node -v` y `npm -v`, conocerás el REPL (esa consola interactiva donde puedes probar JavaScript al instante) y sabrás qué hacer cuando algo salga mal durante la instalación, porque algo siempre sale mal la primera vez y no tiene nada de malo. La clase 3 cierra la unidad con tu primer script real: un archivo `hola.js` con un `console.log`, ejecutado desde la terminal con el comando `node hola.js`, sin navegador, sin HTML, sin página web de por medio. Parece poca cosa, pero ese momento en que tu código corre fuera del navegador por primera vez es el momento en que te conviertes en programador backend. No lo subestimes.

Esta unidad está escrita para ti si ya manejas lo básico de JavaScript. No necesitas ser un experto: con que sepas declarar variables, escribir funciones, usar arreglos y objetos, y entender a grandes rasgos qué es una función callback, tienes suficiente. Si todavía no sabes JavaScript, te recomiendo pausar aquí y tomar primero el curso de JavaScript de la academia, porque este libro da por sentado esas bases y no se detiene a explicarlas. No necesitas saber HTML ni CSS para este libro: del lado del servidor no hay páginas ni estilos, solo lógica, datos y terminal. Tampoco necesitas saber nada de servidores, redes o sistemas operativos: todo eso lo vamos a construir desde cero, con calma y con ejemplos.

Un consejo antes de empezar, y tómallo en serio: este libro no se lee, se trabaja. La programación no se aprende mirando código, se aprende escribiendo código, rompiéndolo y arreglándolo. Cada vez que veas un bloque de código en estas páginas, escríbelo tú mismo en

tu computadora, ejecútalo, cámbiale cosas a propósito para ver qué pasa y equivócate sin miedo. El que se equivoca y arregla su error aprende diez veces más que el que solo lee la solución correcta. Ten tu terminal abierta mientras lees, ten tu editor de texto listo, y convierte cada ejemplo en un experimento. Si un concepto no te queda claro a la primera, no te frustres: sigue leyendo, haz los ejercicios de la sección de práctica, y vuelve a leerlo después. Los conceptos difíciles de esta unidad, como el event loop, se entienden mejor la segunda vez, cuando ya tienes las manos llenas de código.

Otra cosa: no te saltes la instalación pensando que "eso ya lo sé" o que "lo haré después". Instala Node.js hoy, en cuanto termines de leer la clase 2. Todo el resto del libro depende de que tengas Node.js funcionando en tu máquina, y los errores de instalación son mucho más fáciles de resolver ahorita, cuando apenas empiezas, que dentro de tres semanas cuando estés atorado en un proyecto y descubras que tu instalación estaba mal desde el principio. Hazlo bien desde el día uno y el resto del camino será cuesta abajo.

Para cerrar esta introducción, una analogía que nos va a acompañar todo el libro. Piensa en JavaScript como un gran cocinero. Durante años, ese cocinero solo podía trabajar dentro de un restaurante muy específico: el navegador. Ahí cocinaba cosas ricas (páginas interactivas), pero no podía salir de la cocina. Node.js es lo que pasa cuando a ese cocinero le construyen su propia cocina, con sus propios utensilios, y le dicen: ahora puedes cocinar donde quieras, para quien quieras y lo que quieras. El cocinero es el mismo, las recetas son las mismas, pero el restaurante nuevo es muchísimo más grande. En esta unidad vamos a conocer esa cocina nueva: sus herramientas, sus reglas y su forma de trabajar. Vámonos recio.

Explicación detallada

1.1 Qué es Node.js: el entorno que saca a JavaScript del navegador

Node.js es un entorno de ejecución de JavaScript que funciona fuera del navegador. Esa frase, aunque corta, contiene tres ideas que hay que desmenuzar con calma, porque cada palabra importa.

Primero: "entorno de ejecución". En programación, un entorno de ejecución (en inglés, runtime environment) es el programa que se encarga de tomar tu código y hacerlo correr de verdad en la computadora. Tu código JavaScript, por sí solo, es solo texto en un archivo: letras y símbolos que no hacen nada hasta que algo los interpreta y los ejecuta. En el navegador, ese "algo" es el motor JavaScript integrado al navegador. En el servidor, ese "algo" es Node.js. Dicho de otra forma: Node.js es el programa que lee tu archivo .js, entiende cada instrucción y la ejecuta usando los recursos de la computadora (el procesador, la memoria, el disco duro, la

red). Sin un entorno de ejecución, tu código es como una partitura sin músicos: las notas están ahí, pero nadie las toca.

Una analogía mexicana sencilla: imagina que tu código JavaScript es una receta de cocina escrita en un papel. La receta sola no cocina nada. Necesitas una cocina con estufa, ingredientes y utensilios para convertir esa receta en un platillo real. El navegador es una cocina: tiene su estufa (el motor JavaScript) y te deja cocinar, pero es una cocina chiquita, pensada solo para antojitos (páginas web). Node.js es otra cocina, mucho más grande y equipada: tiene estufa industrial, horno de barro, refrigeradores gigantes y acceso directo a la bodega. La receta (tu código) puede ser la misma, pero en la cocina grande puedes preparar banquetes completos (servidores, aplicaciones, herramientas). Node.js no es un lenguaje nuevo: es la cocina nueva donde tu JavaScript de siempre puede cocinar cosas mucho más grandes.

Segundo: "de JavaScript". Esto es clave y es lo que hace a Node.js tan especial: no inventó un lenguaje nuevo. Usa el mismo JavaScript que ya conoces, con la misma sintaxis, las mismas funciones y las mismas reglas. Lo que cambia no es el lenguaje, sino lo que el lenguaje puede hacer. Dentro del navegador, JavaScript tiene superpoderes para manipular páginas web (el famoso DOM), pero está encerrado en una jaula de seguridad: no puede leer archivos de tu disco duro, no puede abrir conexiones de red libremente, no puede ejecutar otros programas. Esa jaula existe por una buena razón: si cualquier página web pudiera leer tus archivos, Internet sería un desastre de virus y robos. Pero del lado del servidor no necesitas esa jaula, porque ahí tú eres el dueño de la computadora. Node.js le quita la jaula a JavaScript y a cambio le da herramientas nuevas: acceso al sistema de archivos, a la red, a los procesos del sistema operativo. Es el mismo perro, pero ahora sí lo dejan salir al patio completo.

Tercero: "fuera del navegador". Este es el cambio de paradigma. Durante más de una década, JavaScript fue un lenguaje de un solo hábitat: el navegador. Todo lo que aprendías de JavaScript servía para una sola cosa: páginas web. Node.js rompió esa limitación y convirtió a JavaScript en un lenguaje de propósito general, capaz de correr en servidores, en tu computadora personal, en dispositivos pequeños e incluso en robots. Cuando ejecutas `node hola.js` en tu terminal, no hay navegador involucrado en absoluto: es Node.js leyendo tu archivo y ejecutándolo directamente sobre el sistema operativo. Ese es el salto mental que tienes que dar en esta unidad: JavaScript ya no es "el lenguaje de las páginas web", es un lenguaje completo que también hace páginas web.

Técnicamente, Node.js está construido sobre el motor V8 de Google Chrome, que es el componente que ejecuta JavaScript dentro del navegador Chrome. Los creadores de Node.js tomaron ese motor, lo sacaron del navegador y lo combinaron con otras piezas (principalmente una biblioteca llamada `libuv`, que maneja las operaciones de entrada y salida de forma no

bloqueante) para crear un programa independiente que corre en el servidor. Pero no te preocupes si estos nombres todavía no te dicen nada: en las siguientes secciones vamos a conocer a cada uno de estos personajes con calma, cómo se debe.

1.2 Historia breve: cómo un programador frustrado cambió la web en 2009

Para entender por qué Node.js existe, hay que entender el problema que lo hizo nacer. Y ese problema tiene nombre: los servidores bloqueantes.

Antes de 2009, la forma más común de construir aplicaciones web era con lenguajes como PHP corriendo sobre servidores como Apache. El modelo era sencillo: cada vez que un usuario pedía una página, el servidor asignaba un hilo (thread) o un proceso completo para atender a ese usuario, y ese hilo se quedaba ocupado hasta terminar de responder. Si cien usuarios pedían una página al mismo tiempo, el servidor necesitaba cien hilos. Si llegaban diez mil usuarios, necesitaba diez mil hilos. Cada hilo consume memoria, y crear y destruir hilos cuesta tiempo y recursos. Era como un restaurante donde cada cliente necesita su propio mesero personal que no puede atender a nadie más hasta que su cliente termina de comer: funciona con diez clientes, pero con mil clientes necesitas mil meseros y el restaurante quiebra pagando sueldos.

Ryan Dahl, un programador estadounidense, estaba frustrado exactamente con eso. Veía cómo los servidores desperdiciaban recursos esperando: esperando a que la base de datos respondiera, esperando a que un archivo se leyera del disco, esperando a que otro servidor contestara. La computadora pasaba la mayor parte del tiempo sin hacer nada, solo esperando, mientras los hilos ocupaban memoria a lo tonto. Dahl tuvo una idea que en ese momento sonó loca: ¿y si en lugar de asignar un hilo por cliente, usamos un solo hilo que nunca se bloquee? ¿Y si cuando una operación tarda (cómo leer un archivo), en lugar de quedarse esperando, el programa sigue atendiendo a otros clientes y retoma la tarea cuando el resultado esté listo? Esa idea, que hoy nos parece obvia, en 2009 era revolucionaria para el desarrollo web.

Para construir su idea, Dahl necesitaba un motor de JavaScript rápido, porque su plan era usar JavaScript como lenguaje del servidor. Probó varios motores y se decidió por V8, el motor que Google acababa de lanzar dentro de Chrome. V8 era notablemente más rápido que todos los demás gracias a su compilación justo a tiempo (JIT), una técnica que convierte el código a lenguaje máquina sobre la marcha en lugar de interpretarlo línea por línea. Era el motor más veloz del mercado, y además era de código abierto, así que Dahl podía tomarlo y usarlo libremente.

En noviembre de 2009, Ryan Dahl presentó Node.js en la conferencia JSConf EU en Berlín. La presentación duró menos de una hora y la sala se puso de pie a aplaudir. Los programadores

que estaban ahí entendieron al instante lo que estaban viendo: JavaScript corriendo en el servidor, atendiendo miles de conexiones con un solo hilo, a una velocidad que nadie había visto. Fue uno de esos momentos históricos de la tecnología, como cuando se presentó la primera Mac o el primer iPhone: la gente en la sala sabía que acababa de ver el futuro.

Después de la presentación, una empresa llamada Joyent (que ofrecía servicios de nube) decidió patrocinar el proyecto y contratar a Dahl para trabajar en Node.js de tiempo completo. En 2011 nació npm, el manejador de paquetes de Node.js, creado por Isaac Schlueter, y con él explotó el ecosistema: de pronto cualquier programador podía publicar una biblioteca de código y cualquier otro podía instalarla con un solo comando. Hoy npm es el registro de software más grande del mundo, con millones de paquetes disponibles. Con el tiempo, el proyecto pasó a manos de la OpenJS Foundation (antes Fundación Node.js), que lo mantiene hasta hoy como un proyecto de código abierto con cientos de contribuidores.

¿Y en la industria? La adopción fue brutal. LinkedIn migró su aplicación móvil a Node.js y reportó que con una décima parte de los servidores atendía veinte veces más tráfico. PayPal reconstruyó parte de su plataforma con Node.js y redujo a la mitad el tiempo de desarrollo. Netflix, Uber, Walmart, eBay, la NASA: todos usan Node.js en producción, en sistemas que atienden a millones de personas todos los días. No es una tecnología experimental ni una promesa: es infraestructura real del Internet moderno, probada bajo el fuego de las empresas más exigentes del planeta.

Moraleja de la historia para ti como estudiante: Node.js no nació porque a alguien se le ocurrió que sería divertido poner JavaScript en el servidor. Nació para resolver un problema real y doloroso (el desperdicio de recursos de los servidores bloqueantes), con una idea técnicamente elegante (un solo hilo no bloqueante con event loop) y sobre los hombros de la mejor tecnología disponible (el motor V8). Cuando entiendes el problema que una tecnología resuelve, la tecnología misma se vuelve mucho más fácil de entender. Guarda esta lección: cada vez que aprendas una herramienta nueva en este libro, pregúntate primero qué problema resuelve. Esa pregunta vale más que cien páginas de documentación.

1.3 El motor V8: el corazón que late dentro de Node.js

Ya mencionamos varias veces al motor V8, así que vamos a conocerlo bien, porque es literalmente el corazón de Node.js: sin V8, no hay Node.js.

Un motor de JavaScript es el programa que toma tu código JavaScript y lo ejecuta. Cuando escribes `console.log("hola")` en la consola del navegador, es el motor del navegador el que lee ese texto, lo entiende y hace que aparezca el mensaje. Cada navegador tiene su motor: Chrome y Edge usan V8, Firefox usa SpiderMonkey, Safari usa JavaScriptCore. Todos hacen

lo mismo (ejecutar JavaScript), pero cada uno lo hace a su manera y con su propio rendimiento.

V8 es el motor que Google desarrolló para Chrome, escrito en el lenguaje C++. Su fama viene de una técnica llamada compilación justo a tiempo, conocida como JIT (Just-In-Time compilation). Para entender por qué esto importa, piensa en dos formas de seguir una receta de cocina escrita en otro idioma. La forma lenta (interpretación pura) es traducir cada paso justo cuando lo vas a ejecutar: lees el paso 1, lo traduces, lo haces; lees el paso 2, lo traduces, lo haces. Si la receta tiene cien pasos, traduces cien veces. La forma rápida (compilación JIT) es traducir toda la receta una vez al principio y luego seguirla a toda velocidad en tu idioma. V8 hace algo parecido: en lugar de interpretar tu JavaScript línea por línea cada vez, lo compila a código máquina (el idioma nativo del procesador) y lo ejecuta directamente. El resultado es una velocidad brutal, y esa velocidad es parte de por qué Node.js puede atender tantas peticiones tan rápido.

Pero ojo con un malentendido común: Node.js NO es V8. V8 es solo el motor que ejecuta el JavaScript. Node.js es V8 más un montón de piezas adicionales que V8 no tiene: acceso al sistema de archivos, herramientas de red para crear servidores, el sistema de módulos para organizar el código, la biblioteca libuv para operaciones no bloqueantes, y el propio programa node que instalas en tu computadora. Una forma de verlo: V8 es el motor de un coche, Node.js es el coche completo con motor, llantas, volante y carrocería. El motor solo no te lleva a ningún lado; necesitas todo lo demás para tener un vehículo funcional.

Otro detalle importante: V8 dentro de Node.js es el mismo V8 de Chrome, pero sin las partes del navegador. En el navegador, V8 viene acompañado del DOM (para manipular páginas), de window, de document, de alert y de todas esas cosas que solo tienen sentido en una página web. En Node.js, nada de eso existe: no hay window, no hay document, no hay DOM, porque no hay página web. En su lugar, Node.js te da otros objetos globales: process (información y control del proceso actual), Buffer (para manejar datos binarios), __dirname (te dice en qué carpeta está tu archivo) y módulos como fs (sistema de archivos) y http (servidor web). Si vienes del JavaScript del navegador y un día intentas usar document en Node.js, te vas a topar con un error que dice que document no está definido. No es un bug: es que estás en otra cocina, con otros utensilios. Acostúmbrate a esa diferencia desde ahorita y te ahorrarás muchas confusiones.

V8 se actualiza constantemente: Google lo mejora semana tras semana para Chrome, y Node.js va adoptando esas mejoras en sus nuevas versiones. Esto significa que el JavaScript moderno (las versiones recientes del lenguaje, con clases, async/await, módulos y demás) funciona en Node.js casi tan pronto cómo funciona en Chrome. No tienes que esperar años para usar las novedades del lenguaje, como pasaba antes con otros entornos. Es una de las ventajas silenciosas de Node.js: siempre hablas un JavaScript fresco y actual.

1.4 El event loop: el mesero que atiende a todo el restaurante sin sentarse

Llegamos al concepto más importante de toda la unidad, y probablemente de todo el libro: el event loop. Si entiendes el event loop, entiendes el alma de Node.js. Si no lo entiendes, Node.js te va a parecer magia negra cada vez que algo se comporte de forma "rara". Así que vamos a desmenuzarlo con la analogía prometida: la del mesero y la taquería.

Imagina una taquería de barrio en hora pico: las mesas están llenas, la gente entra y sale, todos quieren sus tacos al pastor. Ahora imagina dos formas de atender ese restaurante.

La forma tradicional (el modelo de hilos, como Apache con PHP): cada mesa que llega recibe su propio mesero personal. El mesero toma la orden, va a la cocina, se queda parado esperando a que los tacos estén listos, los lleva a la mesa y no atiende a nadie más hasta que esa mesa termina y se va. Con 10 mesas necesitas 10 meseros. Con 200 mesas necesitas 200 meseros. La mayoría del tiempo esos meseros están parados sin hacer nada, solo esperando a que la cocina termine. Están ocupados, pero no están trabajando: están esperando. Y cada mesero cuesta dinero (en una computadora, cada hilo cuesta memoria). Este es exactamente el problema que frustraba a Ryan Dahl: los servidores pasaban la vida esperando y gastando recursos en esa espera.

La forma Node.js (el event loop): hay UN SOLO mesero para todo el restaurante, pero es un mesero con una técnica especial. Toma la orden de la mesa 1 y la pasa a la cocina. En lugar de quedarse parado esperando, va a la mesa 2, toma su orden y la pasa a la cocina. Luego a la mesa 3, a la mesa 4, y así con todas. Cuando la cocina termina los tacos de la mesa 1, le avisa al mesero (le grita "¡órale, mesa 1!"), y el mesero deja lo que está haciendo un segundo, lleva los tacos a la mesa 1 y sigue con lo suyo. El mesero nunca se queda esperando: siempre está en movimiento, siempre está haciendo algo útil. Un solo mesero atiende a todo el restaurante porque nunca se bloquea esperando a la cocina.

Ese mesero es el event loop. Técnicamente, el event loop es un mecanismo que corre en un solo hilo y se la pasa dando vueltas (de ahí lo de "loop", que significa ciclo o vuelta) revisando si hay trabajo pendiente: ¿llegó una nueva petición? ¿terminó de leerse un archivo? ¿ya respondió la base de datos? ¿se cumplió un temporizador? Cada vuelta del ciclo revisa la lista

de pendientes, ejecuta lo que esté listo y sigue girando. Nunca se detiene, nunca se bloquea, nunca se queda dormido esperando.

Vamos a verlo con un ejemplo concreto de código, porque aquí es donde la analogía se vuelve real. Imagina este script:

```
console.log("1. Tomo la orden de la mesa 1");

setTimeout(function() {
  console.log("2. La cocina terminó los tacos de la mesa 1");
}, 2000);

console.log("3. Tomo la orden de la mesa 2");
```

Si vienes de un pensamiento "tradicional", esperarías que la salida fuera 1, 2, 3, en orden. Pero en Node.js la salida es:

```
1. Tomo la orden de la mesa 1
3. Tomo la orden de la mesa 2
2. La cocina terminó los tacos de la mesa 1
```

¿Qué pasó? El event loop ejecutó la primera línea de inmediato (tomó la orden de la mesa 1). Luego se topó con `setTimeout`, que es como pasarle la orden a la cocina: "avísame en 2 segundos". El event loop NO se quedó esperando esos 2 segundos: registró el temporizador y siguió adelante con la siguiente línea (tomó la orden de la mesa 2). Cuando pasaron los 2 segundos, la "cocina" le avisó y el event loop ejecutó la función del temporizador. El programa nunca se detuvo. Esa es la esencia de lo no bloqueante: cuando algo tarda, Node.js no espera, sigue trabajando y retoma después.

Ahora, ¿dónde "esperan" las cosas mientras el event loop sigue girando? Aquí entra otro personaje: `libuv`. `libuv` es la biblioteca (escrita en C) que Node.js usa para manejar las operaciones de entrada y salida: leer archivos, hablar por la red, consultar temporizadores. Cuando tu código pide leer un archivo, el event loop no lo lee él mismo (eso lo bloquearía): le pasa la tarea a `libuv`, y `libuv` la resuelve por debajo, usando al sistema operativo o un pequeño grupo de hilos trabajadores para las operaciones pesadas. Cuando la operación termina, `libuv` pone el resultado en una cola, y en la siguiente vuelta del event loop, tu función callback se ejecuta con ese resultado. Tú nunca ves a `libuv` directamente, pero es la "cocina" de nuestra analogía: el lugar donde se preparan los tacos mientras el mesero sigue atendiendo mesas.

Hay un matiz honesto que debes conocer: Node.js sí usa varios hilos por dentro (`libuv` mantiene un grupo de hilos trabajadores para ciertas operaciones cómo leer archivos o

comprimir datos), pero TU código JavaScript corre en un solo hilo. Tú como programador no tienes que coordinar hilos, ni poner candados, ni preocuparte de que dos hilos modifiquen la misma variable al mismo tiempo, que son las pesadillas clásicas de la programación multihilo. Node.js te da lo mejor de los dos mundos: la simplicidad mental de un solo hilo para tu código, y el trabajo pesado en segundo plano cortesía de libuv. Esa simplicidad es una de las razones por las que tanta gente ama Node.js: los programas concurrentes son mucho más fáciles de escribir y de razonar.

Para que el event loop funcione bien, hay una regla de oro que debes grabarte desde hoy: nunca bloques el event loop. Como tu código corre en el único hilo del mesero, si tú escribes una operación que tarda mucho y es bloqueante (por ejemplo, un ciclo que hace millones de cálculos pesados de forma síncrona), el mesero se queda atorado contigo y todo el restaurante se detiene: ninguna otra petición se atiende, ningún temporizador se cumple, todo se congela hasta que termines. En la práctica esto significa: para operaciones lentas (leer archivos, pedir datos por la red, consultar bases de datos) usa SIEMPRE las versiones no bloqueantes (con callbacks, promesas o async/await, que veremos en unidades posteriores), y evita los cálculos larguísimos en el hilo principal. Un mesero atorado es un restaurante muerto. Esta regla va a reaparecer en todo el libro, así que mejor que nos hagamos amigos de ella desde ahorita.

1.5 Un solo hilo no bloqueante vs. el modelo tradicional de hilos

Vamos a comparar los dos modelos frente a frente, porque entender sus diferencias te va a servir no solo para Node.js, sino para toda tu carrera como programador backend.

El modelo tradicional (un hilo por petición) funciona así: llega un cliente, el servidor le asigna un hilo; el hilo atiende al cliente de principio a fin; cuando termina, el hilo se libera. Es un modelo fácil de entender porque es secuencial: cada hilo hace una sola cosa a la vez, como un cajero de banco que atiende a una persona hasta terminar. El problema es la escala. Cada hilo consume memoria (típicamente entre 1 y 8 megabytes solo por existir, más lo que use tu programa), y el sistema operativo tiene un límite de cuántos hilos puede manejar eficientemente. Con cientos de clientes simultáneos funciona bien. Con miles, el servidor empieza a jadear. Con decenas de miles de conexiones simultáneas (lo que hoy se conoce como "el problema de las 10 mil conexiones"), el modelo se cae: no hay memoria suficiente para tantos hilos y el procesador se la pasa cambiando de un hilo a otro (a eso se le llama cambió de contexto) en lugar de hacer trabajo útil. Es el restaurante con mil meseros: el gasto en sueldos te quiebra antes de que puedas atender a todos.

El modelo de Node.js (un solo hilo con event loop) funciona al revés: no asigna un hilo por cliente. Un solo hilo atiende a todos los clientes por turnos rapidísimos, y cuando una

operación tarda (leer disco, red, base de datos), el hilo no espera: registra un callback y sigue con el siguiente cliente. La memoria que consume es mínima y constante, sin importar si hay 100 o 100,000 clientes conectados, porque no se crea nada nuevo por cliente. El procesador no pierde tiempo cambiando de contexto entre miles de hilos, porque casi no hay hilos. El resultado: con los mismos fierros (el mismo servidor, la misma memoria), Node.js puede atender muchísimas más conexiones simultáneas que un servidor tradicional. ¿Recuerdas el dato de LinkedIn? Con una décima parte de los servidores, veinte veces más tráfico. Esa es la diferencia entre los dos modelos, medida en el mundo real.

Otra analogía para que quede blindado: piensa en las cajas de un supermercado. El modelo tradicional es como tener una caja por cliente: cada cliente tiene su cajero personal que lo atiende de principio a fin. Funciona, pero necesitas un cajero por cliente y la mayoría del tiempo los cajeros están esperando a que el cliente saque su tarjeta o cuente sus monedas. El modelo Node.js es como tener UN cajero súper eficiente para todos: escanea los productos del cliente 1, y mientras el cliente 1 busca su tarjeta, ya está escaneando los productos del cliente 2; cuando el cliente 1 tiene lista la tarjeta, regresa un segundo, cobra, y sigue con el cliente 3. Un solo cajero atiende a todos porque nunca se queda esperando a nadie. ¿Cuál supermercado atiende más gente con menos personal? El segundo, por goleada.

Ahora, seamos justos: el modelo tradicional no es tonto, y tiene sus ventajas. Es más fácil de razonar para tareas donde cada petición necesita mucho cálculo del procesador, porque ahí sí puedes repartir el trabajo entre varios hilos y varios núcleos del procesador. Node.js, con su solo hilo para tu código, no aprovecha múltiples núcleos del procesador de forma automática: si tu tarea es hacer cálculos matemáticos pesadísimos, un solo hilo de Node.js usará un solo núcleo mientras los demás descansan. Para esos casos existen soluciones (como los worker threads o dividir el trabajo en varios procesos), pero es importante que sepas desde ahorita que el punto fuerte de Node.js no es el cálculo pesado, sino la entrada y salida: atender muchas conexiones, leer y escribir datos, mover información de un lado a otro. La gran mayoría de las aplicaciones web son exactamente eso: mover datos, no calcular integrales. Por eso Node.js brilla en la web.

Un resumen en una frase para llevar: el modelo tradicional escala con más hilos y más memoria (escala "a lo bruto"), mientras que Node.js escala con inteligencia, haciendo más con menos. No es que uno sea bueno y el otro malo: son herramientas distintas para problemas distintos. Pero para construir servidores web modernos que atienden miles de usuarios al mismo tiempo, el modelo de Node.js es simplemente superior, y por eso conquistó la industria.

1.6 Bloqueante vs. no bloqueante: la diferencia que lo cambia todo

Ya usamos las palabras "bloqueante" y "no bloqueante" varias veces. Es momento de definir las con precisión, porque son el vocabulario básico del programador Node.js y las vas a ver en cada documentación, en cada tutorial y en cada entrevista de trabajo.

Una operación bloqueante es la que detiene la ejecución de tu programa hasta que termina. El hilo se queda congelado, esperando, sin poder hacer nada más. Ejemplo de la vida real: estás en la fila de las tortillas y la señora de adelante está contando sus monedas una por una durante cinco minutos. Tú no puedes avanzar, no puedes hacer otra cosa: estás bloqueado esperando a que termine. En código, un ejemplo clásico de operación bloqueante sería leer un archivo de forma síncrona: tu programa pide el archivo al disco duro y se queda parado hasta que el disco responde. Si el disco tarda medio segundo, tu programa pierde medio segundo sin hacer nada. Medio segundo no suena a mucho, pero para un servidor que atiende miles de peticiones por segundo, medio segundo parado es una eternidad.

Una operación no bloqueante es la que inicia la tarea y devuelve el control de inmediato, sin esperar el resultado. El programa sigue ejecutando las siguientes líneas, y cuando la tarea termina, se ejecuta una función callback con el resultado. Ejemplo de la vida real: pides tus tacos al pastor y en lugar de quedarte parado frente al taquero mirándolo, te sientas, platicas con tu compa, revisas tu celular, y cuando gritan tu nombre vas por tus tacos. No estuviste bloqueado: hiciste otras cosas mientras tu pedido se preparaba. En Node.js, casi todas las operaciones de entrada y salida (leer archivos, hacer peticiones de red, consultar bases de datos) tienen su versión no bloqueante, y usarlas es la forma correcta de programar en este entorno.

Veamos la diferencia con un ejemplo mental de lectura de archivo. Versión bloqueante (síncrona): "oye disco, léeme este archivo... (espera, espera, espera)... listo, aquí está el contenido, sigo con mi programa". Versión no bloqueante (asíncrona): "oye disco, léeme este archivo y cuando termines me avisas con esta función; mientras tanto yo sigo atendiendo a mis clientes... (sigue trabajando)... ¡ah, ya terminaste! procesa el contenido". La primera versión desperdicia el tiempo de espera; la segunda lo aprovecha. Multiplica esa diferencia por miles de operaciones por segundo y entenderás por qué Node.js puede hacer tanto con tan poco.

En Node.js vas a encontrar las dos versiones de muchas operaciones, y se distinguen por su nombre: las funciones síncronas (bloqueantes) suelen llevar la palabra "Sync" al final, como `readFileSync` (leer archivo de forma síncrona). Las funciones asíncronas (no bloqueantes) son las que reciben un callback o devuelven una promesa, como `readFile`. La regla práctica: en un servidor que atiende usuarios, usa SIEMPRE las versiones no bloqueantes. Las versiones

síncronas existen para casos especiales (cómo leer un archivo de configuración una sola vez al arrancar el programa, cuando todavía no hay clientes que atender), pero si las usas en medio de una petición, estarás bloqueando el event loop y violando la regla de oro que vimos en la sección anterior.

No te preocupes si todavía no has escrito ni una línea de código no bloqueante: en las próximas unidades vamos a practicarlo hasta que te salga natural, primero con callbacks, luego con promesas y finalmente con `async/await`, que es la forma moderna y más legible de escribir código no bloqueante. Por ahora, lo importante es que el concepto te quede claro en la cabeza: en Node.js, esperar sin hacer nada es un pecado, y el event loop premia a los que saben pedir algo y seguir trabajando mientras llega la respuesta. El mesero que no se sienta nunca es el que atiende a todo el restaurante.

1.7 Para qué sirve Node.js: su territorio natural

Ya sabes qué es Node.js y cómo funciona por dentro. Ahora la pregunta práctica: ¿para qué lo voy a usar en la vida real? Node.js tiene un "territorio natural", un conjunto de tareas donde brilla más que cualquier otra tecnología. Conocer ese territorio te va a ayudar a elegir bien tus proyectos y a presumir con fundamento en las entrevistas de trabajo.

Primero y más importante: servidores web y APIs. Este es el uso estrella. Con Node.js puedes crear un servidor que recibe peticiones HTTP (las que hacen los navegadores y las apps) y responde con datos, normalmente en formato JSON. Esas APIs son la columna vertebral de Internet moderno: cuando tu app del banco muestra tu saldo, hay una API respondiendo; cuando una página muestra los productos de una tienda, hay una API respondiendo. Node.js es excelente para esto porque las APIs son puro movimiento de datos (recibir, consultar la base de datos, responder), que es justo el punto fuerte del modelo no bloqueante. En unidades posteriores del libro vamos a construir APIs completas desde cero, así que esto no es teoría: es tu futuro inmediato.

Segundo: aplicaciones en tiempo real. Piensa en un chat como WhatsApp web, en un documento colaborativo donde ves lo que otros escriben al instante, en el tablero de un videojuego multijugador, en las notificaciones en vivo de una red social. Todas estas aplicaciones necesitan mantener miles de conexiones abiertas al mismo tiempo, esperando a que pase algo para avisarle al usuario. Con el modelo tradicional de hilos, mantener cien mil conexiones abiertas significaría cien mil hilos: imposible. Con Node.js, cien mil conexiones abiertas cuestan casi nada, porque son solo entradas en la lista del event loop, esperando su turno sin consumir un hilo cada una. La tecnología que hace esto posible se llama WebSockets, y Node.js la maneja de maravilla. No por casualidad las apps de chat, los juegos en línea y las herramientas colaborativas aman Node.js.

Tercero: herramientas de línea de comandos (CLI). Esos programitas que usas en la terminal para automatizar tareas: el propio npm es un programa hecho con Node.js. Con Node.js puedes crear tus propias herramientas: un script que renombre cientos de archivos de golpe, uno que revise tu código buscando errores, uno que despliegue tu página al servidor con un solo comando. Como Node.js tiene acceso total al sistema de archivos y al sistema operativo, es perfecto para automatizar la talacha diaria del programador. De hecho, gran parte de las herramientas modernas del desarrollo web (los empaquetadores, los linters, los generadores de proyectos) están hechas con Node.js.

Cuarto: microservicios y arquitectura moderna. En lugar de construir una aplicación gigante monolítica, muchas empresas dividen su sistema en piezas pequeñas e independientes (microservicios) que se comunican entre sí. Node.js es ideal para microservicios porque arranca rapidísimo, consume poca memoria y es muy bueno hablando por la red con otros servicios. Un microservicio de Node.js puede vivir feliz en un contenedor chiquito en la nube, costando centavos al mes.

Quinto: el propio ecosistema de desarrollo. Aunque no lo creas, probablemente ya usas herramientas hechas con Node.js sin saberlo: si alguna vez usaste un empaquetador, un preprocesador de CSS o un generador de sitios estáticos, casi seguro había Node.js por debajo. Aprender Node.js te da las llaves de todo ese taller.

1.8 Para qué NO sirve Node.js: conociendo sus límites

Un buen programador no solo sabe para qué sirve su herramienta, sino también para qué no sirve. Usar Node.js para lo que no fue diseñado es cómo usar un cuchillo de cocina como desarmador: se puede, pero va a salir mal. Así que hablemos con honestidad de sus límites.

Node.js NO es ideal para tareas pesadas de cálculo en el procesador (lo que se llama CPU-bound). Si tu programa necesita hacer millones de operaciones matemáticas seguidas (por ejemplo: codificar video, comprimir archivos gigantes, hacer cálculos científicos, entrenar modelos de inteligencia artificial, minar criptomonedas), el modelo de un solo hilo se vuelve una desventaja: mientras tu código está calculando, el event loop está bloqueado y nadie más es atendido. Es como si nuestro mesero estrella de pronto tuviera que ponerse a lavar todos los trastes del restaurante a mano: mientras lava, las mesas se quedan sin atención. Para esas tareas existen mejores herramientas (otros lenguajes, o los worker threads de Node.js para casos específicos, que veremos mucho más adelante).

Tampoco es la mejor opción cuando necesitas aprovechar al máximo una máquina con muchísimos núcleos del procesador para cálculo paralelo, porque un proceso de Node.js usa principalmente un núcleo para tu código JavaScript. (En la práctica esto se resuelve corriendo

varias instancias del programa, una por núcleo, pero es trabajo extra que otros entornos hacen automáticamente.)

Y una advertencia de principiante: Node.js no es una base de datos, no es un servidor de archivos por sí solo y no hace magia. Es una herramienta para construir esas cosas, no esas cosas ya construidas. Cuando alguien dice "voy a usar Node.js para mi proyecto", lo que realmente quiere decir es "voy a construir mi servidor con Node.js, probablemente usando algún framework como Express y alguna base de datos". Node.js es el motor y el chasis; tú construyes el coche completo en las siguientes unidades.

La regla de dedo es sencilla: si tu aplicación se trata de mover datos, atender usuarios y responder rápido (casi todas las aplicaciones web), Node.js es una excelente elección. Si tu aplicación se trata de calcular cosas pesadísimas durante mucho tiempo, busca otra herramienta o complementa Node.js con algo más. Conocer esta frontera te hace un mejor ingeniero que saberte de memoria cien funciones.

1.9 Instalación paso a paso: Node.js y npm en tu computadora (Clase 2)

Llegó la hora de ensuciarse las manos. En esta sección vamos a instalar Node.js en tu computadora paso a paso, sin dar nada por sentado. Da igual si usas Windows, Mac o Linux: hay instrucciones para los tres. Tómate tu tiempo, sigue los pasos en orden y no avances hasta que cada paso funcione. Una instalación bien hecha hoy te ahorra semanas de dolores de cabeza después.

Antes de instalar, una decisión importante: ¿qué versión descargo, LTS o Current? En la página oficial de Node.js (nodejs.org) siempre verás dos botones grandes. LTS significa Long Term Support (soporte a largo plazo): es la versión estable, probada y recomendada para la mayoría de los usuarios, incluyendo producción y aprendizaje. Current es la versión más nueva, con las últimas características, pero menos probada y con soporte más corto. La regla es simple y no tiene excepciones para ti ahorita: descarga la versión LTS. Es la que usan las empresas, la que tienen los tutoriales y la que te va a dar menos sorpresas. Cuando seas experto y necesites una característica nuevecésima, ya probarás la Current. Por ahora, LTS siempre.

Instalación en Windows. Ve a nodejs.org, haz clic en el botón LTS y descarga el instalador (.msi). Ejecuta el instalador y sigue el asistente: acepta la licencia, deja la carpeta de instalación por defecto (no la cambies a menos que sepas lo que haces) y, muy importante, asegúrate de que la casilla que dice "Add to PATH" (agregar al PATH) esté marcada. Esa casilla es la que permite que los comandos node y npm funcionen desde cualquier terminal; si la desmarcas por accidente, tendrás que reinstalar o arreglar el PATH a mano. El instalador

también te puede ofrecer instalar herramientas adicionales de compilación (algo sobre "Chocolatey" o herramientas de Python): para este curso no las necesitas, puedes saltar ese paso. Cuando termine, CIERRA todas las terminales que tengas abiertas y abre una nueva: esto es crucial, porque las terminales viejas no se enteran de los cambios en el PATH. Para abrir una terminal en Windows, presiona la tecla Windows, escribe "cmd" o "PowerShell" y ábrelo.

Instalación en Mac. Tienes dos caminos. El camino fácil: ve a nodejs.org, descarga el instalador LTS (.pkg), ejecútalo y sigue los pasos; es igual de sencillo que instalar cualquier programa de Mac. El camino de programador: usa Homebrew, el manejador de paquetes de Mac. Si ya tienes Homebrew instalado, abre la terminal y ejecuta `brew install node`. Si no tienes Homebrew, mejor usa el instalador de la página: instalar Homebrew es otro tema y no queremos desviarnos. En Mac la terminal se abre desde Aplicaciones, luego Utilidades, luego Terminal. Igual que en Windows: después de instalar, cierra y reabre la terminal.

Instalación en Linux. Aquí depende de tu distribución, pero el método más confiable en Ubuntu, Debian y derivadas es usar el repositorio oficial de NodeSource, porque los repositorios del sistema suelen traer versiones viejísimas de Node.js que no te sirven. Los pasos típicos son: abrir la terminal, descargar el script de configuración de NodeSource para la versión LTS, ejecutarlo y luego instalar con el manejador de paquetes. Otra opción muy recomendada en Linux (y en Mac) es `nvm`, el manejador de versiones de Node, del que hablaremos en la sección 1.11: con `nvm` instalas Node.js sin permisos de administrador y puedes cambiar de versión cuando quieras. Si usas Linux, casi seguro ya te sientes cómodo con la terminal, así que elige el método que prefieras, pero verifica al final que la versión instalada sea LTS y reciente.

Verificación: el momento de la verdad. Con la terminal recién abierta (recuerda: recién abierta, no una vieja), escribe estos dos comandos, cada uno seguido de Enter:

```
node -v  
npm -v
```

El primero te debe mostrar algo como `v20.11.0` (el número exacto depende de la versión LTS actual cuando leas esto; lo importante es que empiece con "v" seguida de números). El segundo te muestra la versión de npm, algo como `10.2.4`. Si ambos comandos responden con números de versión, felicidades: tienes Node.js y npm instalados y funcionando. Haz una captura de pantalla o anota esas versiones, porque en el futuro, cuando algo falle, la primera pregunta que te harán (o que te harás tú mismo) será "¿qué versión de Node tienes?".

Si en lugar de versiones ves un error como "node: command not found" o "no se reconoce como comando", no entres en pánico: es el error más común del mundo y tiene solución. Ve directo a la sección 1.14 de errores comunes, donde lo resolvemos paso a paso. No eres el primero ni el último: casi todos los programadores se topan con el problema del PATH en su primera instalación.

1.10 npm: la tienda de refacciones más grande del mundo

Cuando instalaste Node.js, también se instaló npm, y vale la pena entender qué es, porque va a ser tu compañero inseparable.

npm significa Node Package Manager (manejador de paquetes de Node). Un "paquete" es un pedazo de código reutilizable que alguien más escribió y publicó para que todo el mundo lo use: puede ser una biblioteca para conectarse a una base de datos, una herramienta para validar correos, un framework para construir servidores, o casi cualquier cosa que te imagines. npm es el programa que te permite buscar, descargar e instalar esos paquetes con un solo comando, y además se encarga de sus dependencias (si el paquete que instalas necesita otros paquetes, npm los instala también, automáticamente).

Para dimensionar su tamaño: npm es el registro de código más grande que existe en el planeta, con millones de paquetes publicados y miles de millones de descargas por semana. Hay paquetes para absolutamente todo: para fechas, para contraseñas, para subir archivos, para mandar correos, para graficar datos, para casi cualquier problema común que se te ocurra. Esto cambia por completo la forma de programar: en lugar de reinventar la rueda cada vez, buscas si alguien ya resolvió tu problema, instalas su paquete y te concentras en lo que hace único a tu proyecto. Ojo, esto no significa instalar paquetes a lo loco para todo (cada dependencia es código ajeno que debes mantener y vigilar), pero sí significa que no estás solo: hay un ejército de programadores compartiendo soluciones.

En este libro vamos a usar npm desde las primeras unidades: primero para instalar herramientas, luego para crear nuestros propios proyectos con su archivo package.json (el "acta de nacimiento" de cada proyecto Node.js, donde se anotan sus dependencias y sus datos). Por ahora, quédate con la idea general: Node.js es el motor, npm es la refaccionaria gigante donde consigues las piezas. Y ya lo tienes instalado: el comando `npm -v` que ejecutaste hace un momento lo confirma.

1.11 nvm: el control remoto de tus versiones de Node.js

Conforme avances, te vas a topar con una realidad del mundo profesional: distintos proyectos necesitan distintas versiones de Node.js. Un proyecto viejo de tu trabajo puede necesitar Node 16, mientras que tu proyecto nuevo usa Node 20. Desinstalar y reinstalar Node.js cada vez que cambias de proyecto sería una tortura. Para eso existe nvm, el Node Version Manager.

nvm es una herramienta que te permite tener varias versiones de Node.js instaladas al mismo tiempo y cambiar entre ellas con un solo comando. Por ejemplo: `nvm install 20` instala la versión 20, `nvm use 16` cambia a la versión 16, `nvm ls` lista las versiones que tienes. Así de simple. Además, nvm instala Node.js en tu carpeta de usuario, sin necesidad de permisos de administrador, lo cual resuelve de raíz los famosos problemas de permisos con `sudo` que veremos en la sección de errores.

¿Deberías instalar nvm ahorita? Si usas Mac o Linux, sí, te lo recomiendo mucho: es la forma más limpia y profesional de manejar Node.js, y te va a ahorrar problemas de permisos desde el día uno. Si usas Windows, la historia es un poco distinta: el nvm original no funciona en Windows, pero existe una versión aparte llamada nvm-windows que hace lo mismo; si te sientes cómodo, instálala, y si no, el instalador oficial que ya usaste está bien para empezar. No es obligatorio para este curso: con la instalación normal es suficiente. Pero cuando empieces a trabajar en varios proyectos reales, nvm (o su primo para Windows) se va a volver indispensable. Considéralo una inversión a futuro que puedes hacer hoy mismo si te animas.

1.12 El REPL: tu laboratorio de pruebas instantáneo

Antes de escribir nuestro primer archivo, vamos a conocer una herramienta que vas a usar todos los días: el REPL de Node.js.

REPL son las siglas de Read, Eval, Print, Loop (leer, evaluar, imprimir, repetir). Es un modo interactivo de Node.js: escribes una línea de JavaScript, Node.js la ejecuta al instante y te muestra el resultado, y repite el ciclo. Es como tener una conversación con Node.js: tú preguntas, él responde, tú repreguntas. Para entrar al REPL, abre tu terminal y escribe simplemente:

```
node
```

Sin argumentos, sin nada más. Verás que el prompt cambia a algo como `>` (el símbolo de "mayor que"), que es la señal de que estás dentro del REPL y Node.js está esperando tus instrucciones. Prueba escribir:

```
2 + 2
```

Y presiona Enter. Node.js responde de inmediato:

```
4
```

Prueba con algo más interesante:

```
console.log("Hola desde el REPL")
```

Verás el mensaje impreso. Prueba declarar una variable y usarla:

```
let nombre = "Esteban"  
"Hola, " + nombre
```

El REPL te responde "Hola, Esteban". ¿Ves la magia? No creaste ningún archivo, no abriste ningún navegador, no hiciste nada especial: escribiste JavaScript y se ejecutó al instante. El REPL es perfecto para probar ideas rápidas ("¿cómo funciona este método de arreglos?"), para hacer cálculos, para experimentar con funciones antes de meterlas a tu programa, y para aprender: es el cuaderno de borrador del programador Node.js.

Algunos trucos del REPL que te van a servir: si escribes `.help` y presionas Enter, verás la lista de comandos especiales; `.exit` (o presionar Ctrl + C dos veces, o Ctrl + D) te saca del REPL y te regresa a la terminal normal. Si presionas la flecha hacia arriba, recuperas los comandos anteriores (como en la terminal). Y si defines una variable con `let` o `const`, esa variable sigue viva mientras no salgas del REPL, así que puedes construir experimentos de varias líneas.

Una advertencia importante: el REPL es para probar, no para trabajar en serio. Todo lo que escribas ahí se pierde cuando sales (a menos que lo guardes). Los programas de verdad viven en archivos `.js`, que es justo lo que vamos a hacer en la siguiente sección: nuestro primer script. Piensa en el REPL como el laboratorio y en los archivos como la fábrica: experimentas en el laboratorio, produces en la fábrica.

1.13 Tu primer script: `hola.js` y el comando `node` (Clase 3)

Todo lo que hemos visto hasta ahora fue preparación. Este es el momento estelar de la unidad: vas a escribir tu primer programa de Node.js, guardarlo en un archivo y ejecutarlo desde la terminal, sin navegador, sin HTML, sin nada más que tú, tu código y Node.js.

Paso 1: crea una carpeta para tus prácticas. Abre tu terminal y ejecuta:

```
mkdir curso-node  
cd curso-node
```

El primer comando crea la carpeta, el segundo entra en ella. A partir de ahora, todos los scripts de este libro van a vivir ahí (o en subcarpetas que iremos creando). Tener un lugar ordenado para tu código es un hábito de profesional: nada de archivos regados por el escritorio.

Paso 2: crea el archivo `hola.js`. Abre tu editor de texto favorito (VS Code, Sublime Text, o el que uses; si no tienes ninguno, descarga VS Code, que es gratis y es el estándar de la industria) y escribe exactamente esto:

```
console.log("Hola, mundo desde Node.js");
```

Guarda el archivo con el nombre `hola.js` dentro de la carpeta `curso-node` que acabas de crear. Fíjate en la extensión `.js`: es la que le dice a Node.js (y a ti) que este archivo contiene código JavaScript.

Paso 3: ejecútalo. Regresa a tu terminal (asegúrate de estar dentro de la carpeta `curso-node`; si no, usa `cd` para entrar) y escribe:

```
node hola.js
```

Presiona Enter y verás aparecer:

```
Hola, mundo desde Node.js
```

Felicidades. Acabas de ejecutar JavaScript fuera del navegador por primera vez. Detente un segundo a apreciar lo que pasó: no abriste Chrome, no creaste un archivo HTML, no subiste nada a ningún servidor. Node.js tomó tu archivo, lo leyó, encontró la instrucción `console.log`, la ejecutó usando el motor V8 y mostró el resultado en tu terminal usando las capacidades de entrada y salida del sistema. Todo eso en una fracción de segundo, con un comando de dos palabras.

Vamos a desmenuzar el comando `node hola.js`, porque lo vas a escribir miles de veces en tu vida: la palabra `node` invoca al programa Node.js (el entorno de ejecución que instalaste), y `hola.js` es el argumento que le dice qué archivo debe ejecutar. Es como decirle: "oye Node, corre este archivo". Si el archivo está en otra carpeta, le pasas la ruta: `node carpeta/otro.js`. Si el archivo tiene un error de sintaxis, Node.js te lo dirá con un mensaje de error y el número de línea: lee esos mensajes con calma, porque son tu mejor pista para arreglar el problema.

Y desmenucemos también la única línea de código: `console.log("Hola, mundo desde Node.js")`. `console` es un objeto global que existe tanto en el navegador como en Node.js; `log` es su método para imprimir mensajes. La diferencia está en DÓNDE se imprime: en el navegador, `console.log` muestra el mensaje en la consola de las herramientas de desarrollo; en Node.js, lo muestra en la terminal donde ejecutaste el comando. El texto entre comillas es una cadena (string), el tipo de dato más básico para representar texto. Los paréntesis y el punto y coma son sintaxis de JavaScript que ya conoces. Nada nuevo bajo el sol: el mismo `console.log` de siempre, pero ahora con superpoderes de servidor.

Un experimento para que juegues ahorita mismo: modifica `hola.js` para que tenga tres líneas:

```
console.log("Hola, mundo desde Node.js");  
console.log("Este es mi primer script");  
console.log("2 + 3 =", 2 + 3);
```

Guárdalo, vuelve a ejecutar `node hola.js` y observa la salida. Fíjate en la tercera línea: `console.log` puede recibir varios argumentos separados por comas y los imprime todos, y además puede imprimir el resultado de una operación (`2 + 3` se evalúa a `5`). Cada vez que cambies el archivo, tienes que volver a ejecutar `node hola.js` para ver los cambios: Node.js lee el archivo en el momento de la ejecución, no está "observando" tus cambios automáticamente (eso de observar cambios lo hacen herramientas externas que veremos después). Este ciclo de editar-guardar-ejecutar va a ser tu ritmo de trabajo de ahora en adelante: acostúmbrate, porque lo vas a repetir miles de veces.

1.14 Errores comunes de instalación y cómo resolverlos

Si tu instalación funcionó a la primera, qué chido, sigue adelante. Pero si algo falló, esta sección es para ti, y quiero que sepas algo: los errores de instalación le pasan a todo el mundo, incluyendo a programadores con años de experiencia que cambian de computadora. No significa que seas malo para esto; significa que estás instalando software de verdad en una computadora de verdad, y eso a veces tiene fricción. Vamos con los errores más comunes y sus soluciones.

Error 1: "node no se reconoce como comando" (Windows) o "command not found" (Mac/Linux). Este es el rey de los errores de instalación, y en el 90% de los casos la causa es la misma: la terminal que estás usando se abrió ANTES de instalar Node.js, así que no se enteró de los cambios en el PATH. El PATH es una variable del sistema que le dice a la terminal en qué carpetas debe buscar los programas cuando escribes un comando; el instalador de Node.js agrega ahí la carpeta de Node, pero las terminales ya abiertas no recargan esa lista. Solución: cierra TODAS las terminales abiertas y abre una completamente nueva. Luego vuelve a probar

node -v. Si sigue fallando, reinicia tu computadora (sí, en serio: a veces Windows necesita un reinicio para aplicar cambios del PATH) y prueba de nuevo. Solo si después del reinicio sigue sin funcionar, reinstala asegurándote de que la casilla "Add to PATH" esté marcada durante la instalación.

Error 2: errores de permisos con npm en Mac/Linux (EACCES). Se ve algo así: intentas instalar un paquete global con npm y recibes un error que menciona EACCES o "permission denied". La causa: npm está intentando escribir en una carpeta del sistema donde tu usuario normal no tiene permiso. La "solución" que vas a encontrar en muchos foros viejos es usar sudo (npm install -g algo con sudo), pero NO lo hagas: usar sudo con npm es una mala práctica que puede dejar archivos con permisos revueltos y causarte problemas peores después. Las soluciones correctas son dos: la mejor es usar nvm (sección 1.11), que instala todo en tu carpeta de usuario y elimina el problema de raíz; la alternativa es configurar npm para que use una carpeta propia en tu directorio personal para los paquetes globales (la documentación oficial de npm explica cómo hacerlo en pocos pasos). Si ya usaste sudo alguna vez y ahora tienes errores raros, la salida más limpia suele ser desinstalar Node.js por completo e instalarlo de nuevo con nvm.

Error 3: tienes una versión viejísima de Node.js. Pasa mucho en Linux, donde el comando de instalación del sistema (apt install nodejs) instala versiones antiquísimas que tienen años de obsoletas, o en computadoras donde alguien instaló Node.js hace mucho y nunca lo actualizó. ¿Cómo te das cuenta? Ejecutas node -v y ves algo como v10.x o v12.x (o peor). Muchas de las cosas de este libro no van a funcionar con versiones tan viejas. Solución: desinstala esa versión vieja e instala la LTS actual, de preferencia con nvm, que además te permite conservar la vieja por si algún proyecto antiguo la necesita. Para saber si tu versión es aceptable para este libro: cualquier LTS desde la 18 en adelante está bien; si ves un número menor, actualiza.

Error 4: el antivirus o el firewall bloquean la instalación (Windows). Algunos antivirus marcan al instalador de Node.js como sospechoso, o el firewall de Windows bloquea a Node.js cuando intenta usar la red (esto te va a pasar cuando creemos nuestro primer servidor en la unidad 2: Windows te preguntará si quieres permitir que Node.js se comuniquen por la red). Solución: si el antivirus bloquea el instalador, descárgalo de nuevo desde nodejs.org (el sitio oficial, nunca de otro lado) y agrégalo a las excepciones de tu antivirus; cuando el firewall pregunte por Node.js, acepta y marca las redes privadas. Node.js es software legítimo usado por millones de programadores: no es un virus, aunque a veces los antivirus exagerados no lo sepan.

Error 5: "el puerto ya está en uso" (lo verás en la unidad 2, pero te lo adelanto). Cuando crees tu primer servidor, Node.js te puede decir que el puerto (por ejemplo el 3000) ya está ocupado por otro programa. Solución simple: usa otro puerto (3001, 4000, 8080, el que quieras) o cierra el programa que lo está ocupando. No es un error de instalación, pero como sé que te va a asustar la primera vez, ya quedas avisado: no pasa nada, solo cambia el número del puerto.

Regla general para resolver errores: lee el mensaje completo, no solo la primera línea. Los mensajes de error de Node.js y npm suelen decir exactamente qué pasó y a veces hasta sugieren la solución, pero están en inglés y con formato técnico, así que dan miedo al principio. Respira, léelo despacio, copia el mensaje clave en un buscador y verás que cientos de personas tuvieron el mismo problema antes que tú. Buscar errores en Internet no es hacer trampa: es la habilidad más importante del programador profesional. Nadie se sabe todos los errores de memoria; los buenos programadores son los que saben diagnosticarlos.

1.15 Buenas prácticas desde el día uno

Para cerrar la explicación detallada, una lista de buenas prácticas que te conviene adoptar desde ahorita. No son reglas caprichosas: cada una existe porque alguien (probablemente miles de alguien) se quemó antes que tú.

Primera: usa siempre la versión LTS para aprender y para trabajar, a menos que tengas una razón específica para hacer lo contrario. La LTS es la versión que tiene soporte largo, la que prueban las empresas y la que usan los tutoriales. La versión Current es para curiosos y para probar novedades; no construyas tus proyectos importantes sobre ella.

Segunda: no uses sudo con npm. Ya lo vimos en los errores comunes, pero lo repito porque es el error más cometido por los principiantes en Mac y Linux. Si npm te pide permisos de administrador, el problema es tu instalación, no npm: arréglalo con nvm o con la configuración de carpetas, no con sudo.

Tercera: mantén tu Node.js actualizado dentro de la línea LTS. Las versiones LTS reciben actualizaciones de seguridad y correcciones durante años; cuando salga una nueva LTS, planea migrar tus proyectos. Un Node.js desactualizado es un riesgo de seguridad, sobre todo en servidores.

Cuarta: lee los mensajes de error completos y aprende a buscarlos. El programador que lee sus errores avanza diez veces más rápido que el que los ignora y prueba cosas al azar. Acostúmbrate a que el error es información, no un regaño.

Quinta: organiza tu código en carpetas desde el principio. Un archivo suelto está bien para hola.js, pero en cuanto tengas tres o cuatro scripts, créales su carpeta por proyecto. Tu yo del futuro te lo va a agradecer cuando busque ese script dentro de seis meses.

Sexta: usa el REPL para experimentar y los archivos para construir. ¿Duda rápida de sintaxis? Al REPL. ¿Programa que vas a usar más de una vez? A un archivo .js. No construyas nada importante directamente en el REPL porque se pierde al salir.

Séptima: escribe código que puedas leer dentro de seis meses. Pon nombres claros a tus variables y archivos, deja comentarios en las partes tramposas y no trates de verte inteligente con trucos crípticos. El código se escribe una vez y se lee cien veces: optimiza para la lectura.

Con esto terminamos la explicación detallada. Ya sabes qué es Node.js, por qué nació, cómo funciona por dentro con su event loop, para qué sirve y para qué no, cómo instalarlo, qué es npm, qué es nvm, qué es el REPL y cómo correr tu primer script. Es muchísima base para una sola unidad, y es a propósito: todo lo que sigue en el libro se apoya en estos conceptos. Ahora vamos a la sección de ejemplos, donde convertiremos la teoría en código que puedes tocar.

Ejemplos paso a paso

Ejemplo 1: hola.js, tu primer script

Este es el ejemplo fundamental de la unidad. Si solo haces un ejemplo de todo el libro, que sea este.

Paso 1: abre tu terminal y crea la carpeta de trabajo (si no la creaste antes):

```
mkdir curso-node
cd curso-node
```

Paso 2: crea un archivo llamado hola.js con el siguiente contenido:

```
console.log("Hola, mundo desde Node.js");
```

Paso 3: ejecútalo desde la terminal, estando dentro de la carpeta:

```
node hola.js
```

Paso 4: verifica la salida. Debes ver exactamente esto:

```
Hola, mundo desde Node.js
```

¿Qué aprendiste? Que node seguido del nombre de un archivo ejecuta ese archivo con Node.js, que console.log imprime en la terminal, y que no necesitas navegador ni HTML para correr JavaScript. Todo programa de Node.js que escribas en este libro se ejecuta así: node nombre-del-archivo.js.

Ejemplo 2: script que suma dos números usando process.argv

Ahora vamos a hacer algo más interesante: un script que recibe datos desde la terminal. Para eso usaremos process.argv, un arreglo que contiene los argumentos con los que se ejecutó el programa.

Crea un archivo llamado suma.js con este contenido:

```
// suma.js - Suma dos números recibidos desde la terminal
let numero1 = Number(process.argv[2]);
let numero2 = Number(process.argv[3]);
let resultado = numero1 + numero2;
console.log("La suma es:", resultado);
```

Ejecútalo así:

```
node suma.js 7 5
```

La salida será:

```
La suma es: 12
```

Vamos por partes, porque aquí hay conceptos nuevos. process es un objeto global de Node.js con información del proceso actual; argv es su propiedad que guarda los argumentos de la línea de comandos como un arreglo. Pero ojo: process.argv[0] es la ruta del programa node, y process.argv[1] es la ruta de tu script; TUS argumentos empiezan en process.argv[2]. Por eso el primer número está en [2] y el segundo en [3]. Es el error más común con argv: usar [0] y [1] y preguntarse por qué sale algo raro.

Los valores de argv siempre llegan como texto (strings), aunque escribas números. Por eso usamos Number() para convertirlos: sin esa conversión, numero1 + numero2 no sumaría, sino que pegaría los textos (concatenación), y "7" + "5" daría "75" en lugar de 12. Prueba quitar los Number() y ejecutar de nuevo para ver el desastre con tus propios ojos: es una lección que se aprende mejor rompiendo cosas.

Prueba con otros números: node suma.js 100 250, node suma.js 3 4. El script es el mismo; los datos cambian desde la terminal. Acabas de crear un programa que recibe entrada del usuario sin necesidad de páginas web ni formularios: la terminal es tu interfaz.

Ejemplo 3: script que muestra la hora del sistema

Node.js tiene acceso al sistema, y una de las cosas más sencillas que puede hacer es leer la fecha y hora de la computadora usando el objeto Date de JavaScript (que ya conoces del navegador, porque Date sí existe en ambos mundos).

Crea un archivo llamado hora.js:

```
// hora.js - Muestra la fecha y hora actual del sistema
let ahora = new Date();
console.log("Fecha y hora actual:", ahora.toLocaleString());
console.log("Hora:", ahora.getHours() + ":" + ahora.getMinutes());
console.log("Año:", ahora.getFullYear());
```

Ejecútalo:

```
node hora.js
```

Verás algo como:

```
Fecha y hora actual: 19/9/2026, 09:53:15
Hora: 9:53
Año: 2026
```

(Los números exactos dependerán del momento en que lo ejecutes, claro.) Lo interesante de este ejemplo no es solo la hora: es darte cuenta de que tu script está conversando con el sistema operativo, leyendo su reloj interno. En el navegador, Date te da la hora del usuario que visita la página; en Node.js, te da la hora del servidor donde corre tu programa. Esa diferencia va a importar mucho cuando construyamos servidores: la hora del servidor es la hora oficial de tu aplicación.

Ejemplo 4: recorrido guiado por el REPL

Vamos a hacer un recorrido paso a paso por el REPL, para que lo domines desde ahorita. Abre tu terminal y escribe node (sin argumentos). Verás el prompt >.

Paso 1: escribe 2 + 2 y presiona Enter. Respuesta: 4. El REPL evalúa expresiones y muestra el resultado.

Paso 2: escribe let edad = 25 y presiona Enter. Respuesta: undefined. No te asustes: declarar una variable no produce un valor que mostrar, así que el REPL muestra undefined. Es normal.

Paso 3: escribe edad + 5 y presiona Enter. Respuesta: 30. La variable sigue viva en la sesión del REPL.

Paso 4: escribe console.log("probando el REPL") y presiona Enter. Verás el mensaje impreso, y debajo la palabra undefined (porque console.log no devuelve nada).

Paso 5: escribe .help y presiona Enter para ver los comandos especiales del REPL.

Paso 6: sal del REPL escribiendo .exit y presionando Enter (o Ctrl + C dos veces). Volverás a tu terminal normal.

Todo lo que hiciste se perdió al salir, y está bien: el REPL era solo el laboratorio. Si alguno de esos experimentos te gustó, pásalo a un archivo .js para conservarlo.

Ejemplo 5: script que te saluda por tu nombre

Juntemos lo aprendido: argumentos de la terminal (ejemplo 2) y mensajes personalizados (ejemplo 1). Crea saludo.js:

```
// saludo.js - Saluda a la persona cuyo nombre se pasa como argumento
let nombre = process.argv[2];
console.log("Hola, " + nombre + ". Bienvenido al curso de Node.js");
console.log("Hoy es: " + new Date().toLocaleDateString());
```

Ejecútalo:

```
node saludo.js Esteban
```

Salida:

```
Hola, Esteban. Bienvenido al curso de Node.js
Hoy es: 19/9/2026
```

Cambia el nombre y ejecuta de nuevo: `node saludo.js María`, `node saludo.js "Juan Pérez"` (con comillas si el nombre lleva espacios, para que la terminal lo trate como un solo argumento). Este ejemplo, aunque sencillo, contiene el germen de todo programa interactivo: recibir entrada, procesarla y devolver una respuesta personalizada. Los servidores que construiremos hacen exactamente lo mismo, solo que la entrada llega por la red en lugar de la terminal.

Práctica

Es hora de trabajar. Los siguientes ejercicios están ordenados de menor a mayor dificultad y cubren todo lo visto en la unidad. Intenta resolver cada uno POR TU CUENTA antes de leer la solución: escribe el código, ejecútalo, equivócate y arréglalo. La solución está para verificar y para aprender cuando te atores, no para copiarla a la primera. Si un ejercicio te cuesta, relea la sección correspondiente de la explicación antes de rendirte.

Ejercicio 1: verifica tu instalación

Enunciado: abre tu terminal y ejecuta los comandos necesarios para mostrar la versión de Node.js y la versión de npm que tienes instaladas. Anota ambas versiones. Después entra al REPL, haz la operación $15 * 4$, declara una variable llamada `curso` con el valor "Node.js" y concaténala en un mensaje. Sal del REPL.

Solución paso a paso:

Paso 1: abre una terminal nueva (recién abierta, para que tome los cambios del PATH si acabas de instalar).

Paso 2: ejecuta `node -v` y anota el resultado, por ejemplo `v20.11.0`. Esa es tu versión de Node.js.

Paso 3: ejecuta `npm -v` y anota el resultado, por ejemplo `10.2.4`. Esa es tu versión de npm.

Paso 4: escribe `node` sin argumentos para entrar al REPL. Verás el prompt `>`.

Paso 5: escribe $15 * 4$ y presiona Enter. El REPL responde `60`.

Paso 6: escribe `let curso = "Node.js"` y presiona Enter. El REPL responde `undefined` (normal, es una declaración).

Paso 7: escribe `"Estoy aprendiendo " + curso` y presiona Enter. El REPL responde `"Estoy aprendiendo Node.js"`.

Paso 8: escribe `.exit` para salir del REPL.

¿Qué practicaste? Los dos comandos de verificación que usarás toda tu vida (node -v y npm -v), la entrada y salida del REPL, y operaciones básicas dentro de él. Si alguno de los comandos de versión falló, regresa a la sección 1.14 y resuelve el problema del PATH antes de continuar: no avances con una instalación rota.

Ejercicio 2: tu ficha de presentación

Enunciado: crea un archivo llamado `ficha.js` que imprima en la terminal tu nombre, tu edad, tu ciudad y el nombre de este curso, cada dato en su propia línea. Usa variables para guardar cada dato y `console.log` para imprimirlos. Ejecútalo con node.

Solución paso a paso:

Paso 1: crea el archivo `ficha.js` en tu carpeta `curso-node` con este contenido:

```
// ficha.js - Imprime una ficha de presentación usando variables
let nombre = "Tu Nombre";
let edad = 28;
let ciudad = "Ciudad de México";
let curso = "Curso Completo de Node.js";

console.log("Nombre: " + nombre);
console.log("Edad: " + edad);
console.log("Ciudad: " + ciudad);
console.log("Curso: " + curso);
```

(Cambia los valores por tus datos reales, claro.)

Paso 2: guárdalo y ejecútalo desde la terminal:

```
node ficha.js
```

Paso 3: verifica la salida, que debe verse así (con tus datos):

```
Nombre: Tu Nombre
Edad: 28
Ciudad: Ciudad de México
Curso: Curso Completo de Node.js
```

¿Qué practicaste? Declaración de variables con `let`, concatenación de strings con el operador `+`, impresión de varias líneas y el ciclo completo de editar-guardar-ejecutar. Parece trivial, pero este ciclo es el latido de tu trabajo diario como programador backend. Nota cómo mezclamos

texto y números en la concatenación sin problema: JavaScript convierte el número a texto automáticamente al concatenar.

Ejercicio 3: calculadora de terminal

Enunciado: crea un archivo llamado `calculadora.js` que reciba tres argumentos desde la terminal: un primer número, un segundo número y una operación (suma, resta, multi o divi). El script debe hacer la operación indicada e imprimir el resultado. Debe funcionar así: `node calculadora.js 10 4 resta` imprime 6.

Solución paso a paso:

Paso 1: crea el archivo `calculadora.js` con este contenido:

```
// calculadora.js - Calculadora que recibe números y operación desde la terminal
let a = Number(process.argv[2]);
let b = Number(process.argv[3]);
let operacion = process.argv[4];
let resultado;

if (operacion === "suma") {
  resultado = a + b;
} else if (operacion === "resta") {
  resultado = a - b;
} else if (operacion === "multi") {
  resultado = a * b;
} else if (operacion === "divi") {
  resultado = a / b;
} else {
  resultado = "Operación no válida. Usa: suma, resta, multi o divi";
}

console.log("Resultado:", resultado);
```

Paso 2: pruébalo con las cuatro operaciones:

```
node calculadora.js 10 4 suma
node calculadora.js 10 4 resta
node calculadora.js 10 4 multi
node calculadora.js 10 4 divi
node calculadora.js 10 4 otra
```

Paso 3: verifica las salidas: 14, 6, 40, 2.5 y el mensaje de error para la operación inválida.

¿Qué practicaste? Lectura de múltiples argumentos con `process.argv` (recuerda: tus datos empiezan en [2]), conversión de texto a número con `Number()`, condicionales `if/else if/else`, y manejo básico de entradas inválidas. Fíjate en el detalle del último caso: un buen programa no se cae con entradas raras, sino que avisa qué salió mal. Ese hábito de validar entradas te va a salvar en el backend, donde los usuarios (y los atacantes) mandan de todo.

Ejercicio 4: saludo según la hora del día

Enunciado: crea un archivo llamado `buenos.js` que lea la hora actual del sistema y muestre "Buenos días", "Buenas tardes" o "Buenas noches" según corresponda (días: 6 a 11, tardes: 12 a 19, noches: 20 a 5). Además debe mostrar la hora exacta. Pruébalo.

Solución paso a paso:

Paso 1: crea el archivo `buenos.js` con este contenido:

```
// buenos.js - Saluda según la hora del día
let ahora = new Date();
let hora = ahora.getHours();
let saludo;

if (hora >= 6 && hora < 12) {
  saludo = "Buenos días";
} else if (hora >= 12 && hora < 20) {
  saludo = "Buenas tardes";
} else {
  saludo = "Buenas noches";
}

console.log(saludo + ". Son las " + hora + ":" + ahora.getMinutes() + ".");
```

Paso 2: ejecútalo:

```
node buenos.js
```

Paso 3: verifica que el saludo corresponda a la hora en que lo ejecutaste. Si quieres probar los tres casos sin esperar, cambia temporalmente los rangos del `if` o asigna un valor fijo a `hora` (por ejemplo `let hora = 15;`) y observa cómo cambia el saludo. Luego regresa el código a su forma original.

¿Qué practicaste? Uso del objeto `Date` para leer la hora del sistema (`getHours`, `getMinutes`), condicionales con rangos usando el operador `&&` (Y lógico), y construcción de mensajes

dinámicos. Este ejercicio combina dos ideas de la unidad: Node.js conversando con el sistema operativo (la hora) y tu lógica decidiendo qué hacer con ese dato. Casi todos los programas backend son eso: leer datos del mundo y decidir.

Ejercicio 5 (reto): contador con validación

Enunciado: crea un archivo llamado `contador.js` que reciba un número `N` desde la terminal e imprima los números del 1 al `N`, uno por línea. Si no se recibe ningún número, si no es un número válido o si es menor que 1, debe mostrar un mensaje de error explicando el uso correcto. Ejemplo: `node contador.js 5` imprime 1, 2, 3, 4, 5.

Solución paso a paso:

Paso 1: crea el archivo `contador.js` con este contenido:

```
// contador.js - Imprime los números del 1 al N recibido como argumento
let n = Number(process.argv[2]);

if (isNaN(n) || n < 1) {
  console.log("Uso: node contador.js <numero mayor que 0>");
} else {
  for (let i = 1; i <= n; i++) {
    console.log(i);
  }
}
```

Paso 2: pruébalo con un caso válido:

```
node contador.js 5
```

Salida esperada:

```
1
2
3
4
5
```

Paso 3: prueba los casos de error para verificar la validación:

```
node contador.js
node contador.js hola
node contador.js -3
```

En los tres casos debes ver el mensaje de uso, no un error críptico ni un programa colgado.

Paso 4 (para curiosos): prueba con un número grande, como `node contador.js 100000`, y observa qué tan rápido Node.js imprime cien mil líneas. Esa velocidad es el motor V8 trabajando.

¿Qué practicaste? Validación de entradas con `isNaN()` (que detecta cuando un valor no es un número), el ciclo `for`, y el patrón completo de "validar primero, ejecutar después". Este es el ejercicio más completo de la unidad porque junta todo: argumentos de terminal, conversión de tipos, validación, control de flujo y salida. Si lo resolviste (o lo entendiste a fondo leyendo la solución), estás listo para la unidad 2, donde por fin construiremos nuestro primer servidor.

Resumen de la unidad

Llegamos al final de la primera unidad, y vale la pena detenerse a mirar el camino recorrido, porque fue largo y cada paso sostiene al siguiente. Empezamos con una idea poderosa: JavaScript ya no vive solo en el navegador. Node.js es el entorno de ejecución que lo liberó, construido sobre el motor V8 de Chrome, y que le dio superpoderes de servidor: acceso al sistema de archivos, a la red y al sistema operativo. Aprendimos que Node.js no es un lenguaje nuevo ni una moda, sino la cocina grande donde el cocinero que ya conocías (tu JavaScript) puede preparar banquetes completos.

Conocimos la historia: un programador llamado Ryan Dahl, frustrado en 2009 con el desperdicio de los servidores bloqueantes, tomó el motor V8, le sumó entrada y salida no bloqueante y presentó Node.js en Berlín ante una sala que se puso de pie. Desde entonces, npm se convirtió en el registro de código más grande del mundo y empresas como Netflix, Uber, LinkedIn y PayPal construyeron sobre Node.js sistemas que usan millones de personas a diario. La lección de fondo: Node.js nació para resolver un problema real, y entender ese problema (atender miles de clientes sin desperdiciar recursos) es entender la tecnología.

Nos metimos hasta el corazón del asunto con el event loop, usando la analogía del mesero que atiende todo el restaurante sin sentarse nunca a esperar: un solo hilo que toma órdenes (peticiones), las pasa a la cocina (libuv y el sistema), sigue atendiendo a los demás y retoma cada tarea cuando está lista. Comparamos ese modelo con el tradicional de un hilo por cliente (el restaurante que quiebra pagando mil meseros) y dejamos clara la regla de oro: nunca

bloquees el event loop. Distinguimos operaciones bloqueantes de no bloqueantes con el ejemplo de la fila de las tortillas contra pedir tacos y sentarse a platicar, y aprendimos que en Node.js las funciones síncronas llevan "Sync" en el nombre y se evitan en el camino de las peticiones.

En el terreno práctico, instalamos Node.js y npm paso a paso en Windows, Mac y Linux, aprendimos a elegir siempre la versión LTS, verificamos la instalación con `node -v` y `npm -v`, conocimos npm como la refaccionaria de código más grande del planeta, descubrimos nvm para manejar varias versiones sin dramas de permisos, y jugamos con el REPL como laboratorio de experimentos instantáneos. Escribimos y ejecutamos nuestro primer script (hola.js con `node hola.js`), resolvimos los errores clásicos de instalación (el PATH, los permisos EACCES sin usar sudo, las versiones viejas, el antivirus) y adoptamos buenas prácticas de profesional desde el día uno.

Cerramos con código de verdad: scripts que suman números desde la terminal con `process.argv`, que leen la hora del sistema con `Date`, recorridos por el REPL y cinco ejercicios que te llevaron de verificar tu instalación a construir una calculadora de terminal y un contador con validación. Si hiciste todo eso con tus propias manos, ya no eres solo alguien que "sabe qué es Node.js": eres alguien que tiene Node.js instalado, que ejecuta scripts y que entiende por dentro cómo funciona. Esa es una base sólida. En la unidad 2 vamos a usarla para construir lo que todo el mundo espera de un curso backend: tu primer servidor web.

Términos clave de la unidad

- Node.js: entorno de ejecución de JavaScript fuera del navegador, construido sobre el motor V8.
- Entorno de ejecución (runtime): programa que lee tu código y lo ejecuta usando los recursos de la computadora.
- Motor V8: motor de JavaScript creado por Google para Chrome; compila a código máquina con técnica JIT para máxima velocidad.
- JIT (Just-In-Time): técnica de compilación que convierte el código a lenguaje máquina durante la ejecución.
- Ryan Dahl: creador de Node.js; lo presentó en 2009 en la conferencia JSConf EU de Berlín.
- libuv: biblioteca en C que maneja las operaciones de entrada y salida no bloqueantes por debajo de Node.js.

- Event loop: ciclo que corre en un solo hilo revisando trabajo pendiente y ejecutando callbacks cuando sus operaciones terminan.
- Hilo (thread): secuencia de ejecución dentro de un programa; Node.js ejecuta tu código JavaScript en un solo hilo.
- Operación bloqueante: detiene la ejecución hasta terminar; en Node.js las funciones síncronas llevan "Sync" en el nombre.
- Operación no bloqueante: inicia la tarea, devuelve el control de inmediato y avisa con un callback al terminar.
- Callback: función que se ejecuta cuando termina una operación asíncrona.
- LTS (Long Term Support): versión estable de Node.js con soporte a largo plazo; la recomendada para aprender y producir.
- npm: manejador de paquetes de Node.js; permite instalar y compartir bibliotecas de código; es el registro de software más grande del mundo.
- Paquete: pedazo de código reutilizable publicado para que otros lo instalen con npm.
- nvm: herramienta para instalar y alternar entre varias versiones de Node.js sin permisos de administrador.
- REPL (Read, Eval, Print, Loop): consola interactiva de Node.js para probar código al instante; se entra con node y se sale con .exit.
- PATH: variable del sistema que indica en qué carpetas buscar los programas; causa típica del error "node no se reconoce".
- Script: archivo de código (por ejemplo hola.js) que se ejecuta con el comando node seguido del nombre del archivo.
- process.argv: arreglo con los argumentos recibidos desde la terminal; los argumentos del usuario empiezan en la posición 2.
- console.log: función para imprimir mensajes y valores en la terminal.

UNIDAD 2: El sistema de módulos: require, import y globales

Si la Unidad 1 fue como conocer el motor de un coche, esta unidad es aprender a armar el coche completo: piezas, conexiones y el tablero de instrumentos. En programación, esas piezas se llaman módulos. Cuando termines esta unidad vas a saber dividir tu código en archivos ordenados, conectar esos archivos entre sí de dos formas distintas (require e import), leer lo que el usuario escribe en la terminal, guardar contraseñas y claves fuera de tu código, y usar las variables globales que Node.js te regala. Todo eso es lo que separa un script de juguete de un programa de verdad.

Antes de empezar, date cuenta de algo: en las clases 4, 5 y 6 del curso vimos los temas por separado, pero aquí los juntamos porque en la vida real siempre van juntos. Nadie escribe un programa serio en un solo archivo. Nadie publica código con las contraseñas escritas dentro. Nadie hace una herramienta de terminal sin leer los argumentos que le pasa el usuario. Esta unidad es el puente entre "ya sé escribir JavaScript" y "ya sé construir software con Node.js". Tómalala con calma, porque cada concepto de aquí lo vas a usar todos los días que programes.

Y un aviso importante: esta unidad cubre los dos sistemas de módulos de Node.js, CommonJS (require) y ES Modules (import). Al principio puede parecer que estamos aprendiendo lo mismo dos veces, pero no es así: son dos mundos con reglas distintas, y te los vas a encontrar mezclados en tutoriales, proyectos viejos y proyectos nuevos. Dominar los dos es lo que te hace contratarle como programador Node.js y no solo como "alguien que vio un tutorial".

Explicación detallada

Por qué existen los módulos: piezas de LEGO y la cocina de una fonda

Imagina que quieres armar un castillo de LEGO. No compras una sola pieza gigante con forma de castillo: compras una caja con cientos de piezas pequeñas que encajan entre sí. Cada pieza hace una cosa, y juntándolas construyes lo que quieras. Si una pieza se rompe, la cambias por otra igual y el castillo sigue en pie. Si quieres construir un barco en vez de un castillo, reutilizas muchas de las mismas piezas.

El código funciona igual. Un módulo es una pieza de LEGO de tu programa: un archivo (o un conjunto de archivos) que hace una sola cosa bien hecha y que otros archivos pueden usar.

En vez de escribir un archivo gigante de 5,000 líneas donde todo está revuelto, divides tu programa en archivos pequeños: uno para las matemáticas, otro para manejar usuarios, otro para conectarse a la base de datos, otro para las rutas de tu servidor. Cada archivo exporta lo que otros necesitan e importa lo que él necesita.

Otra forma de verlo: piensa en la cocina de una fonda. Hay una estación para los guisados, otra para las tortillas, otra para las aguas frescas. Cada cocinero es experto en lo suyo y entrega su parte lista: el de los guisados no necesita saber cómo se hacen las tortillas, solo necesita que le lleguen calientes a tiempo. En tu programa, cada módulo es un cocinero: hace su trabajo, entrega el resultado y no se mete en lo demás. Si mañana cambias la receta de las aguas frescas, los guisados ni se enteran.

Los beneficios concretos de trabajar con módulos son estos:

- Orden: cada archivo tiene un propósito claro y un nombre que lo dice. Encontrar un error es mucho más fácil cuando sabes en qué archivo vive.
- Reutilización: una función que escribes una vez la puedes usar en diez programas distintos sin copiarla y pegarla.
- Trabajo en equipo: dos programadores pueden trabajar en dos módulos distintos sin pisarse el código.
- Pruebas: probar una función aislada en su propio archivo es mucho más sencillo que probarla enterrada en medio de un archivo gigante.
- Mantenimiento: cambiar una pieza no rompe las demás, siempre que respetes lo que esa pieza promete entregar.

Node.js nació con su propio sistema de módulos porque en 2009 JavaScript no tenía uno oficial. El navegador tampoco lo tenía: cada script compartía todo con todos los demás scripts, como si en la fonda todos los cocineros usaran la misma olla. Eso funcionaba para páginas pequeñas, pero era un desastre para programas grandes. Node.js inventó CommonJS, su sistema de módulos con `require` y `module.exports`, y años después JavaScript adoptó oficialmente los ES Modules con `import` y `export`. Hoy Node.js soporta los dos sistemas, y tú tienes que dominar los dos porque te los vas a encontrar en todos lados: `require` en proyectos viejos y en muchísimos tutoriales, `import` en los proyectos modernos y en el frontend.

CommonJS a fondo: el sistema clásico de Node.js

CommonJS es el sistema de módulos original de Node.js. Lo reconoces por dos palabras clave: `require` para pedir algo y `module.exports` para entregar algo. Si ya programaste algo en Node.js, casi seguro ya lo usaste sin darte cuenta.

La idea es simple. Tienes un archivo `matemáticas.js` con funciones útiles:

```
function suma(a, b) {  
  return a + b;  
}  
  
function resta(a, b) {  
  return a - b;  
}  
  
module.exports = { suma, resta };
```

Y en otro archivo, `app.js`, las usas así:

```
const { suma, resta } = require("./matematicas");  
  
console.log(suma(5, 3));  
console.log(resta(10, 4));
```

Esa última línea de `matemáticas.js`, `module.exports = { suma, resta }`, es como poner tus guisados en la ventanilla de la cocina: "esto es lo que este archivo ofrece al resto del programa". Y el `require("./matematicas")` en `app.js` es como el mesero que pasa a recoger el pedido: "tráeme lo que ofrece el archivo `matemáticas`".

module.exports vs exports: la diferencia que confunde a todos

Aquí hay un detalle que confunde a casi todos los principiantes, así que vamos despacio. En cada archivo de Node.js, antes de que tu código corra, Node.js crea dos cosas: un objeto llamado `module` y una variable llamada `exports` que al inicio apunta al mismo objeto que `module.exports`. Es como si Node.js escribiera esto invisiblemente al principio de tu archivo:

```
const module = { exports: {} };  
const exports = module.exports;
```

Entonces tienes dos formas de exportar. La forma segura y recomendada es asignar directamente a `module.exports`:

```
module.exports = { suma, resta };
```

También puedes agregar propiedades una por una usando el atajo `exports`:

```
exports.suma = suma;  
exports.resta = resta;
```

Esto funciona porque exports y module.exports apuntan al mismo objeto, así que agregarle propiedades a uno es agregárselas al otro. Pero ojo con la trampa mortal: si reasignas exports directamente, rompes la conexión:

```
exports = { suma, resta };
```

Esta línea NO funciona. ¿Por qué? Porque exports era solo un apodo, una referencia al objeto original. Al hacer exports = {...} le dices a la variable exports que ahora apunte a un objeto nuevo, pero module.exports sigue apuntando al objeto viejo y vacío. Node.js entrega module.exports, no exports, así que tu archivo terminaría exportando un objeto vacío. Es como si en la fonda el cocinero le entregara los guisados a un mesero que renunció ayer: la ventanilla oficial sigue vacía.

Regla de oro que te va a ahorrar horas de frustración: usa siempre module.exports cuando vayas a exportar algo completo, y usa exports.soloNombre solo para agregar propiedades sueltas. Nunca reasignas exports directamente. Si recuerdas una sola cosa de esta sección, que sea esta.

También puedes exportar una sola función o un solo valor, sin objeto:

```
function saludar(nombre) {  
  return "Hola, " + nombre;  
}  
  
module.exports = saludar;
```

Y del otro lado:

```
const saludar = require("../saludos");  
console.log(saludar("Esteban"));
```

Esto es muy común para clases y para funciones principales de un módulo. Cuando veas const express = require("express") en un proyecto, es exactamente este patrón: el módulo exporta una sola cosa (una función constructora) y el que la pide la recibe directamente.

Cómo funciona require por dentro: la cocina paso a paso

Entender qué hace require cuando lo llamas te convierte de usuario en conocedor. Esto es lo que pasa, en orden, cuando escribes `const x = require("./matemáticas")`:

1. Resolución: Node.js averigua a qué archivo te refieres. Si la ruta empieza con `./` o `../`, busca un archivo en tu proyecto. Si no empieza así, primero revisa si es un módulo core (como `fs` o `path`) y si no, busca en la carpeta `node_modules`. Más abajo veremos esto con lupa.
2. Carga: Node.js lee el contenido del archivo del disco.
3. Envoltura: Node.js envuelve tu código en una función invisible antes de ejecutarlo. Tu archivo no corre "al aire libre": corre dentro de algo parecido a esto:

```
function (exports, require, module, __filename, __dirname) {  
  // aquí va todo tu código  
}
```

Por eso dentro de cada archivo tienes gratis las variables `exports`, `require`, `module`, `__filename` y `__dirname` sin haberlas declarado: Node.js te las pasa como argumentos de esa función envolvente. Y por eso las variables que declares con `const` o `let` en un archivo NO se ven desde otros archivos: viven dentro de esa función, son locales al módulo. Cada archivo es su propio cuarto cerrado; lo único que sale del cuarto es lo que pongas en `module.exports`. Esta es la magia que evita que todos los scripts choquen entre sí, el problema que tenía JavaScript en el navegador antes de los módulos.

4. Ejecución: Node.js ejecuta el código del módulo una sola vez.
5. Caché: el resultado (lo que quedó en `module.exports`) se guarda en una caché. Si otro archivo pide el mismo módulo con `require`, Node.js ya no vuelve a leer ni ejecutar el archivo: le entrega la copia guardada.
6. Devolución: `require` te regresa `module.exports` del módulo cargado.

El punto 5, la caché, merece atención porque causa sorpresas. Imagina un módulo `contador.js`:

```

let conteo = 0;

function incrementar() {
  conteo = conteo + 1;
  return conteo;
}

module.exports = { incrementar };

```

Si app.js y otro archivo utilidades.js hacen require("./contador"), los dos reciben el MISMO objeto, con el MISMO conteo compartido. Si app.js llama incrementar() dos veces y luego utilidades.js lo llama una vez, el resultado será 3, no 1. Los módulos con estado se comportan como singletons: hay una sola instancia en todo tu programa. Esto es útil (por ejemplo, para una conexión a base de datos que quieres compartir), pero peligroso si no lo esperas. Si necesitas objetos independientes, exporta una función que cree objetos nuevos en vez de exportar el objeto ya creado:

```

function crearContador() {
  let conteo = 0;
  return {
    incrementar: function () {
      conteo = conteo + 1;
      return conteo;
    }
  };
}

module.exports = { crearContador };

```

Así cada archivo que pida crearContador() obtiene su propio contador con su propio conteo. Este patrón se llama "fábrica" y lo verás muchísimo en código profesional.

Otro detalle de la caché que debes conocer: como el módulo se ejecuta una sola vez, cualquier código que pongas fuera de las funciones corre una sola vez, en el momento del primer require. Esto se usa para inicializaciones: abrir una conexión, leer un archivo de configuración, registrar algo. Pero también significa que si tu módulo tiene efectos secundarios (como imprimir en consola o modificar archivos), esos efectos ocurren en el momento del require, te guste o no. Los módulos bien diseñados exportan funciones y no hacen ruido al cargarse; el trabajo pesado ocurre cuando alguien llama a esas funciones.

Resolución de rutas: cómo Node.js encuentra lo que pides

Cuando escribes `require("algo")`, Node.js tiene que decidir dónde buscar ese "algo". Las reglas son precisas y vale la pena memorizarlas, porque el error "Cannot find module" casi siempre nace de no entenderlas. Hay tres casos:

Caso 1: rutas relativas. Si el texto empieza con `./` o `../`, Node.js busca un archivo en relación a la ubicación del archivo que está pidiendo. `require("./matemáticas")` desde `app.js` busca `matemáticas.js` en la misma carpeta que `app.js`. `require("../utilidades/ayuda")` sube una carpeta y entra a utilidades. Puedes omitir la extensión `.js`: Node.js prueba primero el nombre exacto, luego le agrega `.js`, luego `.json`, luego `.node`. También puedes apuntar a una carpeta: `require("./mi-carpeta")` busca primero `mi-carpeta/package.json` y su campo "main", y si no existe, prueba `mi-carpeta/index.js`. Este es el caso que usas para tu propio código, para conectar tus archivos entre sí.

Caso 2: módulos core. Si el texto NO empieza con `./` ni `../`, Node.js primero revisa su lista interna de módulos core: `fs`, `path`, `http`, `os`, `events`, `util`, `crypto` y otros. Estos módulos vienen incluidos con Node.js, no necesitas instalar nada. `require("fs")` siempre funciona, en cualquier proyecto, sin `node_modules`. Son las herramientas básicas de la caja: leer archivos, manejar rutas, crear servidores, trabajar con el sistema operativo.

Caso 3: `node_modules`. Si no es ruta relativa ni módulo core, Node.js busca en la carpeta `node_modules`. Pero no solo en la de tu proyecto: empieza en la carpeta del archivo que pide, y si no encuentra `node_modules` ahí, sube un nivel y vuelve a buscar, y así hasta llegar a la raíz del disco. Por eso puedes instalar un paquete en la raíz de tu proyecto y usarlo desde subcarpetas profundas sin rutas raras. Cuando encuentra la carpeta del paquete, lee su `package.json`, busca el campo "main" para saber qué archivo cargar, y si no hay `package.json` o no hay "main", carga `index.js` por defecto.

El orden importa: core primero, luego `node_modules`. Esto significa que si instalas un paquete llamado "fs" (no lo hagas), el `require("fs")` seguirá trayendo el módulo core, no tu paquete. Los módulos core siempre ganan.

Un error clásico: olvidar el `./` al pedir tu propio archivo. `require("matemáticas")` NO busca `matemáticas.js` en tu carpeta: como no empieza con `./`, Node.js lo trata como paquete y lo busca en `node_modules`, no lo encuentra y te lanza "Cannot find module 'matemáticas'". La solución es de un carácter y dos puntos: `require("./matemáticas")`. Cuando veas ese error, lo primero que debes revisar es si tu ruta relativa lleva el `./` al inicio.

Otro error clásico: las rutas en `require` son relativas al archivo que las escribe, no a la carpeta donde corres el comando. Si tienes `proyecto/src/app.js` haciendo `require("../datos")`, busca

proyecto/src/datos.js, aunque corras el programa desde proyecto/ o desde tu carpeta personal. Esto confunde al principio, pero es lo correcto: el código debe funcionar igual lo corras desde donde lo corras.

Los módulos core: la caja de herramientas incluida

Node.js trae una colección de módulos core que son tu caja de herramientas básica. No necesitas instalarlos ni importarlos de ningún lado especial: están ahí siempre. Estos son los que vas a usar más:

- fs (file system): leer, escribir, crear y borrar archivos y carpetas. El módulo más usado de todos.
- path: construir y manipular rutas de archivos de forma segura, sin importar si el sistema es Windows, Mac o Linux.
- http y https: crear servidores web y hacer peticiones. La base sobre la que se construyen Express y todos los frameworks.
- os: información del sistema operativo: memoria, procesadores, usuario actual, carpetas del sistema.
- events: el sistema de eventos, la base de la programación reactiva en Node.js.
- útil: utilidades varias, como formatear texto y convertir funciones viejas a promesas.
- crypto: criptografía: generar hashes, cifrar datos, crear tokens seguros.
- url y querystring: trabajar con direcciones web y sus parámetros.

En esta unidad no vamos a estudiar cada uno a fondo (eso viene en unidades posteriores), pero sí necesitas saber que existen y cómo pedirlos, porque los ejemplos de process y __dirname los usan. Por ejemplo, para saber en qué sistema operativo estás:

```
const os = require("os");

console.log("Sistema: " + os.platform());
console.log("Memoria libre: " + os.freemem());
console.log("Usuario: " + os.userInfo().username);
```

Fíjate: require("os") sin ./ porque es módulo core. Si le pusieras ./ buscaría un archivo os.js en tu carpeta y fallaría. La ausencia o presencia del ./ es la señal que le dice a Node.js en qué modo buscar.

ES Modules a fondo: el sistema moderno de JavaScript

Los ES Modules (también llamados ESM) son el sistema oficial de módulos de JavaScript, el que usa el navegador moderno y el que usan los proyectos nuevos de Node.js. Lo reconoces por las palabras `import` y `export`. Si vienes del frontend (React, por ejemplo), esta sintaxis te va a resultar familiar, porque es exactamente la misma: eso es a propósito, para que el mismo código pueda correr en el navegador y en el servidor.

La idea básica es la misma que en CommonJS: un archivo entrega cosas con `export` y otro las pide con `import`. Pero la sintaxis y las reglas cambian. Mira la misma calculadora de antes, ahora en ESM. Archivo `matemáticas.js`:

```
export function suma(a, b) {  
  return a + b;  
}  
  
export function resta(a, b) {  
  return a - b;  
}
```

Archivo `app.js`:

```
import { suma, resta } from "./matematicas.js";  
  
console.log(suma(5, 3));  
console.log(resta(10, 4));
```

Tres diferencias saltan a la vista de inmediato. Primera: `export` va pegado a la declaración, no al final en un objeto. Segunda: `import` usa llaves para pedir nombres específicos. Tercera, y la más traicionera: la ruta lleva la extensión `.js` obligatoria. En CommonJS podías escribir `require("./matemáticas")` sin extensión; en ESM, `import { suma } from "./matemáticas"` (sin `.js`) FALLA. Node.js en modo ESM no adivina extensiones: tienes que escribir la ruta completa y exacta, incluyendo `.js`. Este es el error número uno de quien migra de `require` a `import`, así que grábatelo: en ESM, siempre la extensión.

Exportaciones nombradas vs exportación por defecto

ESM tiene dos formas de exportar, y entender la diferencia es clave porque determina cómo se importa del otro lado.

Las exportaciones nombradas son las que vimos arriba: exportas varias cosas, cada una con su nombre, y el que importa pide exactamente los nombres que quiere, entre llaves:

```
// utilidades.js
export function mayusculas(texto) {
  return texto.toUpperCase();
}

export function minusculas(texto) {
  return texto.toLowerCase();
}

export const VERSION = "1.0";
```

```
// app.js
import { mayusculas, VERSION } from "../utilidades.js";
// pido solo lo que necesito; minusculas se queda fuera
```

Puedes cambiarle el nombre a lo que importas con `as`, útil cuando hay choques de nombres:

```
import { mayusculas as aMayusculas } from "../utilidades.js";
```

La exportación por defecto es distinta: cada archivo puede tener UNA sola exportación marcada como "la principal", la que representa al módulo. Se usa mucho para la cosa más importante del archivo: una clase, una función principal, un objeto de configuración.

```
// saludos.js
export default function saludar(nombre) {
  return "Hola, " + nombre;
}
```

```
// app.js
import saludar from "../saludos.js";
// sin llaves, y puedo llamarla como quiera:
import decirHola from "../saludos.js"; // también válido
```

Fíjate en las diferencias: la exportación por defecto se importa SIN llaves y puedes ponerle el nombre que quieras al importarla, porque no estás pidiendo "la cosa llamada saludar" sino "la cosa principal del archivo, llámala como quieras". Con las nombradas, en cambio, el nombre debe coincidir (o usar `as` para renombrarlo explícitamente).

Puedes mezclar ambas en un mismo archivo, aunque no es lo más común:

```
// calculadora.js
export default function calculadora() {
  return "Soy la calculadora principal";
}

export function suma(a, b) {
  return a + b;
}
```

```
// app.js
import calculadora, { suma } from "./calculadora.js";
```

La convención de la industria: usa exportaciones nombradas para utilidades (varias funciones sueltas) y exportación por defecto para "la cosa principal" del archivo (una clase, un componente de React, una función principal). Cuando dudes, las nombradas son más explícitas y más fáciles de rastrear en proyectos grandes.

Cómo activar ESM en Node.js: el package.json y "type": "module"

Aquí viene la parte administrativa que nadie te explica y que causa la mitad de los dolores de cabeza. Node.js necesita saber en qué modo tratar tus archivos .js: ¿modo CommonJS (require) o modo ESM (import)? Por defecto, los archivos .js se tratan como CommonJS, por compatibilidad con los millones de proyectos viejos. Para decirle a Node.js "este proyecto es moderno, usa ESM", tienes dos caminos:

Camino 1: en tu package.json, agrega el campo "type": "module":

```
{
  "name": "mi-proyecto",
  "version": "1.0.0",
  "type": "module"
}
```

Con eso, todos los .js de tu proyecto se tratan como ES Modules: puedes usar import/export libremente, y require deja de funcionar en esos archivos. Si necesitas un archivo suelto en CommonJS dentro de un proyecto ESM, nómbralo con extensión .cjs y Node.js lo tratará como CommonJS aunque el proyecto sea ESM.

Camino 2: usa la extensión .mjs para tus archivos ESM. Los archivos .mjs siempre se tratan como ES Modules, sin importar lo que diga el package.json. Es menos común en proyectos completos, pero útil para scripts sueltos.

La regla completa, para que la tengas de referencia:

- .js con "type": "module" en package.json -> ESM (import/export).
- .js sin "type": "module" (o con "type": "commonjs") -> CommonJS (require/module.exports).
- .mjs -> siempre ESM.
- .cjs -> siempre CommonJS.

Memoriza esta tabla o guárdala en tus notas: es la respuesta al 90% de los errores de "import no funciona" o "require is not defined".

Diferencias prácticas entre CommonJS y ESM: lo que cambia en tu día a día

Más allá de la sintaxis, los dos sistemas se comportan distinto en formas que te van a afectar. Estas son las diferencias que de verdad importan:

1. Momento de carga: require es dinámico, import es estático. Con require puedes pedir un módulo en cualquier momento, incluso dentro de un if o una función, y puedes construir el nombre del módulo con variables: require("./idiomas/" + idioma). Con import, las declaraciones van al inicio del archivo y la ruta debe ser un texto literal: no puedes hacer import { x } from "." + nombre. (ESM tiene una versión dinámica, import(), que veremos en unidades avanzadas, pero el import normal es estático.) Esto hace que el código ESM sea más predecible y más fácil de analizar para las herramientas, pero menos flexible en casos raros.
2. Los valores importados son de solo lectura. En ESM, si importas una variable y el módulo original la cambia, tú ves el cambio (es una referencia viva), pero TÚ no puedes reasignarla desde tu archivo: import { contador } y luego contador = 5 es un error. En CommonJS, lo que recibes es una copia de la referencia al objeto exportado y puedes hacer lo que quieras con tu copia. En la práctica, esto significa que en ESM los módulos se comunican de forma más disciplinada: si un módulo quiere compartir estado cambiante, exporta funciones para modificarlo en vez de variables sueltas.
3. __dirname y __filename no existen en ESM. En CommonJS los tienes gratis por la envoltura invisible; en ESM no hay tal envoltura, así que no existen. La alternativa es import.meta.url, que veremos en detalle más abajo. Es uno de los errores más comunes al migrar código viejo a ESM.

4. La extensión en las rutas: ya lo dijimos, pero lo repetimos porque es el error más frecuente: ESM exige la extensión completa (.js) en las rutas relativas; CommonJS la adivina.
5. Los JSON: en CommonJS puedes hacer `require("./datos.json")` y Node.js lo parsea automáticamente. En ESM necesitas la sintaxis de atributos de importación: `import datos from "./datos.json" with { type: "json" }`. Más verboso, pero más explícito.
6. `require` de un módulo ESM no se puede (directamente). Un archivo CommonJS no puede hacer `require()` de un archivo ESM. Al revés sí hay una vía: un archivo ESM puede importar un módulo CommonJS usando `import` (Node.js lo convierte automáticamente). Esta asimetría es importante cuando mezclas proyectos: la dirección permitida es ESM -> CommonJS, no al revés.

Cómo elegir entre CommonJS y ESM: la guía práctica

La pregunta del millón: ¿cuál uso en mi proyecto? La respuesta honesta tiene matices, pero aquí va la guía directa:

- Proyecto nuevo desde cero: usa ESM (`import/export`) con `"type": "module"`. Es el estándar oficial de JavaScript, es lo que usa el frontend moderno, es lo que enseñan los cursos actuales y es hacia donde va el ecosistema. Todo lo nuevo que aprendas te servirá en ambos lados (navegador y servidor).
- Proyecto viejo o heredado: respeta lo que ya hay. Si el proyecto usa `require`, sigue usando `require`. Mezclar los dos sistemas en un mismo proyecto es posible pero genera fricción (la asimetría del punto 6, el tema de `__dirname`, las extensiones). No migres un proyecto entero a ESM solo porque sí: es un trabajo delicado que se hace con razón y con pruebas.
- Tutoriales y Stack Overflow: vas a encontrar `require` en muchísimos ejemplos viejos. Ahora ya sabes traducirlos mentalmente a `import` cuando tu proyecto es ESM, y sabes que si copias un ejemplo con `require` a un proyecto ESM, no va a funcionar tal cual.
- Librerías que publicas para otros: el mundo moderno espera ESM, pero muchas librerías publican en ambos formatos para máxima compatibilidad. No es tu preocupación ahora, pero que sepas que existe.
- Scripts rápidos de una sola vez: cualquiera de los dos sirve. Elige el que te salga más rápido escribir.

En este curso, a partir de ahora, usaremos ESM como sistema principal (porque es el presente y el futuro), pero seguiremos mostrando CommonJS donde sea relevante, porque el mundo

real está lleno de require y tienes que leerlo con fluidez. Ser bilingüe en módulos no es opcional: es requisito del oficio.

Un consejo final sobre la mezcla: si en un proyecto ESM necesitas usar una librería vieja que solo existe en CommonJS, no te preocupes: import funciona. Puedes hacer `import express from "express"` aunque express esté escrito con `module.exports`; Node.js hace la traducción automáticamente y la exportación principal del CommonJS se convierte en el default del ESM. Es la dirección fácil de la asimetría, y cubre el 99% de los casos reales.

El objeto process: el tablero de instrumentos de tu programa

Si los módulos son las piezas del coche, process es el tablero de instrumentos: te dice qué está pasando, te deja leer lo que el conductor (el usuario) indica y te permite apagar el motor cuando quieras. process es un objeto global, disponible en cualquier archivo sin importarlo ni pedirlo con require. Representa el proceso actual de Node.js: tu programa corriendo.

Vamos a conocer sus partes más útiles, de la más usada a la más avanzada.

process.argv: leer lo que el usuario escribe en la terminal

Cuando corres un programa desde la terminal, puedes pasarle argumentos: `node saludar.js Esteban 30`. Esos argumentos llegan a tu programa a través de `process.argv`, que es un arreglo (array) con todas las palabras del comando. Pero ojo: las dos primeras posiciones no son tus argumentos, son información del sistema:

- `process.argv[0]`: la ruta del ejecutable de Node.js.
- `process.argv[1]`: la ruta de tu archivo script.
- `process.argv[2]` en adelante: tus argumentos de verdad.

Mira este programa, `saludar.js`:

```
const nombre = process.argv[2];
const edad = process.argv[3];

console.log("Hola, " + nombre + ". Tienes " + edad + " años.");
```

Si lo corres así:

```
node saludar.js Esteban 30
```


`process.argv` vale algo como `["/usr/bin/node", "/home/esteban/saludar.js", "Esteban", "30"]`, y el programa imprime "Hola, Esteban. Tienes 30 años." Cambia los argumentos y el programa responde distinto, sin cambiar el código. Eso es una herramienta de línea de comandos (CLI): un programa que se comporta según lo que le pides al ejecutarlo.

Detalles importantes que te van a ahorrar confusiones:

- Todo lo que llega por `argv` es string. Si pasas 30, llega como el texto "30", no como el número 30. Si necesitas hacer matemáticas, conviértelo: `const edad = Number(process.argv[3])`. Es la misma lección de tipos de la unidad anterior, aplicada a la terminal.
- Si el usuario no pasa un argumento, su posición vale `undefined`. Un programa bien hecho verifica: `if (!process.argv[2]) { console.log("Uso: node saludar.js <nombre> <edad>"); process.exit(1); }`. Mostrar cómo se usa el programa cuando faltan datos es cortesía básica de toda CLI.
- Los argumentos con espacios necesitan comillas en la terminal: `node saludar.js "María José"` funciona y llega como un solo argumento; sin comillas llegarían dos ("María" y "José").

Un ejemplo más completo: una mini-calculadora CLI, `calc.js`:

```

const operacion = process.argv[2];
const a = Number(process.argv[3]);
const b = Number(process.argv[4]);

if (!operacion || Number.isNaN(a) || Number.isNaN(b)) {
  console.log("Uso: node calc.js <suma|resta|multiplica|divide> <num1> <num2>");
  process.exit(1);
}

let resultado;

if (operacion === "suma") {
  resultado = a + b;
} else if (operacion === "resta") {
  resultado = a - b;
} else if (operacion === "multiplica") {
  resultado = a * b;
} else if (operacion === "divide") {
  if (b === 0) {
    console.log("Error: no se puede dividir entre cero");
    process.exit(1);
  }
  resultado = a / b;
} else {
  console.log("Operación no válida. Usa: suma, resta, multiplica o divide");
  process.exit(1);
}

console.log("Resultado: " + resultado);

```

Corres `node calc.js suma 10 5` y obtienes "Resultado: 15". Corres `node calc.js divide 10 0` y el programa te explica el error y se detiene. Fíjate en el patrón: validar primero (¿me dieron todo lo que necesito? ¿son números?), trabajar después. Y fíjate en `process.exit(1)`: detiene el programa inmediatamente. El número entre paréntesis es el código de salida, que veremos ahora.

process.exit: apagar el motor (y decir por qué)

`process.exit()` termina tu programa en ese instante, sin ejecutar lo que sigue. Acepta un código numérico opcional que le dice al sistema operativo cómo terminó todo:

- `process.exit(0)` o simplemente `process.exit()`: todo salió bien. El 0 significa éxito.
- `process.exit(1)`: algo salió mal. Es el código genérico de error.

¿Por qué importa el número? Porque otros programas y scripts leen ese código. Si tu CLI falla y un script automático la está usando, el script necesita saber si funcionó o no, y lo sabe por el código de salida. Es el apretón de manos entre tu programa y el resto del sistema: 0 es "misión cumplida", cualquier otro número es "algo falló".

Úsalo en dos situaciones: cuando los argumentos son inválidos (como en la calculadora) y cuando ocurre un error irreparable del que no tiene sentido seguir. No abuses: en un servidor web casi nunca quieres `process.exit`, porque mataría el servidor completo por el error de una sola petición. Para CLIs y scripts, en cambio, es pan de cada día.

process.env: las variables de entorno y el arte de no quemar tus claves

Esta sección te va a ahorrar uno de los errores más caros y vergonzosos de la carrera: subir tus contraseñas a internet. Vamos por partes.

Tu programa muchas veces necesita datos sensibles: la contraseña de la base de datos, la clave secreta de un servicio, un token de acceso. La tentación del principiante es escribirlos directamente en el código:

```
const passwordBD = "miPasswordSuperSecreta123"; // NUNCA HAGAS ESTO
```

¿Por qué es terrible? Porque el código se comparte: lo subes a GitHub, se lo pasas a un compañero, lo respaldas en la nube. Y cada copia lleva tus claves escritas. Ha habido casos famosos de programadores que subieron claves de servicios en la nube a repositorios públicos y despertaron con cuentas de miles de dólares consumidas por extraños. No es un riesgo teórico: pasa todas las semanas.

La solución profesional: las variables de entorno. Son valores que viven FUERA de tu código, en el ambiente donde corre el programa, y tu código los lee con `process.env`. En la terminal, defines una antes de correr tu programa:

```
PASSWORD_BD="miPasswordSuperSecreta123" node app.js
```

Y en `app.js`:

```
const passwordBD = process.env.PASSWORD_BD;  
console.log("Conectando con la contraseña: " + passwordBD);
```

El código ya no contiene la clave: solo dice "léela del ambiente". Puedes compartir tu código tranquilamente, porque sin el ambiente correcto no sirve de nada. Cada ambiente (tu

computadora, el servidor de pruebas, el servidor de producción) tiene sus propios valores, y el mismo código funciona en todos sin cambios. Esa es la idea: el código es público y genérico, los secretos son privados y específicos del ambiente.

Convenciones que debes seguir: los nombres de variables de entorno van en MAYÚSCULAS con guiones bajos (PASSWORD_BD, API_KEY, PUERTO). Y recuerda: todo lo que lees de process.env es string. Si guardas un puerto como PUERTO="3000", al leerlo obtienes "3000" y para usarlo como número lo conviertes: `const puerto = Number(process.env.PUERTO)`.

dotenv: el archivo .env para no volverte loco

Definir variables en la terminal funciona, pero es incómodo: tendrías que escribirlas cada vez que corres el programa, y con diez variables es un suplicio. Para eso existe dotenv, el paquete más estándar del ecosistema para este problema. La idea: guardas tus variables en un archivo llamado .env (que NO se sube al repositorio), y dotenv las carga a process.env automáticamente al iniciar tu programa.

Primero instalas el paquete (esto lo veremos a fondo en la unidad de npm, pero adelanto el uso):

```
npm install dotenv
```

Luego creas un archivo .env en la raíz de tu proyecto:

```
PASSWORD_BD=miPasswordSuperSecreta123
API_KEY=abc123xyz
PUERTO=3000
```

Sin comillas, sin espacios alrededor del igual (la mayoría de las veces), una variable por línea, sin punto y coma. Simple como una lista de compras.

Y al inicio de tu programa:

```
require("dotenv").config(); // CommonJS
// o en ESM:
// import "dotenv/config";

const passwordBD = process.env.PASSWORD_BD;
const puerto = Number(process.env.PUERTO);

console.log("Puerto configurado: " + puerto);
```

La línea `.config()` lee el archivo `.env` y mete cada variable en `process.env`, como si las hubieras definido en la terminal. Debe ir al principio de tu programa, antes de cualquier código que use las variables.

Y ahora la regla de seguridad más importante de esta unidad: el archivo `.env` NUNCA se sube al repositorio. Se agrega al `.gitignore` (el archivo que le dice a Git qué ignorar) desde el primer día:

```
node_modules
.env
```

En su lugar, se comparte un archivo `.env.example` (o `.env.ejemplo`) con los NOMBRES de las variables pero sin los valores reales:

```
PASSWORD_BD=
API_KEY=
PUERTO=3000
```

Así, quien clone tu proyecto sabe qué variables necesita configurar, pero tus secretos nunca salen de tu máquina. Este flujo (`.env` local + `.env.example` compartido + `.env` en `.gitignore`) es el estándar de la industria. Apréndelo ahora y úsalo en cada proyecto desde hoy.

process.cwd: dónde estoy parado

`process.cwd()` (current working directory) te dice desde qué carpeta se ejecutó el comando `node`. Ojo: no es lo mismo que la carpeta donde está tu archivo. Si estás en `/home/esteban` y corres `node proyectos/app.js`, `process.cwd()` devuelve `/home/esteban` (donde estás parado), mientras que `__dirname` dentro de `app.js` devuelve `/home/esteban/proyectos` (donde vive el archivo).

¿Cuándo importa la diferencia? Cuando tu programa lee archivos con rutas relativas. Si en `app.js` escribes `fs.readFileSync("./datos.txt")`, esa ruta se resuelve contra `process.cwd()`, no contra `__dirname`. Entonces el programa funciona si lo corres desde `proyectos/` pero falla si lo corres desde otro lado. La solución profesional: construir las rutas a partir de `__dirname` (la ubicación del archivo, que no cambia) en vez de confiar en desde dónde te paraste para correr el comando. Volveremos a esto en la sección de `__dirname`.

Otros miembros útiles de `process` que debes conocer aunque no los uses todos los días:

- `process.version`: la versión de Node.js que está corriendo ("v20.11.0"). Útil para verificar compatibilidad.

- `process.platform`: el sistema operativo ("linux", "darwin" para Mac, "win32" para Windows). Útil cuando tu código debe adaptarse al sistema.
- `process.uptime()`: cuántos segundos lleva corriendo tu programa. Útil para monitoreo.
- `process.memoryUsage()`: cuánta memoria está usando tu programa. Útil para detectar fugas de memoria.
- `process.on("exit", ...)`: te permite ejecutar código justo antes de que el programa termine (cerrar conexiones, guardar estado). Es un evento, tema que veremos a fondo más adelante, pero que sepas que existe.

__dirname y __filename: saber dónde vive tu archivo

En CommonJS tienes dos variables gratuitas que te dicen la ubicación de tu archivo: `__dirname` es la carpeta donde vive el archivo actual, y `__filename` es la ruta completa del archivo actual. Si tu archivo está en `/home/esteban/proyectos/app.js`, entonces `__dirname` vale `"/home/esteban/proyectos"` y `__filename` vale `"/home/esteban/proyectos/app.js"`.

¿Para qué sirven? Para construir rutas a archivos que están junto a tu código, sin importar desde dónde se corra el programa. El caso típico: tu programa necesita leer un archivo de datos que vive en la misma carpeta:

```
const fs = require("fs");
const path = require("path");

// Ruta al archivo datos.txt que está junto a este script
const rutaDatos = path.join(__dirname, "datos.txt");
const contenido = fs.readFileSync(rutaDatos, "utf8");
console.log(contenido);
```

`path.join` une pedazos de ruta de forma segura (usa el separador correcto según el sistema operativo). Con `path.join(__dirname, "datos.txt")` obtienes la ruta absoluta al archivo, funcione donde funcione el programa. Sin `__dirname`, dependerías de `process.cwd()` y tu programa se rompería según desde dónde lo invocaran. Regla práctica: para archivos que acompañan a tu código, usa `__dirname`; para archivos que el usuario te indica (como un archivo a procesar pasado por `argv`), usa la ruta tal cual o resuélvela contra `process.cwd()`.

import.meta.url: el __dirname de los ES Modules

En ESM no existen __dirname ni __filename (¿recuerdas? no hay función envolvente que te los pase). En su lugar tienes import.meta.url, que es la URL del archivo actual, algo como "file:///home/esteban/proyectos/app.js". Para convertirla en una ruta de carpeta usable, se combina con dos utilidades del módulo url:

```
import { fileURLToPath } from "url";
import { dirname, join } from "path";
import { readFileSync } from "fs";

const __filename = fileURLToPath(import.meta.url);
const __dirname = dirname(__filename);

const rutaDatos = join(__dirname, "datos.txt");
const contenido = readFileSync(rutaDatos, "utf8");
console.log(contenido);
```

Lo que hace este ritual: fileURLToPath convierte la URL "file:///..." en una ruta normal del sistema; dirname le quita el nombre del archivo y deja solo la carpeta. El resultado es idéntico al __dirname de CommonJS. Es más verboso, sí, pero funciona igual y es el estándar.

Como este ritual aparece en casi todo proyecto ESM que lee archivos, muchos programadores lo ponen una vez en un módulo utilitario y lo reutilizan. Ojo con un detalle: import.meta solo existe en archivos ESM; si lo usas en un archivo CommonJS, obtienes un error. Es otra de esas diferencias entre mundos que ya no te van a sorprender.

Otros globales: console, temporizadores, Buffer y global

Además de process, Node.js te regala otros globales que ya has usado sin pensar mucho en ellos. Vamos a presentarlos formalmente:

console: lo usas desde la Unidad 1, pero tiene más que log. console.error() imprime en la salida de errores (útil para separar mensajes normales de errores cuando rediriges salidas). console.warn() para advertencias. console.table() muestra arreglos y objetos como tablas bonitas en la terminal. console.time("etiqueta") y console.timeEnd("etiqueta") miden cuánto tarda un bloque de código: una herramienta de medición instantánea sin instalar nada.

```

console.time("bucle");
let total = 0;
for (let i = 0; i < 10000000; i++) {
  total = total + i;
}
console.timeEnd("bucle"); // bucle: 3.456ms (el tiempo real varía)

```

setTimeout, setInterval, clearTimeout, clearInterval: los temporizadores que conoces del navegador también existen en Node.js. setTimeout ejecuta una función una vez después de X milisegundos; setInterval la repite cada X milisegundos; los clear cancelan el temporizador. Son globales, no necesitas importarlos.

```

console.log("Empiezo");

setTimeout(function () {
  console.log("Esto sale 2 segundos después");
}, 2000);

let contador = 0;
const intervalo = setInterval(function () {
  contador = contador + 1;
  console.log("Segundo " + contador);
  if (contador === 3) {
    clearInterval(intervalo);
    console.log("Intervalo detenido");
  }
}, 1000);

```

Buffer: sirve para manejar datos binarios (archivos, datos de red) que no son texto. Cuando lees un archivo sin especificar codificación, fs te devuelve un Buffer, no un string. Puedes convertirlo: buffer.toString("utf8"). Y puedes crear uno desde texto: Buffer.from("Hola"). No necesitas entenderlo a fondo ahora; solo registra que existe y que aparece cuando trabajas con archivos y red, temas de próximas unidades.

global: es el objeto global de Node.js, el equivalente a window en el navegador. Todo lo que cuelgues de global se vuelve accesible desde cualquier archivo sin importarlo: global.miVariable = 42 y luego en otro archivo console.log(miVariable) funciona. Suena conveniente, pero es una mala práctica en código serio: es el regreso a la olla compartida de la fonda, donde todo se mezcla y nadie sabe quién modificó qué. Los globales explícitos (process, console, setTimeout) están bien porque Node.js los controla; crear los tuyos propios con global casi siempre es una señal de diseño pobre. Si sientes la tentación de usar global

para compartir algo entre archivos, lo correcto es crear un módulo y compartirlo con `require` o `import`. Menciónalo en entrevistas como lo que es: un mecanismo que existe pero que se evita.

Errores comunes y sus soluciones: el botiquín de la unidad

Esta sección es oro puro: los errores que vas a ver una y otra vez, con su causa y su arreglo. Cuando algo falle, ven aquí primero.

Error 1: "Cannot find module './matemáticas'". Causas posibles, en orden de probabilidad: olvidaste el `./` y escribiste `require("matemáticas")` (Node.js lo buscó en `node_modules` y no lo encontró); el archivo no existe o está en otra carpeta (revisa que `app.js` y `matemáticas.js` estén donde crees); error de mayúsculas/minúsculas en el nombre (en Linux "Matemáticas.js" y "matemáticas.js" son archivos distintos); en ESM, olvidaste la extensión `.js` en el `import`. Solución: verifica la ruta carácter por carácter, confirma que el archivo existe con `ls`, y recuerda: `./` para tus archivos, extensión obligatoria en ESM.

Error 2: "___dirname is not defined" (o "___filename is not defined"). Causa: estás en un archivo ESM (con "type": "module" o extensión `.mjs`) y estas variables solo existen en CommonJS. Solución: usa el ritual de `import.meta.url` con `fileURLToPath` y `dirname` que vimos arriba. Si el error aparece de repente en un proyecto que funcionaba, revisa si alguien agregó "type": "module" al `package.json`: ese cambio convierte todos los `.js` a ESM de golpe.

Error 3: "require is not defined". Causa: estás usando `require` en un archivo ESM. Solución: o conviertes tu código a `import`, o renombras el archivo a `.cjs` para que se trate como CommonJS, o quitas el "type": "module" si el proyecto en realidad es CommonJS. Decide en qué mundo vive cada archivo y sé consistente.

Error 4: "Cannot use import statement outside a module". Causa: estás usando `import` en un archivo que Node.js trata como CommonJS (un `.js` sin "type": "module"). Solución: agrega "type": "module" al `package.json` o renombra el archivo a `.mjs`. Es el espejo del error 3: cada sistema exige su configuración.

Error 5: mezclar `require` e `import` en el mismo archivo. En un archivo CommonJS no puedes usar `import` (salvo el `import()` dinámico, tema avanzado); en un archivo ESM no puedes usar `require`. Si necesitas funcionalidad de ambos mundos, la vía permitida es: desde ESM, importa módulos CommonJS con `import` (Node.js traduce automáticamente). Desde CommonJS hacia ESM no hay vía directa con `require`. Si te encuentras atorado aquí, casi siempre la respuesta es decidir de qué lado está tu archivo y usar solo la sintaxis de ese lado.

Error 6: "exports is not defined" o exportaciones vacías inesperadas. Causa: reasignaste `exports` directamente (`exports = {...}`) en vez de `module.exports`, rompiendo la referencia como

vimos en la sección de `module.exports` vs `exports`. Solución: usa siempre `module.exports = ...` para exportaciones completas.

Error 7: variables de entorno undefined. Causas: olvidaste llamar a `require("dotenv").config()` antes de usarlas; el archivo `.env` no está en la carpeta desde donde corres el programa (dotenv lo busca en `process.cwd()`, no en `__dirname`); error de dedo en el nombre (`PASSWORD_BD` vs `PASSWORD_DB`); o estás esperando que el `.env` funcione en producción cuando en producción las variables se configuran en el servidor, no con archivo. Solución: verifica el orden (`config()` primero), la ubicación del `.env` y el nombre exacto; y recuerda que dotenv es una comodidad para desarrollo, no el mecanismo de producción.

Error 8: el programa lee el archivo equivocado según desde dónde lo corres. Causa: usaste una ruta relativa cruda (`./datos.txt`) que se resuelve contra `process.cwd()` en vez de contra la ubicación del script. Solución: construye las rutas con `path.join(__dirname, "datos.txt")` (o el equivalente con `import.meta.url` en ESM).

Error 9: cambios en un módulo no se reflejan al volver a pedirlo. Causa: la caché de require. Si en un script largo haces require del mismo archivo dos veces esperando ejecuciones frescas, la segunda vez obtienes la caché. (En scripts normales esto rara vez es problema; aparece más en herramientas de desarrollo con recarga.) Solución conceptual: diseña tus módulos para que el trabajo ocurra cuando llamas a sus funciones, no cuando se cargan; así la caché deja de importar.

Error 10: "SyntaxError: Unexpected token 'export'". Causa: usaste `export` en un archivo CommonJS. Solución: decide el sistema del archivo (ver errores 3 y 4) y usa solo su sintaxis.

La estrategia general de depuración para esta unidad, en orden: 1) lee el mensaje de error completo, que casi siempre nombra el problema; 2) pregúntate "¿este archivo es CommonJS o ESM?" y verifica que la sintaxis corresponda; 3) revisa las rutas (`./`, extensiones, mayúsculas); 4) si es variable de entorno, verifica `dotenv.config()` y el `.env`; 5) si nada cuadra, haz un script mínimo de tres líneas que reproduzca el problema aislado: el 90% de las veces el bug se revela solo al simplificar.

Ejemplos paso a paso

Teoría asimilada, manos a la obra. Cada ejemplo es un mini-proyecto completo que puedes copiar, correr y modificar. Te recomiendo crear una carpeta práctica-u2 y trabajar ahí.

Ejemplo 1: calculadora modular con require (CommonJS)

Vamos a construir la calculadora que aparece en la clase 4, pero completa y bien organizada: un módulo con las operaciones y un programa principal que las usa. Tres archivos en la misma carpeta.

Archivo matemáticas.js: el módulo que ofrece las operaciones.

```
function suma(a, b) {
  return a + b;
}

function resta(a, b) {
  return a - b;
}

function multiplica(a, b) {
  return a * b;
}

function divide(a, b) {
  if (b === 0) {
    throw new Error("No se puede dividir entre cero");
  }
  return a / b;
}

module.exports = { suma, resta, multiplica, divide };
```

Observa: cuatro funciones puras (reciben datos, devuelven resultados, sin efectos secundarios) y una sola línea de exportación al final. La función divide protege contra la división entre cero lanzando un error, que el programa principal puede atrapar. Este módulo no sabe nada de terminales ni de usuarios: solo matemáticas. Esa ignorancia es una virtud: mañana puedes usar este mismo archivo en un servidor web sin cambiar nada.

Archivo app.js: el programa principal que usa el módulo.

```
const { suma, resta, multiplica, divide } = require("./matematicas");

const a = 20;
const b = 4;

console.log("Suma: " + suma(a, b));
console.log("Resta: " + resta(a, b));
console.log("Multiplicación: " + multiplica(a, b));
console.log("División: " + divide(a, b));

try {
  console.log("División entre cero: " + divide(a, 0));
} catch (error) {
  console.log("Atrapado el error: " + error.message);
}
```

La primera línea es el require con desestructuración: pide el objeto exportado y extrae las cuatro funciones de un jalón. El try/catch demuestra cómo el programa principal maneja el error que lanza el módulo: el módulo decide qué es un error, el programa decide qué hacer con él. Esa división de responsabilidades es diseño profesional.

Para correrlo: node app.js en la carpeta. Resultado esperado: Suma: 24, Resta: 16, Multiplicación: 80, División: 5, y el mensaje del error atrapado.

Experimento para aprender: agrega una función potencia al módulo, expórtala y úsala en app.js. Luego rompe a propósito la exportación (cambia module.exports por exports = ...) y observa cómo app.js falla: ver el error con tus propios ojos graba la lección mejor que leerla diez veces.

Ejemplo 2: la misma calculadora con import/export (ESM)

Ahora la misma calculadora en el sistema moderno. Necesitas un package.json con "type": "module" en la carpeta (créalo con el contenido mínimo que vimos antes). Los archivos cambian así:

Archivo matemáticas.js:

```

export function suma(a, b) {
  return a + b;
}

export function resta(a, b) {
  return a - b;
}

export function multiplica(a, b) {
  return a * b;
}

export function divide(a, b) {
  if (b === 0) {
    throw new Error("No se puede dividir entre cero");
  }
  return a / b;
}

```

Cada función se exporta en su propia declaración: no hay objeto ni línea final de exportación. Es más directo: "esta función es pública, punto".

Archivo app.js:

```

import { suma, resta, multiplica, divide } from "../matematicas.js";

const a = 20;
const b = 4;

console.log("Suma: " + suma(a, b));
console.log("Resta: " + resta(a, b));
console.log("Multiplicación: " + multiplica(a, b));
console.log("División: " + divide(a, b));

try {
  console.log("División entre cero: " + divide(a, 0));
} catch (error) {
  console.log("Atrapado el error: " + error.message);
}

```

Idéntico al anterior salvo dos detalles: `import` en vez de `require`, y `./matemáticas.js` con extensión obligatoria. Todo lo demás (la lógica, el `try/catch`) es igual, porque la lógica de tu programa no depende del sistema de módulos: solo cambia la fontanería que conecta los archivos.

Compara los dos ejemplos lado a lado y nota qué cambió y qué no: cambió la sintaxis de conexión, no cambió ni una línea de lógica matemática. Esa es la lección profunda: los sistemas de módulos son intercambiables en cuanto a lo que tu programa HACE; difieren en cómo se ESCRIBE la conexión. Por eso ser bilingüe es tan valioso: puedes leer y escribir ambos sin reaprender a programar.

Experimento: quita la extensión `.js` del `import` y corre el programa para ver el error con tus propios ojos. Luego vuelve a ponerla. Después, quita el `"type": "module"` del `package.json` y observa el error `"Cannot use import statement outside a module"`. Cada error que provocas a propósito es un error que ya no te va a asustar en un proyecto real.

Ejemplo 3: CLI que saluda con `process.argv`

Vamos a construir una herramienta de terminal de verdad: un programa que genera saludos personalizados según los argumentos que reciba. Lo haremos modular: un módulo con la lógica de saludo y un programa principal que lee la terminal.

Archivo `saludos.js` (CommonJS, para practicar `require`):

```
function generarSaludo(nombre, idioma) {
  const saludos = {
    es: "Hola",
    en: "Hello",
    fr: "Bonjour",
    pt: "Olá"
  };

  const saludo = saludos[idioma] || saludos.es;
  return saludo + ", " + nombre + ". Bienvenido a Lucas EGA Academy.";
}

module.exports = { generarSaludo };
```

El truco `saludos[idioma] || saludos.es`: si el idioma no existe en el objeto, usa español por defecto. Es un patrón elegante para valores por defecto sin `ifs`.

Archivo `cli.js`:

```
const { generarSaludo } = require("./saludos");

const nombre = process.argv[2];
const idioma = process.argv[3] || "es";

if (!nombre) {
  console.log("Uso: node cli.js <nombre> [idioma]");
  console.log("Idiomas disponibles: es, en, fr, pt");
  process.exit(1);
}

console.log(generarSaludo(nombre, idioma));
```

Pruebas:

```
node cli.js Esteban
// Hola, Esteban. Bienvenido a Lucas EGA Academy.

node cli.js Esteban en
// Hello, Esteban. Bienvenido a Lucas EGA Academy.

node cli.js
// Uso: node cli.js <nombre> [idioma]
// Idiomas disponibles: es, en, fr, pt
```

Fíjate en los detalles profesionales: el idioma es opcional (|| "es" le da valor por defecto), el nombre es obligatorio (sin él, el programa explica cómo usarse y sale con código 1). El módulo saludos.js no sabe nada de la terminal: recibe datos y devuelve texto. El cli.js no sabe nada de idiomas: lee argumentos y los pasa. Cada pieza en su lugar, como los cocineros de la fonda.

Experimento: agrega un idioma más al objeto (por ejemplo, "de" para alemán con "Hallo"). Luego haz que el programa acepte un tercer argumento opcional con el nombre de la academia, para no tenerlo fijo en el código. Estás practicando exactamente lo que hacen las CLIs reales: argumentos obligatorios, opcionales y valores por defecto.

Ejemplo 4: script que lee variables de entorno

Vamos a simular la conexión a un servicio externo (como una base de datos) usando variables de entorno y dotenv. Es el patrón que vas a usar en cada proyecto serio a partir de hoy.

Paso 1: instala dotenv en tu carpeta de práctica (necesitas un package.json; créalo con npm init -y si no lo tienes):

```
npm install dotenv
```

Paso 2: crea el archivo .env (recuerda: este archivo NO se comparte, va al .gitignore):

```
DB_USUARIO=esteban
DB_PASSWORD=claveSecreta123
DB_SERVIDOR=localhost
DB_PUERTO=5432
```

Paso 3: crea .env.example para documentar qué necesita el proyecto (este SÍ se comparte):

```
DB_USUARIO=
DB_PASSWORD=
DB_SERVIDOR=
DB_PUERTO=
```

Paso 4: el programa, conectar.js:

```
require("dotenv").config();

const usuario = process.env.DB_USUARIO;
const password = process.env.DB_PASSWORD;
const servidor = process.env.DB_SERVIDOR;
const puerto = Number(process.env.DB_PUERTO);

const faltantes = [];
if (!usuario) faltantes.push("DB_USUARIO");
if (!password) faltantes.push("DB_PASSWORD");
if (!servidor) faltantes.push("DB_SERVIDOR");
if (!puerto) faltantes.push("DB_PUERTO");

if (faltantes.length > 0) {
  console.log("Faltan variables de entorno: " + faltantes.join(", "));
  console.log("Copia .env.example a .env y llena los valores.");
  process.exit(1);
}

console.log("Conectando a " + servidor + ":" + puerto + " como " + usuario + "...");
console.log("¡Conexión exitosa! (simulada)");
console.log("Nota: la contraseña nunca se imprimió ni se guardó en el código.");
```


Este ejemplo junta varias lecciones: `dotenv.config()` al inicio, lectura de `process.env`, conversión del puerto a número, validación de que nada falte (con mensaje útil que dice exactamente qué falta y cómo arreglarlo), y la buena práctica de NO imprimir la contraseña ni en los logs. En un programa real, aquí iría el código que abre la conexión a la base de datos usando esas credenciales; la estructura sería idéntica.

Prueba romperlo a propósito: renombra el `.env` a `.env.apagado` y corre el programa para ver el mensaje de variables faltantes. Luego regrésale el nombre. Así compruebas que tu validación funciona antes de necesitarla de verdad, que es exactamente cuando quieres descubrir que no funciona: nunca en producción.

¿Sabías que...? El nombre "dotenv" viene de "dot env", por el archivo `.env` (los archivos que empiezan con punto son ocultos en Linux y Mac). La convención del `.env` nació en el mundo Ruby hace más de una década y hoy es estándar en casi todos los lenguajes. Cuando veas un `.env` en un proyecto de Python, PHP o Go, ya sabes qué es y cómo funciona: los buenos patrones cruzan fronteras entre lenguajes.

Práctica

La regla de siempre: intenta cada ejercicio por tu cuenta antes de mirar la solución. Si te atorras más de quince minutos, estudia la solución, ciérrala y reescríbela tú sin mirar. El músculo se entrena con esfuerzo.

Ejercicio 1: biblioteca de textos con CommonJS

Enunciado: crea un módulo `textos.js` que exporte tres funciones: `contarPalabras(texto)` que devuelve cuántas palabras tiene un texto, `invertir(texto)` que devuelve el texto al revés, y `gritar(texto)` que devuelve el texto en mayúsculas con tres signos de admiración al final. Luego crea `app.js` que use las tres funciones con `require` y muestre los resultados para el texto "hola mundo desde node".

```
// textos.js - Solución
function contarPalabras(texto) {
  const palabras = texto.trim().split(" ");
  return palabras.length;
}

function invertir(texto) {
  return texto.split("").reverse().join("");
}

function gritar(texto) {
  return texto.toUpperCase() + "!!!";
}

module.exports = { contarPalabras, invertir, gritar };
```

```
// app.js - Solución
const { contarPalabras, invertir, gritar } = require("../textos");

const frase = "hola mundo desde node";

console.log("Palabras: " + contarPalabras(frase)); // 4
console.log("Invertido: " + invertir(frase)); // edon edsed odnum aloh
console.log("Gritando: " + gritar(frase)); // HOLA MUNDO DESDE NODE!!!
```

Explicación paso a paso: primero diseñamos el módulo pensando solo en la lógica, sin preocuparnos por quién lo va a usar. `contarPalabras` limpia espacios con `trim()`, separa por espacios con `split(" ")` y cuenta los pedazos. `invertir` convierte el texto en arreglo de letras, lo voltea y lo vuelve a unir: el truco clásico `split/reverse/join`. `gritar` usa `toUpperCase()` y concatena los signos. Al final, una sola línea `module.exports` entrega las tres funciones en un objeto. En `app.js`, el `require` con desestructuración extrae las tres de un jalón y las probamos con una frase real. Nota cómo el módulo no imprime nada: solo calcula y devuelve. Imprimir es trabajo del programa principal. Esa separación (módulo calcula, programa presenta) es la que permite reutilizar `textos.js` mañana en un servidor web sin tocarlo.

Ejercicio 2: la misma biblioteca con ES Modules

Enunciado: convierte el ejercicio anterior a ESM. Crea una carpeta nueva con un `package.json` que tenga `"type": "module"`, reescribe `textos.js` con `export` y `app.js` con `import` (recuerda la extensión `.js` en la ruta). Verifica que el resultado sea idéntico.

```
// package.json - Solución
{
  "name": "biblioteca-esm",
  "version": "1.0.0",
  "type": "module"
}
```

```
// textos.js - Solución
export function contarPalabras(texto) {
  const palabras = texto.trim().split(" ");
  return palabras.length;
}

export function invertir(texto) {
  return texto.split("").reverse().join("");
}

export function gritar(texto) {
  return texto.toUpperCase() + "!!!";
}
```

```
// app.js - Solución
import { contarPalabras, invertir, gritar } from "./textos.js";

const frase = "hola mundo desde node";

console.log("Palabras: " + contarPalabras(frase));
console.log("Invertido: " + invertir(frase));
console.log("Gritando: " + gritar(frase));
```

Explicación paso a paso: el package.json con "type": "module" le dice a Node.js que los .js de esta carpeta son ES Modules. En textos.js, cada función lleva su export pegado: ya no hay objeto ni línea final. En app.js, el import pide los mismos tres nombres, pero la ruta ahora es "./textos.js" con extensión obligatoria: sin ella, Node.js en modo ESM no encuentra el archivo. La lógica de las funciones no cambió ni una coma: solo la fontanería de conexión. Si al correr obtienes "Cannot find module", revisa primero la extensión; si obtienes "Cannot use import statement outside a module", revisa el package.json. Estos dos errores son el 90% de los problemas al empezar con ESM, y ahora ya sabes diagnosticarlos en segundos.

Ejercicio 3: CLI de propinas con validación completa

Enunciado: crea propina.js, una herramienta de terminal que recibe dos argumentos: el total de la cuenta y el porcentaje de propina. Debe validar que ambos existan y sean números válidos, calcular la propina y el total a pagar, y mostrar el desglose. Si los argumentos son inválidos, debe explicar el uso correcto y salir con código 1. Usa CommonJS y modulariza: la lógica de cálculo va en un módulo calculos.js.

```
// calculos.js - Solución
function calcularPropina(cuenta, porcentaje) {
  const propina = cuenta * (porcentaje / 100);
  const total = cuenta + propina;
  return { propina, total };
}

module.exports = { calcularPropina };
```

```
// propina.js - Solución
const { calcularPropina } = require("./calculos");

const cuentaTexto = process.argv[2];
const porcentajeTexto = process.argv[3];

if (!cuentaTexto || !porcentajeTexto) {
  console.log("Uso: node propina.js <cuenta> <porcentaje>");
  console.log("Ejemplo: node propina.js 350 15");
  process.exit(1);
}

const cuenta = Number(cuentaTexto);
const porcentaje = Number(porcentajeTexto);

if (Number.isNaN(cuenta) || Number.isNaN(porcentaje)) {
  console.log("Error: la cuenta y el porcentaje deben ser números.");
  process.exit(1);
}

if (cuenta < 0 || porcentaje < 0) {
  console.log("Error: no se aceptan valores negativos.");
  process.exit(1);
}

const { propina, total } = calcularPropina(cuenta, porcentaje);

console.log("Cuenta: $" + cuenta);
console.log("Propina (" + porcentaje + "%): $" + propina.toFixed(2));
console.log("Total a pagar: $" + total.toFixed(2));
```

Explicación paso a paso: el módulo `calculos.js` es puro cálculo: recibe números, devuelve un objeto con `propina` y `total`. No sabe nada de la terminal. El programa `propina.js` hace el trabajo sucio de CLI en tres etapas de validación, en orden: primero verifica que los argumentos existan (¿me dieron los dos datos?); luego los convierte a número y verifica que sean números válidos (¿son de verdad números?); luego verifica que no sean negativos (¿tienen sentido?). Solo si las tres validaciones pasan, llama a la función del módulo y muestra el desglose. Cada validación fallida explica el problema y sale con código 1. El `toFixed(2)` redondea a dos decimales para mostrar dinero cómo se debe. Prueba estos casos: `node propina.js 350 15` (caso feliz), `node propina.js 350` (falta argumento), `node propina.js hola 15` (no es número), `node propina.js 350 -5` (negativo). Una CLI bien validada es la diferencia entre una herramienta profesional y un script frágil.

Ejercicio 4: lector de configuración con variables de entorno

Enunciado: crea config.js, un módulo que lee la configuración de una aplicación desde variables de entorno usando dotenv: APP_NOMBRE (texto), APP_PUERTO (número, por defecto 3000 si no se define), APP_DEBUG (booleano: true solo si vale exactamente "true"). El módulo debe exportar un objeto config ya validado y con los tipos correctos. Luego crea mostrar.js que importe el módulo e imprima la configuración. Incluye un .env y un .env.example.

Archivo .env - Solución

```
APP_NOMBRE=MisSuperApp
APP_PUERTO=8080
APP_DEBUG=true
```

Archivo .env.example - Solución

```
APP_NOMBRE=
APP_PUERTO=3000
APP_DEBUG=false
```

```
// config.js - Solución (CommonJS)
require("dotenv").config();

const config = {
  nombre: process.env.APP_NOMBRE || "AppSinNombre",
  puerto: Number(process.env.APP_PUERTO) || 3000,
  debug: process.env.APP_DEBUG === "true"
};

if (Number.isNaN(Number(process.env.APP_PUERTO)) && process.env.APP_PUERTO) {
  console.log("Advertencia: APP_PUERTO no es un número válido, se usará 3000.");
}

module.exports = { config };
```

```
// mostrar.js - Solución
const { config } = require("./config");

console.log("Nombre: " + config.nombre);
console.log("Puerto: " + config.puerto + " (tipo: " + typeof config.puerto + ")");
console.log("Debug: " + config.debug + " (tipo: " + typeof config.debug + ")");

if (config.debug) {
  console.log("[DEBUG] Modo depuración activado: se mostrarán mensajes extra.");
}
```

Explicación paso a paso: el .env guarda los valores reales y el .env.example documenta qué variables existen (con valores de ejemplo, sin secretos reales). En config.js, dotenv.config() va primero que todo. Luego construimos el objeto config aplicando tres transformaciones típicas: nombre usa || para un valor por defecto si la variable no existe; puerto convierte a número con Number() y usa || 3000 como respaldo (ojo: si APP_PUERTO vale "0", Number("0") es 0 que es falsy y caería al 3000; para puertos esto es aceptable, pero en casos donde el 0 sea válido necesitarías una validación más cuidadosa); debug compara estrictamente con "true" porque todo en process.env es string y "false" como texto es truthy (¡trampa clásica!). Además agregamos una advertencia si el puerto no es numérico. El módulo exporta el objeto ya listo: quien lo pida recibe configuración validada y tipada, sin tener que repetir la lógica. mostrar.js solo imprime y demuestra que puerto es number de verdad y debug es boolean de verdad. Este patrón (módulo de configuración central) existe en prácticamente todo proyecto Node.js profesional: un solo lugar donde la configuración se lee, se valida y se tipa.

Ejercicio 5: encuentra y corrige los cinco errores

Enunciado: este proyecto ESM tiene cinco errores entre sus archivos. Encuéntralos todos, explica cada uno y escribe la versión corregida.

Archivos con errores:

```
// package.json
{
  "name": "proyecto-roto",
  "version": "1.0.0"
}
```

```
// utilidades.js
export function doble(n) {
  return n * 2;
}
exports.triple = function (n) {
  return n * 3;
};
```

```
// app.js
const { doble } = require("../utilidades");
import { triple } from "../utilidades";

console.log(doble(5));
console.log(triple(5));
console.log("Carpeta: " + __dirname);
```

// Solución: los cinco errores

// Error 1: el package.json no tiene "type": "module", pero app.js usa import.
 // Sin ese campo, los .js se tratan como CommonJS y el import falla.
 // Corrección: agregar "type": "module".

```
{
  "name": "proyecto-roto",
  "version": "1.0.0",
  "type": "module"
}
```

// Error 2: utilidades.js mezcla export (ESM) con exports.triple (CommonJS).
 // En un archivo ESM, exports no existe. Todo debe ser sintaxis ESM.
 // Corrección en utilidades.js:

```
export function doble(n) {
  return n * 2;
}
```

```
export function triple(n) {
  return n * 3;
}
```

// Error 3: app.js mezcla require (CommonJS) con import (ESM) en el mismo archivo.
 // Un archivo es de un solo mundo. Como el proyecto es ESM, todo va con import.
 // Corrección en app.js:

```
import { doble, triple } from "../utilidades.js";
```

// Error 4: el import no lleva la extensión .js en la ruta.


```
// En ESM la extensión es obligatoria: "./utilidades" falla.
// Corrección: from "./utilidades.js" (ya aplicado arriba).

// Error 5: __dirname no existe en ESM.
// Corrección: usar el ritual de import.meta.url.
import { fileURLToPath } from "url";
import { dirname } from "path";

const __filename = fileURLToPath(import.meta.url);
const __dirname = dirname(__filename);

// app.js completo y corregido:
import { doble, triple } from "./utilidades.js";
import { fileURLToPath } from "url";
import { dirname } from "path";

const __filename = fileURLToPath(import.meta.url);
const __dirname = dirname(__filename);

console.log(doble(5)); // 10
console.log(triple(5)); // 15
console.log("Carpeta: " + __dirname);
```

Explicación paso a paso: este ejercicio es un simulacro de depuración real, porque estos cinco errores son exactamente los que aparecen cuando alguien convierte un proyecto de require a import a medias. El error 1 es de configuración: el package.json manda sobre cómo se interpretan los archivos. El error 2 es de mezcla de sintaxis dentro de un mismo archivo: cada archivo habla un solo idioma. El error 3 es la misma mezcla pero en el archivo principal: require e import no conviven en el mismo archivo. El error 4 es el clásico de la extensión olvidada, el más frecuente de todos. El error 5 es el de __dirname en ESM, que se resuelve con import.meta.url. La estrategia para cazarlos en orden: primero la configuración (¿qué mundo es este proyecto?), luego la sintaxis de cada archivo (¿habla el idioma de su mundo?), luego los detalles (extensiones, globales disponibles). Si aplicas ese orden, ningún error de módulos te va a quitar más de cinco minutos.

Ejercicio 6: mini-reto integrador, generador de reportes

Enunciado: junta todo lo de la unidad en un mini-proyecto de tres archivos (usa el sistema de módulos que prefieras, pero sé consistente): 1) datos.js exporta un arreglo de ventas (objetos con producto y monto). 2) reporte.js importa los datos y exporta una función generarReporte() que devuelve un objeto con total de ventas, número de ventas y promedio. 3) cli.js lee de process.argv un umbral opcional (por defecto 0), genera el reporte, e imprime solo las ventas

mayores al umbral más el resumen. Además, cli.js debe leer de una variable de entorno REPORTE_TITULO el título del reporte (por defecto "Reporte de ventas").

```
// datos.js - Solución (ESM)
export const ventas = [
  { producto: "Laptop", monto: 12999 },
  { producto: "Mouse", monto: 299 },
  { producto: "Teclado", monto: 899 },
  { producto: "Monitor", monto: 3499 }
];
```

```
// reporte.js - Solución (ESM)
import { ventas } from "./datos.js";

export function generarReporte(umbral) {
  const filtradas = ventas.filter(function (v) {
    return v.monto > umbral;
  });

  let total = 0;
  for (const v of filtradas) {
    total = total + v.monto;
  }

  const promedio = filtradas.length > 0 ? total / filtradas.length : 0;

  return {
    ventas: filtradas,
    total: total,
    cantidad: filtradas.length,
    promedio: promedio
  };
}
```

```
// cli.js - Solución (ESM)
import "dotenv/config";
import { generarReporte } from "./reporte.js";

const titulo = process.env.REPORTE_TITULO || "Reporte de ventas";
const umbral = Number(process.argv[2]) || 0;

const reporte = generarReporte(umbral);

console.log("=== " + titulo + " ===");
console.log("Umbral: montos mayores a $" + umbral);
console.log("");

for (const v of reporte.ventas) {
  console.log("- " + v.producto + ": $" + v.monto);
}

console.log("");
console.log("Total: $" + reporte.total);
console.log("Cantidad de ventas: " + reporte.cantidad);
console.log("Promedio: $" + reporte.promedio.toFixed(2));
```

Explicación paso a paso: `datos.js` es el módulo de datos puros: solo exporta el arreglo, sin lógica. `reporte.js` importa los datos y exporta la función que los procesa: filtra por umbral, suma, cuenta y promedia, devolviendo todo en un objeto. Nota la protección contra división entre cero: si no hay ventas filtradas, el promedio es 0 en vez de NaN. `cli.js` es la capa de interacción: lee el título de la variable de entorno (con valor por defecto si no existe) y el umbral de los argumentos (convirtiendo a número, con 0 por defecto). Luego imprime el reporte formateado. Cada archivo tiene una sola responsabilidad: datos, lógica, interacción. Si mañana los datos vienen de una base de datos en vez de un arreglo, solo cambias `datos.js`. Si el reporte se muestra en una página web en vez de la terminal, solo cambias `cli.js`. Esa es la arquitectura modular en miniatura, y es exactamente así cómo se construyen las aplicaciones reales: capas separadas que se comunican por interfaces claras. Corre estas pruebas: `node cli.js` (todo), `node cli.js 1000` (solo montos grandes), `REPORTE_TITULO="Ventas Q3" node cli.js 500` (título personalizado). Cada prueba ejercita una capa distinta.

Resumen de la unidad

Llegamos al final de la unidad más estructural del curso. Vamos a recapitular lo esencial, eso que debe quedarte grabado aunque olvides los detalles.

Los módulos existen para dividir tu programa en piezas pequeñas, ordenadas y reutilizables, como piezas de LEGO o como las estaciones de una cocina de fonda. Cada archivo es un módulo: un cuarto cerrado donde tus variables son privadas, y lo único que sale al exterior es lo que pongas en la exportación.

CommonJS es el sistema clásico de Node.js: se exporta con `module.exports` y se importa con `require("./archivo")`. Nunca reasignes `exports` directamente: usa `module.exports` para exportaciones completas. `require` resuelve rutas relativas con `./`, módulos core sin `./` (`fs`, `path`, `http`, `os`), y paquetes buscándolos en `node_modules` de abajo hacia arriba. Los módulos se ejecutan una sola vez y su resultado se guarda en caché: todos los que piden el mismo módulo comparten el mismo objeto.

ES Modules es el sistema moderno y oficial de JavaScript: se exporta con `export` (nombradas o `default`) y se importa con `import { nombre } from "./archivo.js"`. La extensión `.js` en las rutas es obligatoria. Para activar ESM en tu proyecto, agrega `"type": "module"` al `package.json`. Las diferencias clave con CommonJS: `import` es estático, los valores importados son de solo lectura, no hay `__dirname` ni `require`, y la dirección de compatibilidad va de ESM hacia CommonJS, no al revés. En proyectos nuevos usa ESM; en proyectos viejos respeta el sistema que ya tienen; y sé bilingüe siempre, porque el mundo real mezcla ambos.

`process` es tu tablero de instrumentos: `process.argv` lee los argumentos de la terminal (recuerda que las posiciones 0 y 1 son del sistema y que todo llega como string), `process.env` lee las variables de entorno (donde viven tus secretos, nunca en el código), `process.exit` termina el programa con un código (0 éxito, 1 error), y `process.cwd` te dice desde dónde se corrió el comando. Con `dotenv` cargas las variables desde un archivo `.env` que jamás se sube al repositorio: el `.env` va al `.gitignore` y se comparte un `.env.example` sin valores reales.

`__dirname` y `__filename` te dicen dónde vive tu archivo en CommonJS; en ESM los reemplazas con el ritual de `import.meta.url`, `fileURLToPath` y `dirname`. Úsalos con `path.join` para construir rutas a archivos que acompañan tu código, sin depender de desde dónde se corra el programa. Otros globales que ya manejas: `console` (con sus variantes `error`, `table`, `time`), los temporizadores `setTimeout` y `setInterval`, `Buffer` para datos binarios, y `global`, que existe pero debes evitar.

Los errores de esta unidad casi siempre caen en cinco canastas: rutas mal escritas (falta el `./` o la extensión), archivo en el mundo equivocado (`import` sin `"type": "module"` o `require` en ESM), `__dirname` en ESM, mezcla de sintaxis en un mismo archivo, y variables de entorno `undefined` (falta `dotenv.config()` o el `.env` no está donde se corre el programa). Con la estrategia de depuración en orden (mensaje de error, mundo del archivo, rutas, entorno, script mínimo) los resuelves en minutos.

En la siguiente unidad vas a usar todo esto para construir herramientas de verdad: leer y escribir archivos con `fs`, manejar rutas con `path` y levantar tu primer servidor. Los módulos, `process` y los globales son los cimientos; lo que sigue es la casa. Y recuerda la lección silenciosa de esta unidad: el código profesional no es el que hace trucos impresionantes, es el que está ordenado en piezas claras, no quema secretos y se comporta bien cuando el usuario se equivoca. Eso es lo que acabas de aprender a construir.

UNIDAD 3: Archivos y rutas: fs y path

Hasta este punto del curso tus programas de Node.js han sido como una mente brillante pero sin memoria: hacen calculos, reciben datos, imprimen resultados en la pantalla y, cuando el programa termina, todo se olvida. La consola se cierra y lo que tu programa "aprendió" durante su ejecución se evapora como si nunca hubiera pasado. En esta unidad eso cambia para siempre. Vas a aprender a darle a tus programas algo que ningun programador serio puede darse el lujo de no dominar: memoria permanente.

Piensa en cualquier programa que uses todos los días. Un editor de texto guarda tus documentos en archivos. Un videojuego guarda tu partida para que la continúes mañana. Tu navegador guarda tus contraseñas, tu historial y tus favoritos. WhatsApp guarda tus mensajes para que estén ahí aunque cierres la aplicación. Ninguno de esos programas funciona de verdad sin la capacidad de leer y escribir datos en el disco de tu computadora. Un programa que no puede tocar archivos es un programa de juguete: sirve para aprender conceptos, pero no puede hacer trabajo real. En esta unidad tu programa deja de ser de juguete.

Los protagonistas de esta unidad son dos módulos del núcleo de Node.js: fs y path. El módulo fs (del inglés "file system", sistema de archivos) es la herramienta que le permite a tu código leer, escribir, crear, borrar y mover archivos y carpetas. Sin el, JavaScript viviría encerrado dentro del navegador o de la memoria temporal; con el, tu programa puede administrar documentos, generar reportes, guardar configuraciones y construir bases de datos sencillas. El módulo path, por su parte, es el especialista en direcciones: te ayuda a construir rutas de archivos que funcionen en cualquier sistema operativo, sin que tengas que pelear con las barras invertidas de Windows ni las barras normales de Linux y Mac. Juntos forman la base de casi todo lo que un programa de Node.js hace en el mundo real: servidores que leen plantillas HTML, aplicaciones que guardan registros de actividad, herramientas que procesan miles de archivos por lote y programas que descargan, copian y organizan información.

Esta unidad corresponde a las clases 7, 8 y 9 del curso. La clase 7 te presenta el módulo fs con su línea más famosa: `const fs = require("fs")`, y te muestra que significa que venga incluido en Node.js sin necesidad de instalar nada. La clase 8 se enfoca en las dos operaciones más importantes de todas: `fs.writeFile` para escribir y `fs.readFile` para leer, ambas con callbacks, y te enseña la idea central de que un programa que guarda datos en archivos es un programa con memoria permanente. La clase 9 te presenta el módulo path, con su joya más usada, `path.join(__dirname, "datos", "notas.txt")`, para unir rutas sin pelear con las barras y escribir código que funcione igual en Windows, Mac y Linux. En esta unidad del libro vamos a profundizar todo eso y mucho más, con la calma y el detalle que un libro permite y un video de clase no siempre alcanza a dar.

Antes de empezar, asegurate de tener Node.js instalado en tu computadora. Puedes comprobarlo abriendo una terminal y escribiendo `node -v`; si ves un número de versión como `v20.11.0` o parecido, estas listo. También necesitas un editor de texto o un entorno de desarrollo; cualquiera sirve, desde el Bloc de notas hasta Visual Studio Code. No necesitas instalar ningun paquete extra con npm para esta unidad: todo lo que vamos a usar viene incluido de fabrica en Node.js. Eso es una de las bellezas de `fs` y `path`: son módulos del nucleo, "core modules", lo que significa que están disponibles en el momento en que instalas Node.js, sin descargas adicionales, sin dependencias que se rompan, sin sorpresas.

Una advertencia importante antes de continuar, y tomatala en serio porque te va a ahorrar dolores de cabeza: en esta unidad vas a escribir código que crea, modifica y borra archivos reales en tu computadora. Eso es exactamente lo que queremos que aprendas, pero hazlo con cuidado. Trabaja siempre dentro de una carpeta de práctica que crees específicamente para este curso, por ejemplo una carpeta llamada `práctica-node` en tu escritorio o en tu carpeta de documentos. No ejecutes los ejemplos de borrado de archivos en carpetas donde tengas documentos importantes, fotos familiares o trabajos de la escuela. Los programas que borran archivos no preguntan dos veces, y un error de dedo en una ruta mal escrita puede borrar algo que no querias borrar. Con esa precaución clara, estas listo para empezar. La buena noticia es que, una vez que domines estas herramientas, vas a sentir que tu poder como programador se multiplica por diez: pasar de "mi programa muestra cosas en pantalla" a "mi programa crea documentos reales en mi disco" es uno de los saltos más satisfactorios que existen en la programación.

Explicación detallada

El módulo `fs`: el bibliotecario de tu programa

Imagina que tu programa es una persona trabajando en una oficina. Esa persona puede pensar, calcular y hablar (eso es lo que ya sabes hacer: variables, funciones, `console.log`), pero necesita un bibliotecario para ir al archivo, sacar documentos de los estantes, guardar papeles nuevos y tirar a la basura lo que ya no sirve. En Node.js, ese bibliotecario se llama `fs`. Cada vez que escribes `const fs = require("fs")` en tu código, estas contratando a ese bibliotecario y poniendolo a tu servicio.

Técnicamente, `fs` es un módulo del nucleo de Node.js, lo que significa tres cosas importantes. Primera: no necesitas instalarlo; viene incluido con cada instalación de Node.js en el planeta. Segunda: es estable y esta probado por millones de programadores; no es un experimento de alguien en internet, es infraestructura seria. Tercera: para usarlo solo necesitas una línea al

inicio de tu archivo JavaScript: `const fs = require("fs")`. Con esa línea, la variable `fs` se convierte en un objeto lleno de funciones listas para trabajar con archivos: `fs.readFile`, `fs.writeFile`, `fs.appendFile`, `fs.unlink`, `fs.mkdir`, `fs.readdir`, `fs.stat`, `fs.rename`, `fs.copyFile` y muchas más. En esta unidad vas a conocer a fondo las más importantes y vas a entender cuando usar cada una.

Hay algo que debes entender desde el primer momento sobre cómo trabaja `fs`, porque es la raíz de casi todos los errores de los principiantes: casi todas sus operaciones son asincrónicas. Esto significa que cuando le pides a `fs` que lea un archivo, tu programa no se queda congelado esperando a que el disco responda. El disco duro es lento comparado con el procesador; leer un archivo puede tomar milisegundos, que para una computadora es una eternidad. En lugar de quedarse parado mirando el disco, Node.js le dice al sistema operativo "oye, leeme este archivo y avisame cuando termines", y mientras tanto tu programa sigue haciendo otras cosas. Cuando la lectura termina, Node.js ejecuta una función que tu le diste de antemano: el `callback`. Este modelo es el corazón de Node.js y es exactamente lo mismo que aprendiste en la unidad de `callbacks` y promesas, solo que ahora aplicado a archivos reales.

Los tres sabores de fs: callbacks, promesas y sincrónico

El módulo `fs` ofrece tres formas distintas de hacer casi lo mismo, y elegir la correcta es una de las habilidades más importantes que vas a desarrollar en esta unidad. Piensa en estas tres formas como tres maneras de pedir comida en un restaurante, porque la analogía es perfecta y no se te va a olvidar.

La primera forma es la de `callbacks`, la forma clásica y original. Es como pedirle al mesero: "traeme una pizza, y cuando este lista, avisame en esta mesa". Tu no te quedas parado en la cocina esperando; te sientas, platicas con tus amigos, revisas tu teléfono, y cuando la pizza está lista el mesero llega a tu mesa (eso es el `callback` ejecutándose). En código se ve así:

```
const fs = require("fs");

fs.readFile("notas.txt", "utf8", (err, datos) => {
  if (err) {
    console.error("Algo salió mal:", err.message);
    return;
  }
  console.log("Contenido del archivo:");
  console.log(datos);
});

console.log("Esta línea se ejecuta ANTES de que termine la lectura");
```


Fíjate en la última línea: se imprime antes que el contenido del archivo, porque `fs.readFile` no bloquea el programa. El programa pide la lectura, sigue su camino, y cuando el sistema operativo termina, Node.js llama al callback con los resultados. El callback siempre recibe primero el parámetro `err` (el error, si hubo alguno) y después los datos. Ese orden, error primero y datos después, es una convención sagrada en Node.js que se repite en todos los callbacks del módulo `fs`. Si `err` tiene un valor, algo falló: el archivo no existe, no tienes permiso para leerlo, la ruta está mal escrita. Si `err` es `null`, todo salió bien y puedes usar los datos con confianza.

La segunda forma es la de promesas, disponible en el submódulo `fs/promises`. Es el mismo restaurante, pero ahora pides con una aplicación moderna: haces tu pedido y recibes una notificación en tu teléfono cuando está listo, sin que el mesero tenga que buscarte entre las mesas. En código, se ve así:

```
const fs = require("fs/promises");

async function leerNotas() {
  try {
    const datos = await fs.readFile("notas.txt", "utf8");
    console.log("Contenido del archivo:");
    console.log(datos);
  } catch (err) {
    console.error("Algo salió mal:", err.message);
  }
}

leerNotas();
```

Esta es la forma que la mayoría de los programadores profesionales prefieren hoy en día, porque el código se lee de arriba hacia abajo como una receta de cocina, sin anidar callbacks dentro de callbacks. La palabra clave `await` hace que la función espere la lectura sin bloquear el resto del programa, y el bloque `try/catch` atrapa los errores de forma limpia. Si ya dominas `async/await` de unidades anteriores, esta forma te va a parecer la más natural del mundo. Para este libro y para tus proyectos nuevos, te recomiendo que esta sea tu forma favorita.

La tercera forma es la sincrónica, con funciones cuyo nombre termina en `Sync`, como `fs.readFileSync` y `fs.writeFileSync`. Volviendo al restaurante: es como pararte frente a la cocina, cruzar los brazos y no moverte hasta que tu pizza esté lista. Nadie más puede pedir, nadie puede pasar, todo el restaurante se detiene por ti. En código:

```
const fs = require("fs");

try {
  const datos = fs.readFileSync("notas.txt", "utf8");
  console.log("Contenido del archivo:");
  console.log(datos);
} catch (err) {
  console.error("Algo salio mal:", err.message);
}
```

Esta forma es la más fácil de leer porque no hay callbacks ni promesas: la función devuelve los datos directamente, como una función normal. Pero tiene un costo enorme: mientras se ejecuta, TODO tu programa se congela. En un script pequeño que corre una vez y termina, eso no importa. Pero en un servidor web que atiende a cientos de usuarios al mismo tiempo, usar una función Sync es un crimen contra el rendimiento: cada usuario tendría que esperar a que el disco termine antes de que el servidor pueda atender al siguiente. La regla de oro es sencilla: usa las versiones Sync solo en scripts pequeños, herramientas de una sola ejecución o al inicio de tu programa para leer archivos de configuración; jamás las uses dentro de un servidor que atiende peticiones de usuarios.

Entonces, cuando usar cada sabor? La respuesta corta: callbacks cuando trabajes con código antiguo o librerías que los exigen; fs/promises con async/await para casi todo tu código nuevo, porque es lo más legible y lo más moderno; y las versiones Sync solo para scripts rápidos y sencillos donde bloquear el programa no le afecta a nadie. Si tienes duda, elige fs/promises. Es la apuesta segura.

Las operaciones esenciales, una por una

Ahora vamos a conocer a fondo las operaciones más importantes del módulo fs. Cada una tiene su personalidad, sus parámetros y sus trucos. Las veremos primero en su versión de callbacks, que es la forma clásica, y ten presente que casi todas existen también en versión de promesas (fs/promises) y en versión sincrónica (terminación Sync).

fs.writeFile es la operación de escritura por excelencia. Toma una ruta, unos datos y un callback, y escribe esos datos en el archivo. Si el archivo no existe, lo crea; si ya existe, lo sobrescribe por completo, borrando lo que había antes. Esa última parte es importantísima: writeFile no agrega texto al final, reemplaza todo el contenido. Es como volver a escribir una carta desde cero en lugar de añadirle un párrafo.

```
const fs = require("fs");

fs.writeFile("saludo.txt", "Hola, mundo desde Node.js", (err) => {
  if (err) {
    console.error("No se pudo escribir:", err.message);
    return;
  }
  console.log("Archivo creado con exito");
});
```

Observa que el callback de `writeFile` solo recibe `err`, porque no hay datos que devolver: o se escribió bien o falló. El segundo parámetro de `writeFile` también puede ser un objeto de opciones; el más común es especificar la codificación, aunque si pasas un string Node.js asume `utf8` por defecto en la mayoría de los casos. Más adelante hablaremos a fondo de la codificación porque es una de esas cosas que parecen detalles hasta que te muerden.

`fs.readFile` es la contraparte: lee el contenido completo de un archivo y te lo entrega en el callback. Su segundo parámetro es la codificación, y aquí si te recomiendo escribirla siempre: `"utf8"`. Si la omites, `readFile` no te devuelve un texto sino un `Buffer`, que es un objeto con los bytes crudos del archivo. Los `Buffers` son útiles cuando trabajas con imágenes, videos o archivos binarios, pero si esperabas un texto y recibes un `Buffer`, tu `console.log` va a mostrar algo raro como `<Buffer 48 6f 6c 61...>` en lugar de `"Hola"`. Regla práctica: si el archivo es texto, pasa `"utf8"`; si es binario, omite la codificación y trabaja con el `Buffer`.

`fs.appendFile` es la prima amable de `writeFile`: en lugar de reemplazar el contenido, agrega los datos al final del archivo. Si el archivo no existe, lo crea. Es perfecta para registros de actividad (logs), donde cada línea nueva es un evento que quieres conservar junto a los anteriores. Imagina un diario: `writeFile` sería arrancar la página y escribir una nueva cada vez; `appendFile` es escribir en la siguiente línea disponible, conservando todo lo anterior.

```
const fs = require("fs");

fs.appendFile("diario.txt", "Hoy aprendi a usar fs.appendFile.\n", (err) => {
  if (err) {
    console.error("No se pudo agregar:", err.message);
    return;
  }
  console.log("Linea agregada al diario");
});
```

El "\n" al final es el salto de línea; sin el, todas tus entradas quedarían pegadas en una sola línea interminable. Es un detalle pequeño que marca la diferencia entre un archivo legible y un desastre.

fs.unlink borra un archivo. El nombre viene de la jerga de los sistemas Unix, donde "unlink" significa desvincular el nombre del archivo de sus datos en el disco. No hay papelera de reciclaje aquí: cuando unlink tiene éxito, el archivo desaparece para siempre. Usalo con respeto y, como te advertí en la introducción, práctica solo dentro de tu carpeta de práctica.

fs.mkdir crea una carpeta (un directorio). Por defecto crea un solo nivel: si intentas crear "a/b/c" y la carpeta "a" no existe, falla. Pero si le pasas la opción { recursive: true }, crea toda la cadena de carpetas necesarias de un solo golpe, como abrir camino con machete. Esta opción es una de las más usadas en la vida real.

```
const fs = require("fs");

fs.mkdir("proyectos/mi-app/datos", { recursive: true }, (err) => {
  if (err) {
    console.error("No se pudo crear la carpeta:", err.message);
    return;
  }
  console.log("Carpetas creadas");
});
```

fs.readdir lee el contenido de una carpeta y te devuelve un arreglo con los nombres de los archivos y subcarpetas que contiene. No entra a las subcarpetas, solo lista el nivel que le pides, como abrir un cajón y ver que hay dentro sin revisar cada caja por dentro. Si le pasas la opción { withFileTypes: true }, en lugar de simples nombres te devuelve objetos Dirent que te dicen si cada entrada es un archivo o una carpeta, lo cual es muy útil para recorrer estructuras completas.

fs.stat te da la ficha técnica de un archivo o carpeta: su tamaño en bytes, cuando fue creado, cuando fue modificado por última vez, y si es archivo o directorio. Es como pedir el expediente de alguien en una oficina: no te da el contenido, te da sus datos. Con stat.size sabes cuánto pesa un archivo, con stat.isFile() confirmas que es un archivo y con stat.isDirectory() confirmas que es una carpeta. Esta función es indispensable cuando tu programa necesita tomar decisiones basadas en las características de los archivos, como "procesa solo los archivos que pesen menos de un megabyte" o "ignora las carpetas".

```
const fs = require("fs");

fs.stat("notas.txt", (err, stats) => {
  if (err) {
    console.error("No se pudo obtener informacion:", err.message);
    return;
  }
  console.log("Tamano en bytes:", stats.size);
  console.log("Es archivo?", stats.isFile());
  console.log("Es carpeta?", stats.isDirectory());
  console.log("Ultima modificacion:", stats.mtime);
});
```

`fs.rename` cambia el nombre de un archivo o lo mueve de lugar. Aunque se llama "rename", en realidad hace las dos cosas: si la ruta de destino está en otra carpeta, el archivo se mueve. Es la forma estandar de mover archivos en Node.js. Ojo: si en el destino ya existe un archivo con ese nombre, lo reemplaza sin preguntar.

`fs.copyFile` copia un archivo de una ruta a otra. Es la fotocopidora del módulo: el original se queda intacto y obtienes un duplicado en el destino. Acepta una bandera opcional, `fs.constants.COPYFILE_EXCL`, que hace que la copia falle si el destino ya existe, protegiendote de sobrescribir algo por accidente.

`fs.existsSync` es un caso especial: solo existe en versión sincrónica y te devuelve `true` o `false` según el archivo exista o no. Suena útil, y lo es para chequeos rápidos, pero tiene una trampa famosa: entre el momento en que preguntas "existe?" y el momento en que intentas usar el archivo, algo pudo haber cambiado (otro programa pudo borrarlo). Los profesionales prefieren intentar la operación directamente y manejar el error si falla, en lugar de preguntar antes. Aun así, para scripts sencillos y validaciones rápidas, `existsSync` es perfectamente aceptable.

La codificación utf8: el idioma en que se escriben los archivos

Vamos a hablar de un tema que parece aburrido hasta que te arruina un viernes por la noche: la codificación de caracteres. Cuando guardas el texto "Hola" en un archivo, la computadora no guarda letras; guarda números, porque la computadora solo entiende números. La codificación es el diccionario que traduce entre letras y números. UTF-8 es el diccionario más usado del planeta: cubre el español con sus acentos y la ñ, el inglés, y prácticamente todos los idiomas del mundo.

Cuando le pasas "utf8" como segundo parámetro a `fs.readFile` o `fs.writeFile`, le estás diciendo a Node.js: "traduce entre texto y bytes usando el diccionario UTF-8". Si escribes con utf8 y lees

con utf8, todo coincide y tus acentos se ven perfectos. El problema aparece cuando omites la codificación al leer: entonces `readFile` te devuelve un Buffer con los bytes crudos, y si intentas mostrarlo como texto sin convertirlo, ves basura o códigos hexadecimales. La solución es simple: convierte el Buffer a texto con `buffer.toString("utf8")`, o mejor aun, pasa "utf8" desde el principio y olvídate del problema.

¿Qué pasa si mezclas codificaciones? Si un archivo se escribió con una codificación distinta (por ejemplo, un archivo viejo de Windows en latin1) y lo lees como utf8, los caracteres especiales se ven corruptos: aparecen símbolos raros como "Ã©" en lugar de "é". Esto se llama "mojibake" y es la pesadilla de quien procesa archivos de distintas fuentes. La lección: cuando tu programa va a intercambiar archivos con el mundo exterior, define y documenta la codificación, y utf8 es casi siempre la respuesta correcta.

Para los archivos que tu programa crea y lee por si mismo, la regla es trivial: usa "utf8" en todas partes y nunca tendrás problemas. Escríbelo siempre, aunque Node.js lo asuma por defecto en algunos casos; ser explícito es gratis y te protege de sorpresas.

El manejo de errores: el primer parámetro err manda

En el mundo de los archivos, los errores no son la excepción: son el pan de cada día. El archivo no existe, el disco está lleno, no tienes permiso, la ruta tiene un error de dedo, otro programa tiene el archivo bloqueado. Un programador novato escribe código que funciona cuando todo sale bien; un programador profesional escribe código que se comporta con dignidad cuando algo sale mal. En fs con callbacks, toda esa dignidad cabe en el primer parámetro del callback: `err`.

La convención es inquebrantable: en cada callback de fs, el primer parámetro es el error. Si la operación tuvo éxito, `err` vale null. Si algo falló, `err` es un objeto `Error` con información útil: `err.code` te dice el tipo de error con un código corto, y `err.message` te lo explica en palabras. El patrón que debes grabarte a fuego es este:

```
fs.readFile("archivo.txt", "utf8", (err, datos) => {
  if (err) {
    // Manejar el error AQUÍ, al inicio, y salir
    console.error("Fallo la lectura:", err.code, "-", err.message);
    return;
  }
  // Solo si llegamos aquí, todo salió bien
  console.log(datos);
});
```

Este patrón se llama "early return" (retorno temprano): revisas el error primero, lo atiendes, y sales de la función con return. Todo el código "feliz" queda después, sin anidarse dentro de un else. Es más legible y evita la famosa "piramide de la perdición", ese código con tantos niveles de anidación que parece una escalera hacia la derecha.

Los códigos de error más comunes que vas a ver tienen nombres que parecen telegramas: ENOENT significa "el archivo o la carpeta no existe" (viene de "Error NO ENTry", no hay tal entrada); EACCES significa "permiso denegado"; EEXIST significa "ya existe" (cuando intentas crear algo que ya está ahí); EISDIR significa "es un directorio" (cuando intentas leer una carpeta como si fuera un archivo); ENOTDIR significa "no es un directorio" (cuando parte de tu ruta atraviesa algo que no es carpeta). Aprenderte estos códigos te convierte en detective: cuando tu programa falle, el código te dice exactamente que buscar.

Un error clásico de principiante es ignorar el parámetro err por completo, como si no existiera. El programa parece funcionar en las pruebas (porque el archivo sí existe en tu máquina) y luego falla misteriosamente en la computadora del usuario, sin ningún mensaje que explique que pasó. Otro error clásico es manejar el error pero seguir adelante como si nada: imprimir el error y luego intentar usar datos que nunca llegaron, lo que provoca un segundo error más confuso. La regla es: si err tiene valor, atiende el error y detente (return); no intentes usar los datos.

Con fs/promises el manejo de errores se hace con try/catch, que ya conoces: envuelves tus awaits en un try y atrapas cualquier falló en el catch. Es el mismo espíritu, con sintaxis más limpia. Y con las versiones Sync, también usas try/catch, porque una función Sync que falla lanza una excepción en lugar de devolverla en un callback.

JSON en archivos: tu primera "base de datos"

Aquí viene una de las ideas más poderosas de esta unidad: puedes usar archivos de texto para guardar datos estructurados de tu programa, como una base de datos sencilla sin instalar nada. La técnica se basa en JSON (JavaScript Object Notation), el formato de texto que ya conoces para representar objetos y arreglos.

El truco tiene dos pasos. Para guardar: tomas tu objeto o arreglo de JavaScript y lo conviertes a texto con JSON.stringify. Ese texto lo escribes en un archivo con fs.writeFile. Para leer: lees el archivo con fs.readFile, obtienes el texto, y lo conviertes de vuelta a objeto con JSON.parse. Así de simple. Tu programa ahora tiene memoria permanente para datos estructurados: listas de tareas, agendas de contactos, configuraciones, puntuaciones de un juego.

```

const fs = require("fs");

// 1. Datos en memoria
const agenda = [
  { nombre: "Ana", telefono: "555-1234" },
  { nombre: "Luis", telefono: "555-5678" }
];

// 2. Convertir a texto JSON y guardar en archivo
const texto = JSON.stringify(agenda, null, 2);
fs.writeFile("agenda.json", texto, "utf8", (err) => {
  if (err) {
    console.error("No se pudo guardar:", err.message);
    return;
  }
  console.log("Agenda guardada");

  // 3. Leer el archivo y convertir de vuelta a objeto
  fs.readFile("agenda.json", "utf8", (err2, contenido) => {
    if (err2) {
      console.error("No se pudo leer:", err2.message);
      return;
    }
    const agendaLeida = JSON.parse(contenido);
    console.log("Contactos recuperados:", agendaLeida.length);
    console.log("Primer contacto:", agendaLeida[0].nombre);
  });
});

```

Fíjate en `JSON.stringify(agenda, null, 2)`: el tercer parámetro, el número 2, le dice que indente el texto con dos espacios por nivel. Sin el, todo el JSON saldría en una sola línea larguísima, válido pero ilegible para un humano. Con el 2, el archivo se ve bonito y ordenado, y cuando necesites abrirlo para revisarlo a mano (que va a pasar más veces de las que crees), lo vas a agradecer. Esos dos caracteres extra de esfuerzo te ahorran minutos de entrecerrar los ojos.

Hay dos trampas clásicas con JSON que debes conocer. Primera: `JSON.parse` falla con un error si el texto no es JSON válido. Esto pasa cuando el archivo está vacío, cuando alguien lo edita a mano y deja una coma de más, o cuando el programa se cerró a la mitad de una escritura y el archivo quedó corrupto. Por eso, el `JSON.parse` siempre debe ir dentro de un `try/catch` (o manejar el error del callback con cuidado). Segunda: `JSON.stringify` falla si tu objeto tiene referencias circulares (un objeto que se contiene a sí mismo, directa o indirectamente). Es raro en datos sencillos, pero si te topas con el error "Converting circular structure to JSON", ya sabes qué es eso.

Una buena práctica para archivos JSON que tu programa lee al arrancar: si el archivo no existe (error `ENOENT`), no lo trates como una catástrofe; trátalo como "todavía no hay datos guardados" y empieza con una estructura vacía. Así tu programa funciona bien la primera vez que se ejecuta, crea el archivo cuando guarda por primera vez, y en las siguientes ejecuciones encuentra los datos anteriores. Ese es exactamente el comportamiento de "memoria permanente" que promete la clase 8: la primera vez el programa nace sin recuerdos, y a partir de ahí recuerda todo.

El módulo `path`: el arquitecto de direcciones

Si `fs` es el bibliotecario, `path` es el arquitecto que dibuja los planos de la biblioteca. Su trabajo no es tocar archivos, sino construir, analizar y manipular las rutas (las direcciones) que llevan a ellos. Y aunque suena menos emocionante que leer y escribir archivos, dominar `path` es lo que separa un programa que "funciona en mi máquina" de un programa profesional que funciona en cualquier computadora del mundo.

El problema que `path` resuelve es real y cotidiano: Windows, Mac y Linux escriben las direcciones de los archivos de forma distinta. En Windows, una ruta se ve así: `C:\usuarios\ana\documentos\notas.txt`, con barras invertidas (backslash) y una letra de unidad al inicio. En Linux y Mac, la misma idea se escribe así: `/home/ana/documentos/notas.txt`, con barras normales (slash) y sin letra de unidad. Si tu código construye rutas pegando texto a mano, como `"carpeta" + "/" + "archivo.txt"`, ese código funciona en Linux y Mac pero puede fallar en Windows, o viceversa. Y como tu programa puede terminar ejecutándose en un servidor Linux aunque tu lo desarrolles en una laptop Windows (qué es el caso más común del mundo), necesitas una forma de construir rutas que funcione en todas partes sin que tu tengas que pensar en el sistema operativo. Esa forma es `path`.

Como `fs`, `path` es un módulo del núcleo: lo cargas con `const path = require("path")` y no necesitas instalar nada. Vamos a conocer sus funciones más importantes.

`path.join` es la estrella del módulo y la función que más vas a usar en tu vida como programador de Node.js. Toma varios segmentos de texto y los une en una sola ruta, usando el separador correcto según el sistema operativo donde se ejecute el programa. Además, es inteligente: elimina barras duplicadas, resuelve los "." (carpeta actual) y los ".." (subir un nivel), y nunca te deja una ruta mal formada por un slash de más o de menos.

```
const path = require("path");

// En Linux/Mac produce: datos/notas.txt
// En Windows produce: datos\notas.txt
const ruta = path.join("datos", "notas.txt");
console.log(ruta);

// Limpia el desorden automáticamente
console.log(path.join("datos", "notas.txt")); // datos/notas.txt
console.log(path.join("datos/", "/notas.txt")); // datos/notas.txt (sin duplicar barras)
console.log(path.join("a", "b", "..", "c")); // a/c (el ".." cancela la "b")
```

Fíjate en la última línea: `path.join` entiende la navegación de carpetas. Unirte "a", "b", "..", "c" equivale a entrar a "a", entrar a "b", salir un nivel (el ".."), y entrar a "c": el resultado es "a/c". Esta normalización automática te ahorra una cantidad enorme de dolores de cabeza cuando las rutas vienen de distintas fuentes y cada una trae sus propias manías con las barras.

`path.resolve` es el primo serio de `join`. Mientras `join` simplemente pega segmentos, `resolve` construye una ruta ABSOLUTA: una dirección completa desde la raíz del sistema hasta el destino, sin ambigüedades. Si le das segmentos relativos, los resuelve tomando como base la carpeta desde donde se ejecutó el programa (el directorio de trabajo actual). Además procesa los segmentos de derecha a izquierda, deteniéndose en cuanto encuentra una ruta absoluta.

```
const path = require("path");

// Supongamos que el programa se ejecuta desde /home/ana/proyecto
console.log(path.resolve("datos", "notas.txt"));
// /home/ana/proyecto/datos/notas.txt

console.log(path.resolve("/tmp", "datos", "notas.txt"));
// /tmp/datos/notas.txt (al encontrar "/tmp", que es absoluta, ignora lo anterior)
```

La diferencia entre `join` y `resolve` merece atención porque es una de las confusiones más comunes. `path.join("a", "b")` te da "a/b": una ruta relativa, útil cuando la vas a combinar con otra base después. `path.resolve("a", "b")` te da "/ruta/completa/a/b": una ruta absoluta lista para

usar. Otra diferencia: si algun segmento es una ruta absoluta, join la trata como un segmento más y la pega (produciendo algo raro como "a//tmp/b"), mientras que resolve la usa como punto de partida y descarta todo lo anterior. Regla práctica: usa join para construir rutas relativas o para combinar segmentos sobre una base que tu controlas (como `__dirname`, que veremos enseguida); usa `resolve` cuando necesites una ruta absoluta definitiva, por ejemplo para mostrarle al usuario exactamente donde quedó un archivo o para pasarle la ruta a otra herramienta que exige direcciones completas.

`path.basename` extrae el nombre del archivo de una ruta, quitando todo el camino de carpetas. Es útil cuando recibes una ruta larga y solo te interesa el nombre final.

```
path.basename("/home/ana/documentos/reporte.pdf"); // "reporte.pdf"
path.basename("/home/ana/documentos/reporte.pdf", ".pdf"); // "reporte" (quita la extension)
```

El segundo parámetro opcional es la extensión a recortar: si el nombre termina con esa extensión, la elimina. Es perfecto para obtener el nombre "limpio" de un archivo.

`path.dirname` hace lo contrario: extrae la carpeta que contiene al archivo, quitando el nombre final.

```
path.dirname("/home/ana/documentos/reporte.pdf"); // "/home/ana/documentos"
```

`path.extname` devuelve la extensión del archivo, incluyendo el punto: `path.extname("foto.png")` devuelve `".png"`. Es la forma confiable de saber de que tipo es un archivo sin adivinar por el nombre completo. Un truco comun es comparar `extname` en minusculas para filtrar archivos: si `(path.extname(nombre).toLowerCase() === ".txt")`, porque algunos archivos vienen con la extensión en mayusculas ("NOTAS.TXT") y no quieres que se te escapen.

`path.parse` descompone una ruta en sus partes (root, dir, base, ext, name) y `path.format` hace lo inverso: arma una ruta a partir de esas partes. Son menos usados en el día a día, pero muy utiles cuando necesitas modificar una parte de la ruta, como cambiarle la extensión a un archivo conservando todo lo demás.

```
const partes = path.parse("/home/ana/documentos/reporte.pdf");
// { root: "/", dir: "/home/ana/documentos", base: "reporte.pdf",
//   ext: ".pdf", name: "reporte" }
```

`path.sep` te dice cual es el separador de carpetas en el sistema operativo actual: `"\"` en Windows, `"/"` en Linux y Mac. Casi nunca necesitas usarlo directamente (porque `join` ya lo

maneja), pero es útil para entender qué está pasando cuando depuras rutas. `path.delimiter` es otra cosa: es el separador que usa el sistema para listas de rutas (como la variable de entorno `PATH`): `;` en Windows y `:` en Linux/Mac. No lo confundas con `sep`.

__dirname y el patrón de oro

Ahora llegamos al concepto más importante de toda la sección de `path`, el que vas a usar en literalmente todos tus proyectos: `__dirname`. (Con dos guiones bajos al inicio y al final; escríbelo mal y no funciona.)

`__dirname` es una variable mágica que Node.js te regala en cada archivo: contiene la ruta absoluta de la carpeta donde vive el archivo JavaScript que se está ejecutando. No la carpeta desde donde abriste la terminal, no la carpeta del proyecto en general: la carpeta exacta del archivo `.js` actual. Esto es oro puro, porque te da un punto de referencia fijo y confiable para construir rutas, sin importar desde donde el usuario ejecute tu programa.

Para entender por qué importa, necesitas conocer a su prima confusa: `process.cwd()`. Esta función devuelve el "current working directory", la carpeta desde donde se lanzó el comando `node` en la terminal. Si tu archivo está en `/home/ana/proyecto/src/app.js` pero abres la terminal en `/home/ana` y ejecutas `node proyecto/src/app.js`, entonces `__dirname` vale `/home/ana/proyecto/src` mientras que `process.cwd()` vale `/home/ana`. Son dos cosas distintas, y confundirlas es una de las fuentes más comunes de errores "en mi máquina sí funciona".

El problema clásico: escribes `fs.readFile("datos/notas.txt", ...)` con una ruta relativa, y funciona perfecto cuando ejecutas el programa desde su propia carpeta. Pero un día lo ejecutas desde otra carpeta (o lo pones en un servidor donde el proceso arranca desde otro directorio), y de repente el archivo "no existe", aunque no moviste nada. Lo que pasó es que la ruta relativa se resolvió contra `process.cwd()`, que cambió, mientras tu archivo seguía en el mismo lugar. La solución es el patrón de oro:

```
const fs = require("fs");
const path = require("path");

// PATRÓN DE ORO: __dirname + path.join
const rutaNotas = path.join(__dirname, "datos", "notas.txt");

fs.readFile(rutaNotas, "utf8", (err, datos) => {
  if (err) {
    console.error("No se pudo leer:", err.message);
    return;
  }
  console.log(datos);
});
```

`path.join(__dirname, "datos", "notas.txt")` construye una ruta absoluta que apunta a la carpeta "datos" que esta junto a tu archivo .js, sin importar desde donde se ejecute el programa ni en que sistema operativo corra. `__dirname` te da la base absoluta y confiable, `join` une los segmentos con el separador correcto. Memoriza este patrón como si fuera tu número de teléfono: lo vas a escribir cientos de veces en tu carrera. Cuando veas código profesional de Node.js, lo vas a encontrar por todas partes: sirviendo archivos HTML, cargando configuraciones, leyendo plantillas, guardando subidas de usuarios. Todo lo que involucre una ruta a un archivo real pasa por `__dirname + path.join`.

Existe también `__filename`, la prima de `__dirname`, que contiene la ruta absoluta del archivo actual incluyendo su nombre (por ejemplo `/home/ana/proyecto/src/app.js`). Es menos usada, pero útil cuando necesitas saber exactamente que archivo se esta ejecutando, por ejemplo para mensajes de error detallados o herramientas de diagnostico. Y si alguna vez necesitas solo la carpeta a partir de `__filename`, `path.dirname(__filename)` te devuelve exactamente lo mismo que `__dirname`.

Por qué nunca debes concatenar rutas a mano

Ya vimos que `path.join` existe para unir rutas; ahora entendamos por qué hacerlo a mano con `+` o con plantillas de texto es una mala idea, aunque "funcione" en tus pruebas. El primer problema son los separadores: si escribes `"datos" + "/" + "notas.txt"`, en Windows obtienes `"datos/notas.txt"`, que a veces funciona (Windows tolera el slash en muchos casos) y a veces no, dependiendo de la herramienta que reciba la ruta. Es un bug intermitente, la peor clase de bug, porque a veces falla y a veces no, y nunca sabes por qué.

El segundo problema son las barras duplicadas o faltantes. Si una variable trae su slash al final y tu le agregas otro, obtienes `"datos//notas.txt"`. Si no lo trae y tu asumes que si, obtienes

"datosnotas.txt", que apunta a un archivo que no existe con un nombre pegado. `path.join` normaliza todo esto automáticamente; la concatenación manual te deja a merced de como venga cada pedazo.

El tercer problema, y el más grave, es la seguridad: concatenar rutas con texto que viene del usuario abre la puerta al ataque de "path traversal" (viaje entre carpetas), del que hablaremos en la sección de buenas prácticas. Un usuario malicioso puede enviarte un nombre de archivo como `".././secretos/claves.txt"` y, si tu lo pegas a mano a tu carpeta base, terminas leyendo archivos fuera de tu zona permitida. `path.join` por si solo no te protege de esto (normaliza la ruta, pero igual te lleva fuera), así que necesitas validación extra; pero concatenar a mano ni siquiera te da la normalización.

En resumen: cada vez que sientas la tentación de escribir ruta + "/" + archivo, detente y escribe `path.join(ruta, archivo)`. Es más corto de leer, funciona en todos los sistemas operativos, normaliza las barras y es lo que todo programador de Node.js espera ver. Hacerlo a mano es como clavar con una piedra teniendo un martillo en la mano: se puede, pero no hay razón.

Permisos y errores comunes: cuando el disco dice que no

Trabajar con archivos reales significa lidiar con un mundo donde muchas cosas pueden salir mal, y la mayoría de esos problemas se manifiestan como errores con códigos crípticos. Ya conociste `ENOENT` (no existe) en la sección de manejo de errores; ahora vamos a ver el catálogo completo de problemas típicos y, más importante, como resolver cada uno.

`EACCES` (permiso denegado) aparece cuando tu programa intenta leer o escribir algo para lo que el sistema operativo no le dio permiso. Los casos típicos: intentar escribir en una carpeta del sistema (como `C:\Windows` o `/etc`), leer un archivo de otro usuario en un servidor compartido, o ejecutar tu programa con un usuario distinto al dueño de los archivos. Como resolverlo: primero, verifica que la ruta sea la que crees (muchos `EACCES` son en realidad rutas mal escritas que apuntan a lugares protegidos); segundo, revisa los permisos del archivo o carpeta con las herramientas de tu sistema; tercero, en desarrollo, trabaja dentro de tu carpeta de usuario donde tienes todos los permisos. En un servidor real, la solución correcta es configurar bien los permisos y el usuario con el que corre tu aplicación, no ejecutar todo como administrador "para que funcione", que es un agujero de seguridad.

`EPERM` (operación no permitida) es primo de `EACCES` pero más específico de Windows: aparece cuando intentas borrar o modificar un archivo que esta bloqueado. Y eso nos lleva a uno de los problemas más frustrantes: los archivos bloqueados. En Windows, si un archivo está abierto en otro programa (un editor de texto, un visor de PDF, incluso el explorador de

archivos generando una vista previa), el sistema puede negarte el acceso para escribirlo o borrarlo. Tu programa recibe un error y tu juras que el código está bien... y si está bien, el problema es que el archivo está ocupado. La solución: cierra el programa que lo tiene abierto. En tus scripts, un patrón útil es reintentar la operación unas cuantas veces con una pequeña espera entre intentos, porque a veces el bloqueo es momentáneo (un antivirus escaneando el archivo, por ejemplo).

EBUSY (recurso ocupado) es similar: el archivo o la carpeta está en uso y no se puede tocar ahora mismo. ENOTEMPTY aparece cuando intentas borrar una carpeta que todavía tiene archivos dentro (`fs.rmdir` en versiones viejas se negaba; la forma moderna es `fs.rm` con `{ recursive: true }`). EEXIST ya la conoces: intentaste crear algo que ya existe. EISDIR: trataste una carpeta como si fuera un archivo (por ejemplo, `fs.readFile` sobre una carpeta). ENOTDIR: parte de tu ruta atraviesa algo que no es carpeta.

Más allá de memorizar códigos, lo importante es desarrollar un método de diagnóstico. Cuando una operación de `fs` falle, sigue estos pasos en orden. Primero: lee el `err.code` y el `err.message` completos; no adivines, lee. Segundo: imprime la ruta exacta que le pasaste a la función con `console.log`, porque el 80% de las veces el problema es que la ruta no es la que creías (un `__dirname` mal usado, una carpeta que no existe, un nombre con un error de dedo). Tercero: verifica manualmente que la ruta exista y que tengas permiso, usando tu explorador de archivos o la terminal. Cuarto: si el código es `ENOENT`, crea las carpetas padre con `fs.mkdir` y `{ recursive: true }` antes de escribir; si es `EACCES` o `EPERM`, revisa permisos y archivos bloqueados. Este método te resuelve el 99% de los problemas de archivos.

Un caso especial que merece mención: escribir un archivo en una carpeta que no existe. `fs.writeFile` NO crea las carpetas padre automáticamente; si la carpeta "datos" no existe, obtienes `ENOENT` aunque el nombre del archivo esté perfecto. La solución es crear la carpeta primero con `fs.mkdir(rutaCarpeta, { recursive: true })`. Este es quizá el error número uno de los principiantes con `writeFile`, así que grabatelo: primero la carpeta, después el archivo. En tus programas robustos, es buena idea asegurarte de que la carpeta de destino exista antes de cada escritura importante.

Buenas prácticas: escribe código de archivos como un profesional

Ya conoces las herramientas; ahora las reglas no escritas que distinguen al código aficionado del código profesional. Estas prácticas vienen de años de experiencia colectiva y de errores que otros ya cometieron por ti.

Primera: nunca uses las versiones Sync en un servidor o en código que atiende usuarios. Ya lo explicamos con la analogía del restaurante, pero vale repetirlo porque es el error de

rendimiento más común: un solo `fs.readFileSync` en la ruta de una petición web congela a todos los usuarios mientras el disco responde. Usa `fs/promises` con `async/await`. Las versiones Sync tienen su lugar (scripts de una sola ejecución, lectura de configuración al arrancar), pero ese lugar nunca es el interior de un servidor.

Segunda: valida y sanitiza cualquier ruta que venga del usuario. Este es un tema de seguridad serio. Imagina un servidor que sirve archivos así: `fs.readFile(path.join(__dirname, "público", nombrePedido))`. Si `nombrePedido` viene del usuario y el usuario pide `"../secretos.txt"`, el `join` normaliza la ruta y terminas fuera de tu carpeta "público", entregando archivos privados. Esto se llama `path traversal` y es una de las vulnerabilidades más explotadas de la web. La defensa tiene dos capas: una, normaliza la ruta resultante con `path.resolve` y verifica que empiece con tu carpeta base permitida; si la ruta final no está dentro de la base, rechaza la petición. Dos, sanitiza el nombre: quédate solo con `path.basename(nombrePedido)` para eliminar cualquier intento de navegación entre carpetas, o valida el nombre contra una lista de caracteres permitidos. Nunca confíes en texto que viene de afuera para construir rutas sin revisarlo.

```
const path = require("path");
const basePermitida = path.resolve(__dirname, "publico");

function rutaSegura(nombrePedido) {
  // Nos quedamos solo con el nombre del archivo, sin carpetas
  const nombreLimpio = path.basename(nombrePedido);
  const rutaFinal = path.resolve(basePermitida, nombreLimpio);
  // Verificamos que no se salga de la carpeta permitida
  if (!rutaFinal.startsWith(basePermitida)) {
    throw new Error("Ruta no permitida");
  }
  return rutaFinal;
}
```

Tercera: para archivos grandes, no uses `readFile`. Leer un archivo completo lo carga entero en la memoria RAM; si el archivo pesa 2 GB y tu servidor tiene 1 GB de memoria, el programa se muere. Para archivos grandes existen los streams (flujos), que procesan el archivo por pedazos sin cargarlo todo de golpe: `fs.createReadStream` y `fs.createWriteStream`. Los streams son un tema avanzado que merece su propia unidad, pero que sepas que existen y cuando usarlos: archivos pequeños y medianos (textos, JSONs, configuraciones), `readFile/writeFile` están perfectos; archivos grandes (videos, respaldos, logs gigantes), streams.

Cuarta: escribe archivos importantes de forma atómica. Si tu programa se cierra a la mitad de un `fs.writeFile` (se va la luz, alguien lo detiene), el archivo puede quedar a medias y corrupto. La técnica profesional es escribir primero a un archivo temporal y luego renombrarlo al nombre

final con `fs.rename`, porque el renombrado es una operación atómica: o se completa o no, nunca queda a medias. Así, tus lectores siempre ven o la versión vieja completa o la nueva completa, jamás un híbrido corrupto. Para tu agenda de contactos o tu base de datos en JSON, este patrón vale oro.

Quinta: maneja siempre el error, incluso cuando "no puede fallar". El disco se llena, los permisos cambian, los archivos se borran por fuera de tu programa. Cada operación de `fs` sin manejo de errores es una bomba de tiempo. Con callbacks, revisa `err` siempre; con promesas, usa `try/catch` siempre. Un programa que falla con un mensaje claro es mil veces mejor que uno que falla en silencio o, peor, que sigue funcionando con datos corruptos.

Sexta: no abuses de `existsSync` para decidir y luego actuar. Ya mencionamos la trampa: entre que preguntas y actúas, el mundo cambia. El patrón profesional es intentar la operación y manejar el error (es más fácil pedir perdón que pedir permiso, dicen). La excepción: chequeos rápidos en scripts sencillos o validaciones de inicio, donde `existsSync` es perfectamente razonable.

Septima: usa codificación `utf8` explícita en cada lectura y escritura de texto. No confíes en los valores por defecto; escríbelo siempre. Tu yo del futuro, depurando un archivo con caracteres corruptos a las once de la noche, te lo va a agradecer.

Octava: cuando guardes JSON, valida al leer. Envuelve `JSON.parse` en `try/catch`, y si el archivo no existe en el primer arranque, empieza con datos vacíos en lugar de fallar. Tu programa debe nacer sin recuerdos y construir su memoria con el uso, no exigir que el archivo ya exista.

Con esto cerramos la explicación detallada. Tienes en tus manos el modelo mental completo: qué es `fs` y sus tres sabores, cada operación esencial, la codificación, el manejo de errores, JSON como base de datos sencilla, el módulo `path` a fondo, el patrón de oro con `__dirname`, y las buenas prácticas que te hacen escribir como profesional. Ahora vamos a ver todo esto en acción con ejemplos completos, paso a paso.

Ejemplos paso a paso

La teoría ya la tienes; ahora vamos a ensuciarnos las manos. En esta sección vamos a construir cuatro programas completos y funcionales, explicados línea por línea, como si estuviéramos programando juntos y yo te fuera narrando lo que pienso mientras escribo. Copia cada ejemplo a tu carpeta de práctica, ejecútalo, modifícalo, rompelos y arreglalos. Así se aprende de verdad.

Ejemplo 1: Diario personal en archivo de texto

Vamos a crear un programa de diario: cada vez que lo ejecutas con un mensaje, agrega una entrada con la fecha al archivo `diario.txt`. Si es la primera vez, crea el archivo. Al final, muestra cuantas entradas tiene el diario. Este ejemplo usa `appendFile` (agregar sin borrar), `readFile` (leer todo) y `writeFile`, y te ensena el flujo típico de "programa con memoria".

Primero, pensemos el diseno antes de escribir código, como hacen los profesionales. Necesitamos: 1) recibir el texto de la entrada (lo pasaremos como argumento en la terminal); 2) construir la línea con fecha y hora; 3) agregarla al final del archivo sin borrar lo anterior; 4) leer el archivo para contar las entradas. El módulo `path` con `__dirname` nos asegura que el diario siempre quede junto a nuestro programa.

```

const fs = require("fs");
const path = require("path");

// 1. Ruta del diario: siempre junto a este archivo, en cualquier sistema
const rutaDiario = path.join(__dirname, "diario.txt");

// 2. El mensaje viene como argumento: node diario.js "Hoy aprendi fs"
const mensaje = process.argv[2];
if (!mensaje) {
  console.log('Uso: node diario.js "tu mensaje del dia"');
  process.exit(1);
}

// 3. Construimos la entrada con fecha y hora en formato legible
const ahora = new Date();
const fecha = ahora.toLocaleString("es-MX");
const entrada = `[${fecha}] ${mensaje}\n`;

// 4. Agregamos la entrada AL FINAL del archivo (sin borrar lo anterior)
fs.appendFile(rutaDiario, entrada, "utf8", (err) => {
  if (err) {
    console.error("No se pudo escribir en el diario:", err.message);
    return;
  }
  console.log("Entrada guardada.");
});

// 5. Leemos el diario completo para contar las entradas
fs.readFile(rutaDiario, "utf8", (err2, contenido) => {
  if (err2) {
    console.error("No se pudo leer el diario:", err2.message);
    return;
  }
  // Cada entrada ocupa una linea; las lineas vacias no cuentan
  const lineas = contenido.split("\n").filter((l) => l.trim() !== "");
  console.log(`Tu diario tiene ${lineas.length} entradas.`);
});
});

```

Vamos paso a paso. `process.argv` es el arreglo con los argumentos de la terminal; `process.argv[2]` es el primero que escribe el usuario después del nombre del archivo. Si no hay mensaje, mostramos cómo se usa y salimos con `process.exit(1)` (el 1 indica que algo salió mal, por convención). La fecha con `toLocaleString("es-MX")` produce algo como "19/9/2026, 10:30:00", en formato mexicano familiar. El `"\n"` al final de la entrada es crucial: sin él, todas las entradas quedarían en una sola línea.

`fs.appendFile` crea el archivo si no existe y agrega al final si ya existe: exactamente lo que un diario necesita. Si hubiéramos usado `writeFile` aquí, cada ejecución borraría las entradas anteriores y el diario solo tendría la última: un diario con amnesia. Después de guardar, leemos todo el archivo y lo partimos por saltos de línea con `split("\n")`; filtramos las líneas vacías (la última línea del archivo suele terminar en `\n`, lo que deja un pedazo vacío al final) y contamos. Ejecútalo varias veces con distintos mensajes y abre `diario.txt` en tu editor: verás como tu programa recuerda todo entre ejecuciones. Esa es la magia de la memoria permanente.

Nota de diseño: este ejemplo usa callbacks anidados (el `readFile` dentro del callback del `appendFile`) porque necesitamos que la lectura ocurra DESPUÉS de la escritura. Con callbacks, el anidamiento es la forma de decir "primero esto, luego aquello". En la versión con `fs/promises` y `async/await`, el mismo flujo se escribiría en líneas secuenciales sin anidar, que es una de las razones por las que te recomende ese sabor. Si quieres practicar, reescribe este diario con `fs/promises`: es un excelente ejercicio.

Ejemplo 2: Agenda de contactos en JSON (guardar, leer y agregar)

Ahora subimos de nivel: en lugar de texto plano, vamos a guardar datos estructurados. Nuestra agenda será un arreglo de objetos con nombre y teléfono, guardado en `agenda.json`. El programa debe: cargar la agenda existente (o empezar vacía si es la primera vez), agregar el contacto nuevo que le pasemos, guardar todo de vuelta y mostrar la lista completa. Este es el patrón de "mini base de datos" que vas a reutilizar en decenas de proyectos.

```
const fs = require("fs");
const path = require("path");

const rutaAgenda = path.join(__dirname, "agenda.json");

// Uso: node agenda.js "Ana" "555-1234"
const nombre = process.argv[2];
const telefono = process.argv[3];
if (!nombre || !telefono) {
  console.log('Uso: node agenda.js "Nombre" "Telefono"');
```

```

    process.exit(1);
  }

  // Paso 1: leer la agenda existente (o empezar vacía si no existe)
  fs.readFile(rutaAgenda, "utf8", (err, contenido) => {
    let agenda = [];

    if (err) {
      if (err.code === "ENOENT") {
        // El archivo no existe: es la primera vez, empezamos vacíos
        console.log("Creando agenda nueva...");
      } else {
        console.error("Error al leer la agenda:", err.message);
        return;
      }
    } else {
      // El archivo existe: convertir el texto JSON de vuelta a arreglo
      try {
        agenda = JSON.parse(contenido);
      } catch (e) {
        console.error("La agenda está corrupta, no se pudo interpretar.");
        return;
      }
    }
  }

  // Paso 2: agregar el contacto nuevo al arreglo en memoria
  agenda.push({ nombre: nombre, telefono: telefono });

  // Paso 3: convertir el arreglo a texto JSON bonito y guardarlo
  const texto = JSON.stringify(agenda, null, 2);
  fs.writeFile(rutaAgenda, texto, "utf8", (err2) => {
    if (err2) {
      console.error("No se pudo guardar la agenda:", err2.message);
      return;
    }
    console.log(`Contacto "${nombre}" agregado.`);
    console.log(`La agenda tiene ${agenda.length} contactos.`);
    agenda.forEach((c, i) => {
      console.log(`${i + 1}. ${c.nombre} - ${c.telefono}`);
    });
  });
});

```

Observa con atención el manejo del error ENOENT en el paso 1: si el archivo no existe, NO es un error fatal; significa que es la primera ejecución y empezamos con un arreglo vacío.

Cualquier otro error (permisos, por ejemplo) si detiene el programa. Esta distinción entre "no existe todavía" y "algo salió mal de verdad" es una de las técnicas más importantes de esta unidad: tu programa nace sin recuerdos y los construye con el uso.

El try/catch alrededor de JSON.parse protege contra archivos corruptos: si alguien edita agenda.json a mano y dejó una coma de más, el programa avisa con un mensaje claro en lugar de romperse con un error críptico. Y JSON.stringify(agenda, null, 2) genera el archivo bonito e indentado, para que puedas abrirlo y leerlo como humano.

Prueba este flujo completo: ejecuta el programa tres veces con tres contactos distintos, abre agenda.json en tu editor para ver como crece, y luego borra el archivo y vuelve a ejecutar: verás el mensaje "Creando agenda nueva..." y todo empieza de cero. Ese ciclo (nacer vacío, recordar, crecer) es exactamente cómo funcionan las aplicaciones reales con sus bases de datos, solo que en versión miniatura y sin instalar nada.

Una mejora que te dejo como reto: agrega un modo para buscar contactos, por ejemplo node agenda.js buscar "Ana", que lea la agenda y muestre solo los que coincidan. Necesitaras filter y un poco de lógica con process.argv. Si lo logras, ya dominas el patrón.

Ejemplo 3: Lector de carpeta que lista archivos con detalles

Este ejemplo es una herramienta de exploración: dado un nombre de carpeta, lista todo lo que contiene indicando si es archivo o carpeta, cuanto pesa y cuando se modifico. Es el tipo de programa que procesa archivos por lote: el primer paso de cualquier herramienta que convierte, comprime, analiza u organiza archivos en masa.

```
const fs = require("fs");
const path = require("path");

// Uso: node listar.js [carpeta] (si no se indica, usa la carpeta actual del script)
const carpeta = process.argv[2]
  ? path.resolve(process.argv[2])
  : __dirname;

// Paso 1: leer los nombres dentro de la carpeta
fs.readdir(carpeta, { withFileTypes: true }, (err, entradas) => {
  if (err) {
    console.error("No se pudo leer la carpeta:", err.message);
    return;
  }
  if (entradas.length === 0) {
    console.log("La carpeta está vacía.");
    return;
  }
});
```

```

}

console.log(`Contenido de: ${carpeta}`);
console.log("-----");

// Paso 2: para cada entrada, obtener sus detalles con fs.stat
let pendientes = entradas.length;
entradas.forEach(entrada => {
  const rutaCompleta = path.join(carpeta, entrada.nombre);
  const tipo = entrada.isDirectory() ? "CARPETA " : "archivo ";

  fs.stat(rutaCompleta, (err2, stats) => {
    if (err2) {
      console.log(`${tipo} ${entrada.nombre} (sin datos: ${err2.code})`);
    } else {
      const kb = (stats.size / 1024).toFixed(1);
      const fecha = stats.mtime.toLocaleDateString("es-MX");
      console.log(`${tipo} ${entrada.nombre} - ${kb} KB - mod: ${fecha}`);
    }
    pendientes--;
    if (pendientes === 0) {
      console.log("-----");
      console.log("Listo.");
    }
  });
});
});
});

```

Hay varias técnicas interesantes aquí. Primero, `path.resolve(process.argv[2])` convierte la carpeta que pida el usuario en ruta absoluta, para que funcione la escriba como la escriba. Segundo, `readdir` con `{ withFileTypes: true }` nos da objetos con métodos `isDirectory()` e `isFile()`, así sabemos el tipo de cada entrada sin llamar a `stat` solo para eso. Tercero, y esto es lo más instructivo: lanzamos un `fs.stat` por cada entrada dentro de un `forEach`, lo que significa que todas las consultas corren EN PARALELO y sus callbacks llegan en orden impredecible. Por eso usamos el contador `pendientes`: cada callback que termina resta uno, y cuando llega a cero sabemos que ya terminaron todas y podemos imprimir el mensaje final. Este patrón del contador es la forma clásica de esperar a varias operaciones asíncronas con callbacks; con `fs/promises` lo harías con `Promise.all`, mucho más elegante.

El tamaño lo mostramos en KB dividiendo entre 1024 y con `toFixed(1)` para un decimal: son esos toques de presentación que convierten una herramienta cruda en algo agradable de usar. Y la fecha de modificación con `toLocaleDateString("es-MX")` la pone en formato mexicano.

Ejecutalo sobre tu carpeta de práctica, luego sobre una carpeta con subcarpetas, y observa que el orden de las líneas puede variar entre ejecuciones: eso es el paralelismo asincrono en acción. Si quisieras el listado ordenado alfabeticamente, ordenarias el arreglo entradas con sort antes del forEach. Pequeño detalle, gran diferencia en herramientas reales.

Ejemplo 4: Copia de seguridad con copyFile y path

Cerramos los ejemplos con una utilidad que todo programador necesita alguna vez: un programa que hace copia de seguridad de un archivo importante, guardandola en una carpeta de respaldos con la fecha en el nombre. Usa copyFile, mkdir con recursive, path para construir todo, y stat para verificar antes de copiar.

```
const fs = require("fs");
const path = require("path");

// Uso: node respaldo.js <archivo-a-respaldar>
const origenPedido = process.argv[2];
if (!origenPedido) {
  console.log("Uso: node respaldo.js <archivo>");
  process.exit(1);
}

// Paso 1: resolver la ruta del archivo original a una ruta absoluta
const rutaOrigen = path.resolve(origenPedido);

// Paso 2: verificar que el origen exista y sea un archivo (no una carpeta)
fs.stat(rutaOrigen, (err, stats) => {
  if (err) {
    console.error("No se puede respaldar:", err.message);
    return;
  }
  if (!stats.isFile()) {
    console.error("Solo se pueden respaldar archivos, no carpetas.");
    return;
  }
}

// Paso 3: preparar la carpeta de respaldos (crear si no existe)
const carpetaRespaldo = path.join(__dirname, "respaldos");
fs.mkdir(carpetaRespaldo, { recursive: true }, (err2) => {
  if (err2) {
    console.error("No se pudo crear la carpeta de respaldos:", err2.message);
    return;
  }
}
```



```

// Paso 4: construir el nombre del respaldo con fecha y hora
const ahora = new Date();
const fecha = ahora.toISOString().slice(0, 10); // "2026-09-19"
const hora = String(ahora.getHours()).padStart(2, "0")
    + String(ahora.getMinutes()).padStart(2, "0"); // "1030"
const base = path.basename(rutaOrigen, path.extname(rutaOrigen));
const ext = path.extname(rutaOrigen);
const nombreRespaldo = `${base}_${fecha}_${hora}${ext}`;
const rutaDestino = path.join(carpetasRespaldo, nombreRespaldo);

// Paso 5: copiar, sin sobrescribir si el destino ya existe
fs.copyFile(rutaOrigen, rutaDestino, fs.constants.COPYFILE_EXCL, (err3) => {
    if (err3) {
        if (err3.code === "EEXIST") {
            console.error("Ya existe un respaldo con ese nombre, espera un minuto.");
        } else {
            console.error("No se pudo copiar:", err3.message);
        }
        return;
    }
    console.log(`Respaldo creado: ${nombreRespaldo}`);
    console.log(`Ubicación: ${rutaDestino}`);
});
});
});

```

Analicemos las decisiones. Verificar con stat antes de copiar nos evita dos problemas: intentar respaldar algo que no existe (error claro al inicio) y pasarle una carpeta a copyFile (que fallaría con EISDIR y un mensaje confuso). mkdir con recursive garantiza que la carpeta "respaldos" exista; si ya existe, no pasa nada, no da error: esa es la belleza de recursive.

El nombre del respaldo combina path.basename (para quedarnos solo con el nombre, sin la ruta que escribió el usuario), path.extname (para conservar la extensión) y la fecha/hora para que cada respaldo sea único y ordenable. Si respaldas "notas.txt" el 19 de septiembre a las 10:30, obtienes "notas_2026-09-19_1030.txt" dentro de la carpeta respaldos. El formato de fecha YYYY-MM-DD tiene una ventaja sutil: ordenado alfabéticamente queda ordenado cronológicamente.

La bandera COPYFILE_EXCL hace que la copia falle si el destino ya existe, en lugar de sobrescribirlo en silencio. Como el nombre incluye fecha y hora hasta el minuto, una colisión significa que ya hiciste un respaldo en el mismo minuto: mejor avisar que destruir el anterior. Fíjate como manejamos ese caso específico revisando err3.code === "EEXIST": mensajes de error específicos para situaciones específicas, en lugar de un genérico "algo falló".

Prueballo: crea un archivo de prueba, respalda dos veces en minutos distintos y lista la carpeta respaldos. Luego intenta respaldar un archivo que no existe y una carpeta, para ver los mensajes de error claros. Este programa, aunque pequeño, ya tiene la estructura de una herramienta profesional: validación de entrada, verificación previa, preparación del destino, operación principal con protección contra sobrescritura, y errores específicos y comprensibles.

Práctica

Llegó la hora de que lo hagas tú. Estos cinco ejercicios están ordenados de menor a mayor dificultad y cubren todo lo importante de la unidad: escribir y leer archivos, los tres sabores de fs, JSON como base de datos, path y __dirname, y manejo de errores. Intenta cada uno por tu cuenta antes de leer la solución; si te atorras más de quince minutos, lee la explicación paso a paso, entiéndela, ciérrala y vuelve a escribir el código tú solo. Esa es la forma en que el conocimiento se queda.

Ejercicio 1: Contador de visitas persistente

Enunciado: escribe un programa llamado visitas.js que lleve la cuenta de cuántas veces se ha ejecutado. Cada vez que corre, lee el número guardado en visitas.txt, le suma 1, lo guarda de vuelta y muestra un mensaje como "Esta es la visita número 5". La primera vez que se ejecute (cuando el archivo no exista), debe empezar en 1 sin mostrar errores.

Pistas: usa fs.readFile y fs.writeFile con callbacks. El caso del archivo inexistente (ENOENT) no es un error fatal: significa "empieza en 0". Recuerda que lo que lees del archivo es texto y necesitas convertirlo a número.

Solución explicada paso a paso:

Paso 1: construir la ruta con path.join(__dirname, "visitas.txt") para que el contador viva junto al programa. Paso 2: intentar leer el archivo. Si err existe y su código es ENOENT, asumimos que es la primera visita y partimos de 0; si es otro error, lo reportamos y nos detenemos. Paso 3: si la lectura tuvo éxito, convertir el texto a número con Number() y validar que sea un número válido (si el archivo tiene basura, Number devuelve NaN; en ese caso también partimos de 0, porque un contador corrupto se reinicia con dignidad). Paso 4: sumar 1, escribir el nuevo valor con writeFile y mostrar el mensaje.

```

const fs = require("fs");
const path = require("path");

const rutaVisitas = path.join(__dirname, "visitas.txt");

fs.readFile(rutaVisitas, "utf8", (err, contenido) => {
  let visitas = 0;

  if (err) {
    if (err.code !== "ENOENT") {
      console.error("Error al leer el contador:", err.message);
      return;
    }
    // ENOENT: primera ejecución, visitas ya vale 0, seguimos
  } else {
    const numero = Number(contenido.trim());
    visitas = Number.isNaN(numero) ? 0 : numero;
  }

  visitas += 1;

  fs.writeFile(rutaVisitas, String(visitas), "utf8", (err2) => {
    if (err2) {
      console.error("Error al guardar el contador:", err2.message);
      return;
    }
    console.log(`Esta es la visita número ${visitas}`);
  });
});

```

Detalles que vale la pena notar: `contenido.trim()` elimina espacios o saltos de línea que pudieran colarse al final del archivo; `Number.isNaN` protege contra archivos corruptos; `String(visitas)` convierte el número a texto antes de escribirlo, porque `writeFile` espera un string (aunque acepta números en algunos casos, ser explícito evita sorpresas). Ejecútalo cinco veces seguidas y verás el número crecer: tu programa recuerda entre ejecuciones. Eso, en una línea de resumen, es toda la unidad.

Ejercicio 2: El mismo contador, pero con fs/promises

Enunciado: reescribe el ejercicio 1 usando el submódulo `fs/promises` con `async/await`, sin callbacks. El comportamiento debe ser idéntico: primera vez empieza en 1, las siguientes incrementa, y los errores reales se reportan con mensajes claros.

Pistas: la estructura es try/catch. El truco está en distinguir ENOENT dentro del catch: si el código es ENOENT, sigues con visitas = 0; si es otro error, lo reportas y sales. Fíjate cómo el código se lee de arriba hacia abajo sin anidarse.

Solución explicada paso a paso:

Paso 1: requerir fs/promises y path. Paso 2: crear una función async principal. Paso 3: dentro de un try, intentar leer; en el catch, revisar err.code: si es ENOENT, continuar con 0; si no, lanzar el error de nuevo o reportarlo y salir. Paso 4: convertir, validar, incrementar y escribir con await, todo en secuencia lineal. Paso 5: envolver la escritura también en try/catch (o en el mismo bloque) y llamar a la función principal con manejo de rechazo.

```
const fs = require("fs/promises");
const path = require("path");

const rutaVisitas = path.join(__dirname, "visitas.txt");

async function contarVisita() {
  let visitas = 0;

  try {
    const contenido = await fs.readFile(rutaVisitas, "utf8");
    const numero = Number(contenido.trim());
    visitas = Number.isNaN(numero) ? 0 : numero;
  } catch (err) {
    if (err.code !== "ENOENT") {
      console.error("Error al leer el contador:", err.message);
      return;
    }
    // Primera ejecución: visitas se queda en 0
  }

  visitas += 1;

  try {
    await fs.writeFile(rutaVisitas, String(visitas), "utf8");
    console.log(`Esta es la visita número ${visitas}`);
  } catch (err) {
    console.error("Error al guardar el contador:", err.message);
  }
}

contarVisita();
```

Compara esta versión con la del ejercicio 1 y siente la diferencia: no hay pirámide de callbacks, el flujo "leer, decidir, escribir, mostrar" se lee como una receta. El manejo de ENOENT quedó dentro del catch con una revisión explícita del código, que es el patrón estándar con promesas. Esta es la forma en que escribirás casi todo tu código de archivos de ahora en adelante. Como reto extra: ¿qué pasaría si dos copias del programa se ejecutan exactamente al mismo tiempo? Ambas leerían el mismo número y una visita se perdería. Ese problema se llama "condición de carrera" y tiene soluciones avanzadas; por ahora basta con saber que existe.

Ejercicio 3: Lista de tareas con comandos (mini base de datos JSON)

Enunciado: escribe tareas.js, un programa que administre una lista de tareas guardada en tareas.json. Debe aceptar tres comandos: agregar ("node tareas.js agregar Comprar leche"), listar ("node tareas.js listar") y completar ("node tareas.js completar 2", que marca la tarea número 2 como hecha). Cada tarea es un objeto { texto, hecha }. Si el archivo no existe, empieza con lista vacía. Usa fs/promises.

Pistas: process.argv[2] es el comando y process.argv[3] el argumento. Para "agregar", el texto puede venir en varias palabras: únelas con process.argv.slice(3).join(" "). Para "completar", el número que da el usuario empieza en 1 pero el arreglo empieza en 0. Guarda siempre con JSON.stringify(lista, null, 2).

Solución explicada paso a paso:

Paso 1: funciones auxiliares cargarTareas() (lee el JSON o devuelve [] si no existe o está corrupto... mejor: si no existe devuelve [], si está corrupto avisa y sale) y guardarTareas(lista). Paso 2: leer el comando. Paso 3: según el comando, operar sobre el arreglo en memoria y guardar. Paso 4: validar entradas (número válido para completar, texto no vacío para agregar).

```
const fs = require("fs/promises");
const path = require("path");

const rutaTareas = path.join(__dirname, "tareas.json");

async function cargarTareas() {
  try {
    const contenido = await fs.readFile(rutaTareas, "utf8");
    return JSON.parse(contenido);
  } catch (err) {
    if (err.code === "ENOENT") return [];
    if (err instanceof SyntaxError) {
      console.error("El archivo de tareas está corrupto.");
    }
  }
}
```

```

        process.exit(1);
    }
    throw err;
}
}

async function guardarTareas(lista) {
    await fs.writeFile(rutaTareas, JSON.stringify(lista, null, 2), "utf8");
}

async function main() {
    const comando = process.argv[2];

    if (comando === "agregar") {
        const texto = process.argv.slice(3).join(" ").trim();
        if (!texto) {
            console.log('Uso: node tareas.js agregar "texto de la tarea"');
            return;
        }
        const lista = await cargarTareas();
        lista.push({ texto: texto, hecha: false });
        await guardarTareas(lista);
        console.log(`Tarea agregada: "${texto}"`);
    } else if (comando === "listar") {
        const lista = await cargarTareas();
        if (lista.length === 0) {
            console.log("No hay tareas. ¡Disfruta tu día libre!");
            return;
        }
        lista.forEach((t, i) => {
            const marca = t.hecha ? "[x]" : "[ ]";
            console.log(`${i + 1}. ${marca} ${t.texto}`);
        });
    } else if (comando === "completar") {
        const num = Number(process.argv[3]);
        const lista = await cargarTareas();
        if (!Number.isInteger(num) || num < 1 || num > lista.length) {
            console.log(`Indica un número válido entre 1 y ${lista.length}`);
            return;
        }
        lista[num - 1].hecha = true;
        await guardarTareas(lista);
        console.log(`Tarea ${num} marcada como completada. ¡Bien hecho!`);
    } else {
        console.log("Comandos: agregar <texto> | listar | completar <número>");
    }
}

```

```
}  
}  
  
main().catch((err) => console.error("Error inesperado:", err.message));
```

Fíjate en los detalles profesionales: `cargarTareas` distingue entre "no existe" (empieza vacío), "corrupto" (avisa y sale con código de error) y "otro error" (lo relanza para que lo atrape el `catch` final). La validación del número en "completar" usa `Number.isInteger` y el rango válido, con mensaje que dice exactamente qué números acepta. El `main().catch` final atrapa cualquier error no previsto. Prueba el ciclo completo: agrega tres tareas, lístalas, completa la 2, vuelve a listar. Acabas de construir una aplicación con persistencia real, la misma arquitectura (en miniatura) que usan las apps de verdad: cargar estado, modificarlo en memoria, guardarlo.

Ejercicio 4: Organizador de descargas por extensión

Enunciado: escribe `organizar.js`, que recibe una carpeta y mueve cada archivo dentro de ella a una subcarpeta según su extensión: los `.txt` a "textos", los `.jpg` y `.png` a "imagenes", los `.pdf` a "documentos", y el resto a "otros". Debe crear las subcarpetas si no existen, no debe intentar mover carpetas (solo archivos) y debe informar cuántos archivos movió a cada categoría. Usa `path.extname` para detectar la extensión y `fs.rename` para mover.

Pistas: lee la carpeta con `readdir` y `{ withFileTypes: true }` para saber qué es archivo. Normaliza la extensión con `toLowerCase()`. Crea cada subcarpeta con `mkdir` y `{ recursive: true }` antes de mover. Como hay varias operaciones asíncronas, usa `fs/promises` con un `for...of` (que sí espera cada `await`, a diferencia de `forEach`).

Solución explicada paso a paso:

Paso 1: resolver la carpeta base a ruta absoluta. Paso 2: definir el mapa de extensión a subcarpeta. Paso 3: leer las entradas filtrando solo archivos. Paso 4: para cada archivo, determinar su categoría, crear la subcarpeta destino, mover con `rename` y contar. Paso 5: mostrar el resumen.

```

const fs = require("fs/promises");
const path = require("path");

const CATEGORIAS = {
  ".txt": "textos",
  ".md": "textos",
  ".jpg": "imagenes",
  ".jpeg": "imagenes",
  ".png": "imagenes",
  ".gif": "imagenes",
  ".pdf": "documentos",
};

async function main() {
  const carpeta = process.argv[2] ? path.resolve(process.argv[2]) : process.cwd();
  const conteo = {};

  const entradas = await fs.readdir(carpeta, { withFileTypes: true });

  for (const entrada of entradas) {
    if (!entrada.isFile()) continue; // las carpetas se quedan donde están

    const ext = path.extname(entrada.name).toLowerCase();
    const categoria = CATEGORIAS[ext] || "otros";
    const destinoDir = path.join(carpeta, categoria);
    await fs.mkdir(destinoDir, { recursive: true });

    const origen = path.join(carpeta, entrada.name);
    const destino = path.join(destinoDir, entrada.name);
    await fs.rename(origen, destino);

    conteo[categoria] = (conteo[categoria] || 0) + 1;
    console.log(`${entrada.name} -> ${categoria}/`);
  }

  console.log("\nResumen:");
  for (const cat of Object.keys(conteo)) {
    console.log(` ${cat}: ${conteo[cat]} archivos`);
  }
}

main().catch((err) => console.error("Error:", err.message));

```


Tres lecciones en uno: el `for...of` con `await` procesa los archivos uno por uno en orden (con `forEach` los `awaits` no se esperarían y el resumen saldría antes de tiempo); el objeto `CATEGORIAS` como mapa hace que agregar nuevos tipos sea trivial (una línea más, sin tocar la lógica); y `mkdir` con `recursive` antes de cada movimiento elimina el error de "carpeta no existe" de raíz. Prueba creando una carpeta de prueba con archivos de mentira (unos `.txt` vacíos, unas imágenes falsas) y ejecútalo: verás cómo tu programa reorganiza el caos en segundos. Esa sensación de "la computadora trabajando para mí" es la recompensa de esta unidad.

Ejercicio 5: Servidor de archivos seguro (integra todo + path traversal)

Enunciado: escribe un mini servidor que sirva archivos de una carpeta "público" usando el módulo `http` (que ya conoces de unidades anteriores). Debe: construir la ruta pedida con `path.join(__dirname, "público", ...)`, protegerse contra path traversal verificando que la ruta final esté dentro de "público", devolver 404 si el archivo no existe, 403 si intentan salirse de la carpeta, y el contenido del archivo con código 200 si todo está bien. Usa `fs/promises`.

Pistas: la URL pedida viene en `req.url` (por ejemplo `"/hola.txt"`); quítale el `"/` inicial. Sanitiza con `path.basename` para eliminar cualquier `"../"`. Luego verifica con `path.resolve` y `startsWith` que la ruta final esté dentro de la base permitida. Lee con `readFile` y responde; si el error es `ENOENT`, responde 404.

Solución explicada paso a paso:

Paso 1: crear la carpeta base con `path.resolve(__dirname, "público")`. Paso 2: en cada petición, tomar `req.url`, decodificarla y limpiarla. Paso 3: construir la ruta y validarla contra la base (doble protección: `basename` + verificación de prefijo). Paso 4: leer el archivo y responder según el resultado.

```
const http = require("http");
const fs = require("fs/promises");
const path = require("path");

const BASE = path.resolve(__dirname, "publico");

function rutaSegura(urlPedida) {
  // Quitamos el "/" inicial y decodificamos caracteres como %2e
  const pedido = decodeURIComponent(urlPedida).replace(/^\/\//, "");
  // Primera capa: nos quedamos solo con el nombre, sin carpetas
  const limpio = path.basename(pedido);
  const rutaFinal = path.resolve(BASE, limpio);
  // Segunda capa: la ruta final debe estar dentro de BASE
```

```

    if (rutaFinal !== BASE && !rutaFinal.startsWith(BASE + path.sep)) {
      return null; // intento de salirse: prohibido
    }
    return rutaFinal;
  }

const servidor = http.createServer(async (req, res) => {
  const ruta = rutaSegura(req.url || "/");

  if (!ruta) {
    res.writeHead(403, { "Content-Type": "text/plain; charset=utf-8" });
    res.end("403 Prohibido: ruta no permitida.");
    return;
  }

  try {
    const stats = await fs.stat(ruta);
    if (!stats.isFile()) {
      res.writeHead(404, { "Content-Type": "text/plain; charset=utf-8" });
      res.end("404 No encontrado.");
      return;
    }
    const contenido = await fs.readFile(ruta, "utf8");
    res.writeHead(200, { "Content-Type": "text/plain; charset=utf-8" });
    res.end(contenido);
  } catch (err) {
    if (err.code === "ENOENT") {
      res.writeHead(404, { "Content-Type": "text/plain; charset=utf-8" });
      res.end("404 No encontrado.");
    } else {
      res.writeHead(500, { "Content-Type": "text/plain; charset=utf-8" });
      res.end("500 Error interno.");
    }
  }
});

servidor.listen(3000, () => {
  console.log("Servidor de archivos en http://localhost:3000");
  console.log("Sirviendo la carpeta:", BASE);
});

```

Para probarlo: crea la carpeta "público" junto al script, ponle un archivo hola.txt, ejecuta y visita <http://localhost:3000/hola.txt>. Luego intenta la prueba del atacante: <http://localhost:3000/../tarear.js> o con %2e%2e%2f codificado. El servidor debe responder 403 en lugar de entregar el archivo. La doble protección (basename que recorta cualquier

navegación, más la verificación de que la ruta final empiece con BASE) es el estándar profesional: si una capa falla, la otra detiene el ataque. Este ejercicio conecta todo lo de la unidad (fs, path, __dirname, errores, seguridad) con lo que ya sabías (http), y el resultado es algo que se parece muchísimo a cómo funcionan los servidores reales cuando sirven archivos estáticos. Si entendiste este ejercicio a fondo, entendiste la unidad.

Resumen de la unidad

Llegamos al final de la unidad 3, y vale la pena detenerse a mirar el camino recorrido, porque fue largo y cada paso construyó sobre el anterior. Empezaste con programas que olvidaban todo al cerrarse y terminas con programas que recuerdan: diarios que acumulan entradas, agendas que crecen con cada ejecución, contadores que nunca se reinician, organizadores que ordenan carpetas enteras y hasta un servidor que entrega archivos de forma segura. Esa es la diferencia entre un programa de juguete y un programa de verdad, y ahora tú sabes construir los de verdad.

Repasemos las ideas grandes, las que deben quedarte grabadas aunque olvides los detalles de sintaxis. Primera: fs es el módulo del núcleo de Node.js para trabajar con el sistema de archivos, y se carga con `const fs = require("fs")`, sin instalar nada. Es tu bibliotecario: lee, escribe, agrega, borra, crea carpetas, lista contenidos, consulta información, mueve y copia archivos. Segunda: casi todas sus operaciones son asíncronas y existen en tres sabores: callbacks (la forma clásica, con el error siempre en el primer parámetro), fs/promises con `async/await` (la forma moderna y recomendada, la más legible) y las versiones Sync (las más simples de leer pero que congelan todo el programa; úsalas solo en scripts pequeños, jamás en un servidor). Tercera: en los callbacks de fs, el primer parámetro `err` manda; revísalo siempre al inicio con retorno temprano, aprende los códigos (ENOENT, EACCES, EEXIST, EISDIR) y distingue entre "el archivo no existe todavía" (empieza vacío) y "algo salió mal de verdad" (detente con un mensaje claro).

Cuarta: `writeFile` crea o sobrescribe por completo (¡cuidado, borra lo anterior!), `appendFile` agrega al final sin borrar (perfecta para diarios y registros), `readFile` lee el contenido completo, `unlink` borra sin papelera, `mkdir` crea carpetas (con `{ recursive: true }` crea toda la cadena), `readdir` lista el contenido de una carpeta, `stat` entrega la ficha técnica (tamaño, fechas, si es archivo o carpeta), `rename` mueve o renombra, `copyFile` duplica (con `COPYFILE_EXCL` evitas sobrescribir por accidente) y `existsSync` responde `true` o `false` para chequeos rápidos. Quinta: usa siempre la codificación "utf8" al leer y escribir texto; si la omites al leer, recibirás un Buffer con bytes crudos en lugar de texto, y si mezclas codificaciones verás caracteres corruptos. Sexta: `JSON.stringify` convierte tus objetos a texto para guardarlos y `JSON.parse` los

reconstruye al leerlos; con ese par de funciones y un archivo .json tienes una mini base de datos sin instalar nada, siempre con el `JSON.parse` protegido por `try/catch` y tratando el archivo inexistente como "lista vacía".

Séptima: `path` es el especialista en rutas, también del núcleo, y su función estrella es `path.join`, que une segmentos con el separador correcto del sistema operativo y normaliza barras duplicadas y navegaciones `".."`. `path.resolve` construye rutas absolutas definitivas; `join` pega segmentos (relativos), `resolve` ancla al sistema de archivos (absolutos). `path.basename` extrae el nombre del archivo, `path.dirname` la carpeta contenedora, `path.extname` la extensión, `path.parse` descompone y `path.format` recompone. Octava: el patrón de oro es `path.join(__dirname, "datos", "notas.txt")`: `__dirname` te da la carpeta del archivo actual como base absoluta y confiable, `join` une el resto de forma multiplataforma. No confundas `__dirname` con `process.cwd()` (la carpeta desde donde se lanzó el comando): las rutas relativas se resuelven contra el segundo, que cambia, mientras el primero es fijo. Novena: nunca concatenes rutas a mano con `+` o plantillas de texto; usa `path.join` siempre, porque las barras difieren entre Windows y Linux/Mac, porque las barras duplicadas o faltantes generan rutas rotas, y porque el texto pegado a mano abre la puerta a ataques. Décima: los errores de archivos se diagnostican con `método` (lee el código, imprime la ruta real, verifica manualmente, crea carpetas padre con `recursive`, revisa permisos y archivos bloqueados), y el código profesional sigue buenas prácticas: nada de `Sync` en servidores, validar y sanitizar toda ruta que venga del usuario (doble defensa contra `path traversal`: `basename` más verificación de prefijo con la carpeta base), `streams` para archivos grandes en lugar de `readFile`, escritura atómica (`temporal + rename`) para datos importantes, manejo de errores en cada operación, `utf8` explícito siempre y `JSON` validado al leer.

Si tuviera que condensar la unidad en tres frases, serían estas: tus programas ahora tienen memoria permanente gracias a `fs`; tus rutas funcionan en cualquier sistema operativo gracias a `path` y al patrón de oro con `__dirname`; y el código que toca archivos debe ser defensivo, porque el disco, los permisos y los usuarios son impredecibles. Con esas tres ideas claras, los detalles (nombres de funciones, parámetros, opciones) los puedes consultar en la documentación cada vez que los necesites; nadie los memoriza todos, ni siquiera los programadores con años de experiencia.

Un consejo para seguir practicando: elige un programita de tu vida diaria y dale memoria. Un registro de gastos donde anotes cada compra, un contador de vasos de agua, una lista de películas por ver, un generador de reportes que lea datos de un CSV y escriba un resumen en texto. Todos esos proyectos usan exactamente lo que aprendiste aquí: leer, escribir, JSON, path. Cada programa con memoria que construyas te acerca un paso más a pensar como desarrollador profesional, porque la persistencia de datos es el corazón de casi todo el software que existe.

Y una mirada hacia adelante: en las próximas unidades vas a combinar lo aprendido aquí con servidores web más completos, donde fs servirá plantillas HTML y archivos estáticos, y path construirá las rutas de forma segura. También conocerás los streams para procesar archivos gigantes sin agotar la memoria, y las bases de datos reales, donde el patrón "cargar, modificar en memoria, guardar" que practicaste con JSON se repite a escala mayor. Todo lo que viene se apoya en esta unidad, así que si algo quedó flojo, este es el momento de repasar los ejemplos y rehacer los ejercicios. Nos vemos en la unidad 4, donde tus programas con memoria empezarán a conversar con el mundo a través de la red.

UNIDAD 4: El sistema operativo y tu primer servidor HTTP

En las unidades anteriores aprendiste a mover archivos con el módulo `fs`, a recibir argumentos con `process.argv`, a manejar rutas con `path` y a instalar paquetes con `npm`. Todo eso fue tu programa hablando consigo mismo: leyendo, escribiendo, organizando. Pero un programa que solo vive dentro de su propia carpeta es como un chef que solo cocina para el: la comida esta buena, pero nadie la prueba. Esta unidad es el momento en que tu programa sale al mundo. Vas a aprender dos cosas enormes. Primera: como preguntarle a la máquina donde corre tu programa quien es, cuanta memoria tiene, cuantos procesadores trae, con el módulo `os`. Segunda, y esta es la grande: como convertir tu programa en un SERVIDOR que responde por internet con el módulo `http`. Al terminar esta unidad, vas a escribir cuatro líneas de código, abrir tu navegador, y ver tu propia página web servida por tu propio servidor. Ese momento, cuando lo veas, te va a cambiar la forma de ver la programación para siempre.

Dejame ponerte en contexto con una analogía que vas a usar toda tu carrera. Piensa en tu programa como un restaurante. Hasta ahora, tu restaurante solo tenía cocina: preparaba platillos (archivos, datos, calculos) pero no tenía comedor ni meseros. Nadie podía entrar a pedir nada. El módulo `http` es lo que abre las puertas del restaurante: pone meseros (el servidor) que reciben pedidos (peticiones) de los clientes (navegadores) y les llevan lo que pidieron (respuestas). Y el módulo `os` es como el encargado que conoce el edificio: sabe cuantos pisos tiene, cuanta luz hay, quienes trabajan ahí. Un buen restaurante necesita las dos cosas: conocer su edificio y atender a sus clientes. Esta unidad te da las dos.

Por qué importa el módulo `os`? Porque tu programa no corre en el vacío: corre en una máquina real, con un sistema operativo real, con memoria limitada y procesadores contados. Un programa profesional se adapta a su máquina. Si tu app necesita mucha memoria, primero pregunta cuanta hay disponible antes de pedir más de la cuenta. Si tu app puede usar varios procesadores, primero pregunta cuantos hay para repartir el trabajo. Si tu app guarda archivos de configuración en la carpeta del usuario, primero pregunta quien es el usuario y cual es su carpeta personal. El módulo `os` es la ventanilla de información de la máquina: tu preguntas, ella responde. Y lo mejor es que no necesitas instalar nada: viene incluido con Node.js, como `fs` y `path`.

Y por qué importa el módulo `http`? Porque es la puerta de entrada al desarrollo web de verdad. Todo lo que ves en internet (páginas, videos, tiendas, redes sociales) funciona con el mismo principio que vas a aprender aquí: un cliente pide algo, un servidor responde algo. Cuando entiendes `http.createServer`, entiendes el corazon de Express, de Fastify, de Next.js y de todos

los frameworks que existen: todos, por dentro, hacen lo que tú vas a hacer a mano en esta unidad. Aprenderlo a mano primero es cómo aprender a cocinar antes de usar el microondas: cuando después uses herramientas más cómodas, vas a saber exactamente que están haciendo por ti.

Una advertencia honesta antes de empezar: esta unidad tiene un momento magico (tu primer servidor respondiendo) y un momento frustrante (el primer error `EADDRINUSE`, el puerto ocupado). Los dos son normales y los dos te los voy a explicar antes de que te pasen, para que cuando lleguen los reconozcas y no te asustes. Programar servidores es un poco cómo aprender a manejar: al principio todo es nuevo (el volante, los pedales, los espejos), pero después de una semana ya no piensas en eso y solo manejas. Tu primer servidor es tu primera vuelta a la manzana: corta, emocionante y el inicio de todo.

Cómo se conecta esta unidad con lo que ya sabes? Directo. Vas a usar `console.log` para ver la información del sistema. Vas a usar funciones flecha para el callback del servidor (las viste en JavaScript y las has usado en los callbacks de `fs`). Vas a usar `if/else` para las rutas, igual que en la unidad de decisiones. Vas a usar `template literals` o concatenación para armar respuestas HTML. Y el concepto de callback asincrono que aprendiste con `fs.readFile` aparece aquí también: el servidor no responde de inmediato, responde CUANDO llega una petición. Todo lo anterior se junta aquí. Y lo que sigue después de esta unidad (Express, APIs REST, bases de datos) se construye ENCIMA de esto. Esta unidad es un cimiento: todo lo web que aprendas después se para sobre estas cuatro líneas.

Hablemos del plan de la unidad, para que sepas a donde vamos. Primero, el módulo `os` a fondo: cada función importante, que devuelve, y para que la usarias en un programa real. Segundo, HTTP explicado desde cero, sin dar nada por sentado: qué es una petición, que es una respuesta, que es un método, que es una URL. Tercero, los puertos: qué son, por qué existen, por qué el 3000 es el favorito de los desarrolladores. Cuarto, el módulo `http`: `createServer`, el callback con `req` y `res`, y las cuatro líneas mágicas. Quinto, `req` y `res` por dentro: que información trae cada petición y como construir cada respuesta. Sexto, `codigos de estado` y `content-type`: el idioma con el que servidores y navegadores se entienden. Séptimo, rutas: como responder diferente según lo que pidan, que es basicamente como nace cualquier sitio web. Octavo, errores comunes y como resolverlos, porque te van a pasar y quiero que llegues preparado. Y al final, ejemplos completos y práctica para que lo fijes con las manos.

Un consejo de ritmo: no intentes memorizar todas las funciones de `os` ni todos los `codigos de estado`. Memoriza las ideas (el servidor recibe peticiones y manda respuestas; cada ruta es un `if`; el puerto es la puerta) y consulta los detalles cuando los necesites. Los profesionales no se saben de memoria cada función: saben que existe y saben donde buscarla. Lo que si quiero que te quede en el cuerpo es el flujo completo: crear el servidor, definir que responde,

escucharlo en un puerto, probarlo en el navegador. Ese flujo lo vas a repetir cientos de veces en tu carrera; que te salga sin pensar es la meta.

Y un último consejo práctico antes de arrancar: a partir de esta unidad vas a correr programas que NO terminan solos. Hasta ahora, tus scripts hacían su trabajo e imprimían algo y se acababan. Un servidor es diferente: se queda corriendo, esperando peticiones, hasta que tu lo detengas con Ctrl+C. Eso es normal y es lo correcto: un restaurante no cierra la cocina después de atender a un cliente. Acostúmbrate a la idea de "programa que se queda vivo" y al atajo Ctrl+C para detenerlo. Lo vas a usar muchísimo.

Listo? Vamos a conocer primero a la máquina donde vive tu programa, y después vamos a abrirle las puertas al mundo.

Explicación detallada

El módulo os: preguntale a la máquina quien es

El módulo os (de "operating system", sistema operativo) es la ventanilla de información de tu computadora. Le preguntas cosas sobre la máquina y te responde con datos. No necesitas instalarlo: viene con Node.js. Lo importas como cualquier módulo interno:

```
const os = require("os");
```

Y ya. Desde ese momento, os es un objeto lleno de funciones que describen la máquina. Vamos a verlas una por una, con que devuelven y para que sirven en la vida real.

os.platform(): en que sistema operativo estoy?

```
console.log(os.platform());
```

Esto devuelve una cadena corta que identifica el sistema operativo: "win32" para Windows, "darwin" para macOS, "linux" para Linux. Ojo con el detalle: Windows se reporta como "win32" aunque sea de 64 bits; es un nombre histórico que se quedó.

Para que sirve? Para adaptar el comportamiento de tu programa según el sistema. Ejemplo real: los comandos de terminal son diferentes en cada sistema. Si tu programa necesita abrir una carpeta en el explorador de archivos, el comando es "explorer" en Windows, "open" en macOS y "xdg-open" en Linux. Con os.platform() decides cual usar:


```
const os = require("os");
const { exec } = require("child_process");

function abrirCarpeta(ruta) {
  const plataforma = os.platform();
  let comando;
  if (plataforma === "win32") {
    comando = "explorer";
  } else if (plataforma === "darwin") {
    comando = "open";
  } else {
    comando = "xdg-open";
  }
  exec(comando + " " + ruta);
}
```

Este patrón (preguntar la plataforma y decidir) aparece en instaladores, herramientas de desarrollo y cualquier programa que se distribuya para varios sistemas. También sirve para rutas: en Windows las rutas usan backslash y en los demás forward slash, aunque el módulo path ya te ayuda con eso. La idea general es: no asumas en que máquina corre tu programa; preguntalo.

os.arch(): de cuantos bits es el procesador?

```
console.log(os.arch());
```

Devuelve la arquitectura del procesador: "x64" (la más común en computadoras de escritorio y laptops), "arm64" (las Mac con chip M1/M2/M3 y muchas tablets), "ia32", "arm", etc. Para que sirve en la práctica? Principalmente para descargar o elegir binarios correctos: si tu programa instala una herramienta externa, necesita la versión compilada para la arquitectura correcta. También para diagnostico: cuando un usuario reporta un error, pedirle la salida de os.platform() y os.arch() te dice exactamente en que ambiente esta. Los programas profesionales suelen incluir esta información en sus reportes de error automaticos.

os.totalmem() y os.freemem(): cuanta memoria hay?

```
console.log(os.totalmem());
console.log(os.freemem());
```

Devuelven la memoria RAM total y la memoria libre, en BYTES. Ojo: en bytes, así que los números son enormes. Una máquina con 8 GB de RAM reporta algo como 8589934592. Para que el número sea legible, lo conviertes a gigabytes dividiendo entre 1024 tres veces (bytes -> kilobytes -> megabytes -> gigabytes):

```
const os = require("os");

function bytesAGB(bytes) {
  return (bytes / 1024 / 1024 / 1024).toFixed(2);
}

const total = os.totalmem();
const libre = os.freemem();
const usada = total - libre;

console.log("Memoria total: " + bytesAGB(total) + " GB");
console.log("Memoria libre: " + bytesAGB(libre) + " GB");
console.log("Memoria usada: " + bytesAGB(usada) + " GB");
console.log("Porcentaje usado: " + ((usada / total) * 100).toFixed(1) + "%");
```

Para que sirve? Para programas que consumen mucha memoria: antes de cargar un archivo gigante en memoria, preguntas cuanta hay libre y decides si lo procesas por partes o completo. Para monitoreo: un servidor que reporta su memoria cada minuto te avisa antes de quedarse sin memoria (qué es una de las formas más feas en que un servidor se cae). Para diagnostico: "mi programa se traba" + "la máquina solo tiene 2 GB libres" = ya sabes por donde empezar a buscar. La memoria es un recurso finito y los buenos programas la respetan: primero preguntan, después usan.

Un detalle importante: `os.freemem()` te dice la memoria libre EN ESTE MOMENTO, y ese número cambia todo el tiempo porque el sistema la asigna y libera constantemente. No lo uses para calculos exactos de "cuanto puedo usar", usalo como referencia aproximada. Y recuerda que tu programa de Node.js, por defecto, tiene un limite de memoria propio (alrededor de 2 GB en versiones viejas, más en las nuevas con `--max-old-space-size`), así que la memoria total de la máquina no es lo mismo que la memoria disponible para tu proceso. Son dos cosas distintas: la de la máquina (`os.totalmem`) y la de tu programa.

os.cpus(): cuantos procesadores tengo?

```
console.log(os.cpus());
```

Devuelve un ARREGLO con la información de cada nucleo lógico del procesador: modelo, velocidad en MHz y tiempos de uso (user, nice, sys, idle, irq). En una máquina con 8 nucleos, el arreglo tiene 8 elementos. Lo más comun es que solo necesites el número:

```
const os = require("os");
console.log("Nucleos disponibles: " + os.cpus().length);
console.log("Modelo: " + os.cpus()[0].model);
```

Para que sirve? Para repartir trabajo. Node.js corre tu código JavaScript en un solo hilo, pero con el módulo cluster o con worker threads puedes aprovechar varios nucleos. La regla de oro es: crea tantos trabajadores como nucleos tengas. Como sabes cuantos hay? `os.cpus().length`. Así tu programa se adapta solo: en una laptop de 4 nucleos crea 4 trabajadores, en un servidor de 32 crea 32, sin que cambies el código. También sirve para decidir: si solo hay 1 o 2 nucleos, mejor no intentes paralelizar agresivamente porque no hay con que.

Cada elemento del arreglo trae tiempos de CPU en milisegundos: cuanto tiempo paso el nucleo trabajando para el usuario (user), para el sistema (sys) o sin hacer nada (idle). Comparando dos mediciones separadas por un segundo puedes calcular el porcentaje de uso de cada nucleo. Es la base de los monitores de sistema: así funcionan las apps que te muestran el uso de CPU en graficas. No lo vas a necesitar todos los días, pero es bueno saber que la información esta ahí cuando la quieras.

os.hostname(): cómo se llama esta máquina?

```
console.log(os.hostname());
```

Devuelve el nombre de la computadora en la red, como "MI-LAPTOP" o "servidor-producción-01". Para que sirve? Para identificar MÁQUINAS cuando tienes varias. Imagina que tu app corre en 5 servidores: cuando algo falla, el log debe decir EN CUAL servidor falló. Incluyes `os.hostname()` en cada mensaje de log y listo: "Error en servidor-producción-03: base de datos no responde". Sin el hostname, tendrías 5 servidores escribiendo logs identicos y no sabrías cual es el enfermo. También sirve para configuración por máquina: "si el hostname es mi laptop de desarrollo, usa la base de datos local; si es el servidor, usa la de producción".

os.uptime(): cuanto tiempo lleva encendida?

```
console.log(os.uptime());
```

Devuelve los segundos que lleva encendida la máquina desde que arranco. 3600 = una hora, 86400 = un día. Para pasarlo a formato legible:

```
const os = require("os");
const segundos = os.uptime();
const dias = Math.floor(segundos / 86400);
const horas = Math.floor((segundos % 86400) / 3600);
const minutos = Math.floor((segundos % 3600) / 60);
console.log("La maquina lleva encendida: " + dias + " dias, " + horas + " horas, "
+ minutos + " minutos");
```

Para que sirve? Para monitoreo y diagnostico. Un servidor que se reinicia solo cada dos días tiene un problema (algo lo esta tumbando). Un uptime larguísimo en un servidor de producción es buena señal de estabilidad. Muchas páginas de "estado del sistema" muestran el uptime como prueba de confiabilidad. Y en tu máquina de desarrollo, si algo raro pasa, un uptime de 30 días te sugiere "oye, quizá ya es hora de reiniciar".

os.userInfo(): quien esta usando la máquina?

```
console.log(os.userInfo());
```

Devuelve un objeto con información del usuario actual: username (nombre de usuario), uid y gid (identificadores numericos en Linux/macOS), shell (la terminal que usa) y homedir (su carpeta personal). Ejemplo de salida en Linux:

```
{
  uid: 1000,
  gid: 1000,
  username: "esteban",
  homedir: "/home/esteban",
  shell: "/bin/bash"
}
```

Para que sirve? Lo más útil es homedir: la carpeta personal del usuario. Si tu programa guarda configuración o archivos del usuario, la guarda ahí, no en una ruta fija que solo existe en tu máquina:

```
const os = require("os");
const path = require("path");

const carpetaConfig = path.join(os.userInfo().homedir, ".miapp");
console.log("Guardare la configuracion en: " + carpetaConfig);
```

Así funciona en Windows (C:\Users\esteban\miapp), en macOS (/Users/esteban/.miapp) y en Linux (/home/esteban/.miapp) sin cambiar el código. También sirve para personalizar: saludar al usuario por su nombre en una herramienta de terminal. Ojo con la privacidad: no mandes el username a servidores externos sin necesidad; es información del usuario.

Existe también `os.homedir()` que devuelve directamente la carpeta personal sin el resto del objeto. Es un atajo comodo cuando solo necesitas eso.

os.EOL y otras utilidades: los detalles que evitan dolores de cabeza

`os.EOL` es el carácter (o caracteres) de fin de línea del sistema: `"\n"` en Linux y macOS, `"\r\n"` en Windows. Para que sirve? Cuando tu programa genera archivos de texto (CSVs, logs, reportes), usa `os.EOL` en lugar de escribir `"\n"` a mano:

```
const os = require("os");
const fs = require("fs");

const lineas = ["nombre,edad", "Ana,25", "Luis,30"];
fs.writeFileSync("datos.csv", lineas.join(os.EOL));
```

Así el archivo se ve bien en el bloc de notas de Windows y en cualquier editor de Linux. Es un detalle pequeno que separa a los programas caseros de los profesionales: el programa profesional genera archivos correctos en cualquier sistema.

Otras funciones utiles que debes conocer aunque las uses menos: `os.tmpdir()` devuelve la carpeta de archivos temporales del sistema (ahí van los archivos que se borran solos); `os.networkInterfaces()` devuelve información de las tarjetas de red y sus direcciones IP (útil cuando tu servidor necesita saber en que IP esta escuchando); `os.loadavg()` devuelve la carga promedio del sistema en Linux/macOS (un número que dice que tan ocupada esta la máquina); `os.release()` y `os.version()` dan detalles de la versión del sistema operativo; y `os.constants` trae constantes del sistema como senales de proceso y códigos de error.

Para que sirve adaptar tu app a la máquina: la filosofía detras de os

Ya viste muchas funciones; ahora la idea que las une. Hay dos formas de programar: la forma ingenua, que asume ("asumo que hay 8 GB de RAM", "asumo que es Windows", "asumo que la carpeta personal es C:\Users\yo"), y la forma profesional, que pregunta. La forma ingenua funciona en TU máquina y se rompe en todas las demás. La forma profesional funciona en cualquier máquina porque nunca asume: pregunta con os y se adapta.

Esto se vuelve critico cuando tu programa sale de tu laptop. Ese servidor que vas a crear en esta unidad, tarde o temprano, va a correr en una computadora que no es la tuya: un servidor rentado, la computadora de un cliente, un contenedor en la nube. Ahí no hay "C:\Users\esteban", quizá hay 1 GB de RAM en lugar de 16, quizá es Linux en lugar de Windows. Si tu código pregunta en lugar de asumir, sobrevive al cambio de máquina. Si asume, muere en el intentó.

El módulo os es barato de usar (las preguntas son instantaneas, no cuestan nada) y te da robustez gratis. Acostumbrate a usarlo cada vez que tu programa dependa de algo de la máquina: rutas personales (homedir), fin de línea (EOL), número de nucleos (cpus), memoria disponible (freemem/totalmem), sistema operativo (platform). Es la diferencia entre un script que solo te sirve a ti y un programa que le sirve a cualquiera.

Que es HTTP: el idioma de la web explicado desde cero

HTTP significa HyperText Transfer Protocol, o sea, Protocolo de Transferencia de Hipertexto. Suena técnico, pero la idea es simple: es el conjunto de reglas que usan dos computadoras para pedir y entregar cosas por internet. Vamos a desarmarlo con la analogía del restaurante, porque te va a acompañar toda la unidad.

Imagina un restaurante. Hay clientes (los que piden) y hay meseros con cocina (los que responden). El cliente no entra a la cocina a cocinarse solo: le dice al mesero que quiere, el mesero lo lleva a la cocina, la cocina lo prepara, y el mesero le trae el platillo. En la web pasa exactamente lo mismo:

- El CLIENTE es el navegador (Chrome, Firefox) o cualquier programa que pide algo. También puede ser tu celular, una app, o tu propio programa de Node.js pidiendo datos a otro servidor.
- El SERVIDOR es tu programa de Node.js con el módulo http. Recibe pedidos y responde.
- La PETICIÓN (request) es el pedido del cliente: "quiero ver la página de inicio", "quiero estos datos en JSON".

- La RESPUESTA (response) es lo que el servidor le manda de vuelta: la página, los datos, o un mensaje de error si no hay lo que pidió.

El cliente siempre inicia la conversación. El servidor nunca le habla primero al cliente: se queda esperando, llega una petición, la atiende, responde, y vuelve a esperar. Como el mesero que esta atento hasta que un cliente levanta la mano. Este modelo se llama petición-respuesta (request-response) y es la base de TODA la web. Cada vez que abres una página, tu navegador manda una o varias peticiones HTTP y el servidor le responde con el HTML, las imagenes, los estilos.

Ahora, que trae una petición por dentro? Cuatro cosas importantes:

1. El MÉTODO: qué quiere hacer el cliente. Los más comunes son GET (quiero VER algo: "traeme la página de inicio") y POST (quiero ENVIAR algo: "aquí van los datos de mi formulario"). Hay más (PUT, DELETE, PATCH), pero GET y POST cubren el 90% de lo que vas a hacer al inicio. Piensa en el método como el verbo del pedido: "traeme" vs "toma esto".
2. La URL: QUE quiere. Por ejemplo / significa "la página principal", /acerca significa "la página acerca de", /api/usuarios significa "los datos de usuarios". La URL es como decirle al mesero "quiero el platillo número 5 del menu": identifica que cosa especifica pides.
3. Los ENCABEZADOS (headers): información extra sobre la petición. Como el cliente diciendo "hablo espanol" (que idioma prefiere), "soy Chrome en Windows" (que navegador es), "acepto imagenes comprimidas" (que formatos entiende). Son metadatos: no son lo que pides, son COMO lo pides.
4. El CUERPO (body): los datos que manda el cliente, si es que manda. En un GET normalmente no hay cuerpo (solo pides). En un POST si hay: los datos del formulario, el JSON con la información nueva. Es como entregarle al mesero la receta escrita de lo que quieres que cocinen.

Y que trae una respuesta por dentro? Tres cosas:

1. El CÓDIGO DE ESTADO (status code): un número que dice como le fue. 200 = "todo bien, aquí esta lo que pediste". 404 = "no encuentre lo que pides". 500 = "algo se rompio de mi lado". Es como el mesero diciendo "en seguida se lo traigo" (200), "no tenemos ese platillo" (404) o "se nos quemó la cocina, disculpe" (500). Los veremos a fondo en su sección.

2. Los ENCABEZADOS de respuesta: información extra sobre lo que se manda. El más importante es Content-Type: "esto que te mandó es HTML" (text/html), "es texto simple" (text/plain), "son datos JSON" (application/json). Sin esto, el navegador no sabría si lo que recibió es una página para mostrar o un archivo para descargar.
3. El CUERPO de la respuesta: lo pedido. El HTML de la página, el texto, el JSON con los datos. Es el platillo servido en la mesa.

Fíjate que la conversación es sencilla y sin memoria: cada petición es independiente. El servidor atiende una petición, responde, y la olvida; si el mismo cliente pide otra cosa un segundo después, es una petición nueva desde cero. A esto le decimos que HTTP es "sin estado" (stateless). Tiene consecuencias importantes: si quieres recordar quien es el cliente entre peticiones (como una sesión de usuario logueado), tienes que inventar un mecanismo encima (cookies, tokens), porque HTTP por si solo no recuerda nada. Es como un mesero con amnesia selectiva: atiende perfecto cada pedido, pero no recuerda que ya te atendió antes.

Analogía completa del restaurante para que no se te olvide: el cliente (navegador) se sienta y levanta la mano (abre una URL). El mesero (servidor) se acerca y pregunta "que va a querer?" (recibe la petición). El cliente dice "quiero la página de inicio, y hablo español" (método GET, URL /, encabezados). El mesero va a la cocina (tu código), preparan el platillo (generan el HTML), y el mesero regresa: "aquí está su página, todo en orden" (código 200, Content-Type text/html, cuerpo con el HTML). El cliente come (el navegador muestra la página). Fin de la interacción. Siguiente cliente.

HTTP es solo texto con reglas. Si pudieras espiar lo que viaja por el cable cuando pides una página, verías algo así:

```
GET /hola HTTP/1.1
Host: localhost:3000
User-Agent: Mozilla/5.0 (Windows...)
Accept: text/html
```

Y la respuesta:

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 48

<html><body><h1>Hola desde mi servidor</h1></body></html>
```

No hay magia: son líneas de texto con un formato acordado. Tu programa de Node.js con el módulo http se encarga de leer ese texto de entrada y de escribir ese texto de salida; tu solo

decides QUE responder. Entender que HTTP es texto con reglas te quita el miedo: no es un monstruo, es una conversación educada entre programas.

Que es un puerto: las puertas del edificio

Tu computadora puede correr muchos programas que usan la red al mismo tiempo: el navegador, el Spotify, tu servidor de Node.js, el antivirus actualizándose. Cuando llega un paquete de datos de internet, como sabe la computadora a cual programa entregárselo? Con el PUERTO. Piensa en tu computadora como un edificio de oficinas y en los puertos como las puertas numeradas del edificio. La dirección IP es la dirección del edificio ("esta computadora"), y el puerto es el número de oficina ("el programa número 3000 dentro de esa computadora"). Un paquete que llega a "mi IP, puerto 3000" va directo a tu servidor de Node.js; uno que llega al puerto 443 va al servidor web profesional.

Los puertos son números del 0 al 65535. Algunos están reservados por convención para servicios conocidos, y te conviene conocerlos:

- Puerto 80: HTTP normal, las páginas web sin cifrar. Cuando escribes `http://algo.com`, el navegador conecta al puerto 80 sin decirlo.
- Puerto 443: HTTPS, las páginas web cifradas (el candadito del navegador). Cuando escribes `https://algo.com`, va al 443.
- Puerto 21: FTP, transferencia de archivos (viejo pero aun usado).
- Puerto 22: SSH, para conectarte a servidores remotos por terminal.
- Puerto 25/587: correo electrónico (SMTP).
- Puerto 3306: MySQL, la base de datos.
- Puerto 5432: PostgreSQL, otra base de datos.
- Puerto 27017: MongoDB, otra base de datos.

Los puertos del 0 al 1023 se llaman "puertos bien conocidos" y en muchos sistemas solo los puede usar el administrador. Por eso en desarrollo NO usas el 80: necesitarías permisos especiales y además probablemente ya hay algo ahí. En su lugar, la comunidad de desarrollo adoptó el puerto 3000 como el estándar para servidores locales. Cuando veas un tutorial que dice "abre `http://localhost:3000`", ya sabes por qué: es la convención. Otros puertos comunes en desarrollo son el 8080, el 5000 y el 8000. No hay nada mágico en el 3000; es simplemente el acuerdo social de los programadores, como manejar por la derecha.

¿Qué es localhost? Es el nombre especial que significa "esta misma computadora". Cuando tu servidor escucha en el puerto 3000 y abres `http://localhost:3000` en tu navegador, le estas

diciendo: "conectate a mi propia computadora, puerta 3000". El navegador y el servidor viven en la misma máquina y hablan entre ellos. También puedes escribir `http://127.0.0.1:3000`, que es lo mismo (127.0.0.1 es la dirección IP de "yo mismo"). En desarrollo, todo pasa en localhost: tú eres el cliente y el servidor a la vez.

Y que significa que un servidor "escucha" (listen) en un puerto? Significa que se sienta junto a esa puerta del edificio y espera a que alguien toque. Mientras tu programa corre, el puerto 3000 esta "ocupado" por el: ningun otro programa puede usarlo al mismo tiempo. Si intentas correr DOS servidores en el mismo puerto, el segundo falla con el famoso error `EADDRINUSE` (dirección en uso). Es como intentar que dos restaurantes usen la misma puerta: no se puede. La solución es simple: detienes uno (Ctrl+C) o usas otro puerto. Este error te va a pasar, garantizado, la primera semana; cuando lo veas, ya sabes qué significa: "ese puerto ya tiene dueño".

Una pregunta frecuente: "mi servidor solo lo puedo ver yo?" Si, mientras escuche en localhost, solo tu computadora puede conectarse. Para que otros lo vean necesitas que escuche en la IP publica de la máquina y abrir el puerto en el firewall, y eso ya es tema de despliegue (deployment), fuera del alcance de esta unidad. En desarrollo, localhost es tu laboratorio privado y seguro: nadie de fuera puede entrar.

El módulo http: tu restaurante abre sus puertas

Llegamos al momento grande. El módulo http viene incluido con Node.js, no hay que instalar nada. Con el creas servidores web. Vamos a ver primero las cuatro líneas magicas completas, y después las desarmamos pieza por pieza hasta que cada palabra tenga sentido.

```
const http = require("http");

const servidor = http.createServer((req, res) => {
  res.end("Hola desde mi servidor");
});

servidor.listen(3000, () => {
  console.log("Servidor corriendo en http://localhost:3000");
});
```

Guardalo como `servidor.js`, correlo con `node servidor.js`, abre tu navegador en `http://localhost:3000`, y vas a ver "Hola desde mi servidor". Cuatro líneas y ya sirves páginas web. Respira ese momento: acabas de hacer lo mismo que hacen los servidores de las páginas más grandes del mundo, en su forma más pura.

Vamos a desarmar cada parte.

Línea 1: `const http = require("http");` Importas el módulo, igual que hiciste con `fs`, `os` y `path`. Nada nuevo aquí: `http` es un objeto con funciones para crear servidores (y también clientes, pero eso es otra historia).

Línea 3: `http.createServer((req, res) => { ... });` Esta es la línea más importante de la unidad. `createServer` CREA el servidor pero no lo enciende todavía: es como construir el restaurante pero aun no abrir las puertas. Recibe UN argumento: una función callback que se va a ejecutar CADA VEZ que llegue una petición. Esa función recibe dos objetos: `req` (la petición, `request`) y `res` (la respuesta, `response`). Todo lo que tu servidor sabe hacer vive dentro de esa función: ahí decides que responder según lo que te pidan.

Fíjate en algo crucial: esta función NO se ejecuta cuando creas el servidor. Se ejecuta después, una vez por cada petición que llegue. Si nadie abre tu página, la función nunca corre. Si 100 personas la abren, corre 100 veces. Es el mismo patrón de callback que viste con `fs.readFile`: "cuando pase X, ejecuta esta función". Aquí X es "cuando llegue una petición HTTP". Tu defines el comportamiento una vez; Node.js lo repite por cada cliente. El mesero aprende la rutina una vez y la repite con cada cliente que llega.

Dentro del callback, la línea `res.end("Hola desde mi servidor");` hace dos cosas: manda el texto como cuerpo de la respuesta y TERMINA la respuesta. El `end` es obligatorio: sin el, el servidor nunca termina de responder y el navegador se queda esperando eternamente (la famosa página "colgada" cargando sin fin). Piensa en `res.end` como el mesero diciendo "aquí está su platillo, provecho" y retirándose: si no se retira, el cliente se queda esperando sin saber si ya terminó.

Línea 7: `servidor.listen(3000, () => { ... });` Aquí si se ABREN las puertas: `listen` pone al servidor a escuchar peticiones en el puerto 3000. El segundo argumento es un callback opcional que se ejecuta UNA vez, cuando el servidor ya quedó encendido correctamente. Ahí pones el `console.log` de confirmación: "Servidor corriendo en `http://localhost:3000`". Ese mensaje en tu terminal es tu señal de que todo salió bien: si lo ves, el restaurante está abierto; si ves un error en su lugar, algo falló (casi siempre el puerto ocupado, que veremos en la sección de errores).

El orden importa: primero creas (`createServer`), después enciendes (`listen`). Y el programa NO termina después del `listen`: se queda corriendo, esperando. En tu terminal vas a ver el mensaje de confirmación y el cursor parpadeando sin regresar: eso significa que el servidor está vivo. Para detenerlo, `Ctrl+C`. Acostúmbrate a ese estado: servidor corriendo = terminal ocupada = `Ctrl+C` para recuperar tu terminal.

El objeto req: lo que el cliente te pide

Cada vez que llega una petición, Node.js crea el objeto req con toda la información de lo que pidió el cliente. Las tres propiedades que más vas a usar:

req.url: la URL pedida, como "/" o "/hola" o "/api/usuarios". Es la parte de la dirección después del dominio y el puerto. Si abres `http://localhost:3000/hola`, req.url vale `"/hola"`. Esta es la propiedad estrella del enrutamiento: según su valor, respondes diferente. Ojo: req.url incluye también los parámetros de consulta si los hay (lo de después del ?), como `"/buscar?q=nodejs"`. Para rutas simples no te estorba, pero que sepas que ahí viene todo junto.

req.method: el método HTTP, como "GET" o "POST". Cuando abres una página en el navegador, el método es GET. Cuando un formulario se envía, normalmente es POST. Al inicio casi todo lo que hagas será GET, pero conocer req.method te permite decidir: "si es GET muestro el formulario, si es POST proceso los datos". Es el verbo del pedido.

req.headers: un objeto con los encabezados que mandó el cliente. Por ejemplo req.headers["user-agent"] te dice que navegador es, req.headers["accept-language"] que idioma prefiere. Las llaves van en minúsculas. No los vas a usar mucho al inicio, pero existen y a veces son útiles: por ejemplo, para responder en el idioma del cliente o para rechazar peticiones de programas sospechosos.

Vamos a ver req en acción con un servidor que te cuenta lo que recibe. Es un ejercicio excelente para entender: en lugar de responder algo fijo, el servidor responde describiendo la petición.

```
const http = require("http");

const servidor = http.createServer((req, res) => {
  res.setHeader("Content-Type", "text/plain; charset=utf-8");
  const info = "Método: " + req.method + "\n" +
    "URL: " + req.url + "\n" +
    "Navegador: " + req.headers["user-agent"];
  res.end(info);
});

servidor.listen(3000, () => {
  console.log("Servidor corriendo en http://localhost:3000");
});
```

Abre `http://localhost:3000/hola` y vas a ver algo como: Método: GET, URL: /hola, Navegador: Mozilla/5.0... Prueba con /otra/cosa y mira como cambia la URL. Este servidor "espejo" es una

herramienta de aprendizaje buenisima: te muestra exactamente que esta pidiendo el navegador, sin misterios.

El objeto res: lo que tu respondes

El objeto res es tu herramienta para construir la respuesta. Tiene tres métodos principales y los vas a usar en este orden: primero los encabezados, después el contenido, al final cerrar.

res.writeHead(código, encabezados): escribe el código de estado y los encabezados de la respuesta. Ejemplo: res.writeHead(200, { "Content-Type": "text/html" }); significa "todo bien (200), y lo que te mandó es HTML". Se llama UNA vez por respuesta y ANTES de mandar contenido. Es como el mesero anunciando "aquí viene su platillo" antes de ponerlo en la mesa: el anuncio va primero.

res.write(datos): manda un pedazo del cuerpo de la respuesta. Puedes llamarlo varias veces para ir mandando el contenido por partes. Es útil cuando la respuesta es grande o la vas armando poco a poco. Cada write es como llevar un plato de la orden a la mesa: puedes llevar varios viajes.

res.end(datos): termina la respuesta. Puede llevar datos finales (el último pedazo) o ir vacío si ya mandaste todo con write. Es OBLIGATORIO llamarlo una vez por petición: sin end, la respuesta nunca se completa. Es el mesero retirandose: "provecho, ya esta todo".

También existe res.statusCode = 404; para fijar el código de estado sin usar writeHead, y res.setHeader("Content-Type", "text/html"); para fijar un encabezado suelto. Son formas más granulares de hacer lo mismo. En la práctica vas a ver los dos estilos; elige el que te resulte más claro.

Ejemplo que usa las tres piezas:

```
const http = require("http");

const servidor = http.createServer((req, res) => {
  res.writeHead(200, { "Content-Type": "text/html; charset=utf-8" });
  res.write("<h1>Hola</h1>");
  res.write("<p>Esto llego en dos partes.</p>");
  res.end("<p>Y esto al cerrar.</p>");
});

servidor.listen(3000, () => {
  console.log("Servidor corriendo en http://localhost:3000");
});
```

El navegador recibe un HTML armado en tres pedazos y lo muestra todo junto. Para respuestas pequeñas, lo normal es saltarse el write y mandar todo en el end: `res.end("<h1>Hola</h1>");` Menos líneas, mismo resultado. Usa write cuando la respuesta se arma por partes de verdad (por ejemplo, leyendo un archivo grande por pedazos).

Un detalle técnico importante: los encabezados se tienen que fijar ANTES de mandar el primer pedazo de contenido. Si llamas `res.write` y después intentas `res.setHeader`, Node.js te va a reclamar con un error ("Cannot set headers after they are sent"). Tiene lógica: una vez que el mesero empezó a servir, ya no puede anunciar que va a traer. Orden correcto siempre: encabezados primero, contenido después, end al final.

Códigos de estado HTTP: el semaforo de la web

El código de estado es el número que resume como le fue a la petición. El navegador lo usa para decidir que hacer, y tu como desarrollador lo usas para diagnosticar. Los códigos se agrupan en familias por su primer digito, y con saber las familias ya entiendes el 90%:

- 2xx = Éxito. Todo salió bien. El 200 OK es el rey: "aquí esta lo que pediste". El 201 Created significa "todo bien Y ADEMÁS cree algo nuevo" (se usa cuando un POST crea un recurso, como registrar un usuario).
- 3xx = Redireccion. "Lo que buscas ya no esta aquí, ve allá". El 301 Moved Permanently es un cambio de domicilio definitivo: el navegador aprende la nueva dirección. Se usa cuando cambias las URLs de tu sitio y no quieres romper los enlaces viejos.
- 4xx = Error del CLIENTE. El cliente pidió algo mal. El 404 Not Found es el más famoso: "no existe lo que pides". El 400 Bad Request es "tu petición viene mal formada". El 401 Unauthorized es "necesitas identificarte primero" y el 403 Forbidden es "se quien eres y aun así no te dejo pasar". La familia 4xx dice: el problema está en el pedido, no en mi cocina.
- 5xx = Error del SERVIDOR. Yo falle. El 500 Internal Server Error es "algo se rompio aquí adentro" (un bug en tu código, la base de datos caída). El 503 Service Unavailable es "estoy saturado o en mantenimiento, intenta más tarde". La familia 5xx dice: el pedido estaba bien, el que falló fui yo.

Memoriza estos cinco: 200 (todo bien), 201 (creado), 301 (se mudo), 404 (no existe), 500 (me rompi). Con esos cubres casi todo lo que vas a hacer en tus primeros proyectos. Las familias te dan el resto por deducción: si empieza con 2, bien; con 4, el cliente la rego; con 5, tu la regaste.

Cómo se fija el código en Node.js? Con `writeHead` o con `statusCode`:

```
res.writeHead(404, { "Content-Type": "text/plain; charset=utf-8" });
res.end("No encuentre lo que buscas");
```

O así:

```
res.statusCode = 404;
res.setHeader("Content-Type", "text/plain; charset=utf-8");
res.end("No encuentre lo que buscas");
```

Si no fijas ningún código, Node.js asume 200. Eso está bien cuando todo sale bien, pero es un error común devolver 200 con un mensaje de "no encontrado": el navegador y los programas que consuman tu servidor creen que todo salió bien cuando no. Regla profesional: el código de estado siempre debe reflejar la realidad. Si no encontraste algo, 404 aunque el mensaje sea amable. Si algo se rompió, 500 aunque te de pena. Los códigos son el idioma honesto entre tu servidor y el resto del mundo.

Un truco de diagnóstico que vas a usar toda tu carrera: cuando algo no funciona en la web, lo primero que miras es el código de estado. Página en blanco? Mira el código: 404 = la URL está mal; 500 = tu código se rompió; 200 con página en blanco = el código llegó bien pero el contenido está vacío (el bug está en lo que generas, no en el servidor). El código de estado es la primera pista del detective: te dice en qué mitad del problema buscar.

Content-Type: diciendo que idioma hablas

El encabezado Content-Type le dice al cliente QUE TIPO de contenido le estás mandando, para que sepa qué hacer con él. Sin Content-Type, el navegador tiene que adivinar, y adivinar en computación siempre termina mal. Los tres que vas a usar al inicio:

- text/plain: texto simple, sin formato. El navegador lo muestra tal cual, como un bloc de notas. Útil para mensajes de diagnóstico y respuestas simples.
- text/html: página web HTML. El navegador lo interpreta y muestra la página con formato, títulos, colores. Es lo que quieres cuando sirves páginas.
- application/json: datos en formato JSON. El navegador (o el programa cliente) lo entiende como datos estructurados, no como página. Es lo que usan las APIs.

Se fija así:

```
res.writeHead(200, { "Content-Type": "text/html; charset=utf-8" });
```

Ese "; charset=utf-8" del final es importante cuando hay acentos o eñes: le dice al navegador "este texto viene en UTF-8, interpretalo bien". Sin el, tus acentos pueden verse como símbolos raros en algunos navegadores. Acostúmbrate a ponerlo siempre en text/plain y text/html. Es un detalle de dos palabras que te ahorra el clásico problema de "por qué se ven mal mis acentos".

Ejemplo de cada uno para que veas la diferencia en el navegador:

```
const http = require("http");

const servidor = http.createServer((req, res) => {
  if (req.url === "/texto") {
    res.writeHead(200, { "Content-Type": "text/plain; charset=utf-8" });
    res.end("Esto es texto simple, sin formato. <h1>Esto NO se interpreta</h1>");
  } else if (req.url === "/pagina") {
    res.writeHead(200, { "Content-Type": "text/html; charset=utf-8" });
    res.end("<h1>Esto es HTML</h1><p>El navegador <b>si</b> lo interpreta.</p>");
  } else if (req.url === "/datos") {
    res.writeHead(200, { "Content-Type": "application/json; charset=utf-8" });
    res.end(JSON.stringify({ nombre: "Ana", edad: 25 }));
  }
});

servidor.listen(3000, () => {
  console.log("Servidor corriendo en http://localhost:3000");
});
```

Abre /texto: verás el <h1> escrito como texto, sin formato. Abre /página: verás un título grande de verdad. Abre /datos: verás el JSON. MISMO mecanismo (res.end con una cadena), TRES resultados distintos, y la única diferencia es el Content-Type. El encabezado es el que le dice al navegador como tratar lo que recibe. (JSON.stringify convierte un objeto de JavaScript a texto JSON; lo verás a fondo cuando hablemos de APIs, por ahora solo usalo así.)

Rutas: así nacen las páginas de tu sitio

Hasta ahora el servidor responde lo mismo sin importar que le pidan. Eso no es un sitio web: un sitio web responde DIFERENTE según la URL. La página principal muestra una cosa, la de contacto otra, y una URL inventada muestra un error 404. Esa idea (URL distinta = respuesta distinta) se llama ENRUTAMIENTO (routing), y en su forma más simple es un if/else sobre req.url. Así nacen las rutas de tu sitio.


```

const http = require("http");

const servidor = http.createServer((req, res) => {
  res.setHeader("Content-Type", "text/html; charset=utf-8");

  if (req.url === "/") {
    res.statusCode = 200;
    res.end("<h1>Bienvenido a mi sitio</h1><p>Esta es la pagina principal.</p>");
  } else if (req.url === "/acerca") {
    res.statusCode = 200;
    res.end("<h1>Acerca de</h1><p>Soy un servidor hecho con Node.js.</p>");
  } else if (req.url === "/contacto") {
    res.statusCode = 200;
    res.end("<h1>Contacto</h1><p>Escribeme a hola@misitio.com</p>");
  } else {
    res.statusCode = 404;
    res.end("<h1>404</h1><p>Esa pagina no existe.</p>");
  }
});

servidor.listen(3000, () => {
  console.log("Servidor corriendo en http://localhost:3000");
});

```

Lee el flujo: llega una petición, miras req.url, y según su valor respondes. La última rama (else) es la red de seguridad: si la URL no coincide con ninguna ruta conocida, respondes 404. Esa rama es OBLIGATORIA en todo servidor con rutas: sin ella, las URLs desconocidas se quedarían sin respuesta o con una respuesta incorrecta. Pruebalo: abre /, /acerca, /contacto y luego /algo-que-no-existe. Tres páginas distintas y un 404, todo con if/else que ya conoces de unidades anteriores. No hay magia nueva aquí: el enrutamiento es lógica de decisiones aplicada a URLs.

Fíjate en algo hermoso: acabas de construir, a mano, lo que los frameworks como Express hacen por dentro. Cuando después uses app.get("/", ...) en Express, por dentro es este mismo if sobre req.url, solo que más elegante y con más funciones. Entender la versión manual primero hace que la versión elegante tenga sentido en lugar de parecer magia. Así se aprende de verdad: primero a mano, después con herramientas.

Un patrón útil: normalizar la URL antes de comparar. Los navegadores a veces piden /acerca y a veces /acerca/ (con diagonal al final), y para tu if son cadenas distintas. Una técnica simple es quitar la diagonal final si existe, o comparar sin distinguir. Al inicio no te compliques, pero que sepas que existe el detalle. Otro patrón: usar un objeto como "mapa de rutas" en lugar de una cadena larga de if/else, cuando tengas muchas rutas:

```
const rutas = {
  "/": "<h1>Inicio</h1>",
  "/acerca": "<h1>Acerca de</h1>",
  "/contacto": "<h1>Contacto</h1>"
};

const servidor = http.createServer((req, res) => {
  res.setHeader("Content-Type", "text/html; charset=utf-8");
  if (rutas[req.url]) {
    res.statusCode = 200;
    res.end(rutas[req.url]);
  } else {
    res.statusCode = 404;
    res.end("<h1>404 - No encontrado</h1>");
  }
});
```

Mismo resultado, código más limpio cuando las rutas crecen. Cada ruta es una llave del objeto; si la URL pedida existe como llave, sirves su contenido; si no, 404. Este es el embrion de un "router", la pieza central de todos los frameworks web.

Sirviendo HTML de verdad: tu primera página completa

Responder "<h1>Hola</h1>" está bien para practicar, pero una página real es un documento HTML completo. Puedes armarlo como una cadena larga (usando template literals con backticks para escribirlo en varias líneas comodamente):

```

const http = require("http");

const pagina = `
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  <title>Mi primer servidor</title>
  <style>
    body { font-family: Arial; background: #f0f0f0; text-align: center; padding:
50px; }
    h1 { color: #2c3e50; }
    .tarjeta { background: white; padding: 20px; border-radius: 10px; display:
inline-block; }
  </style>
</head>
<body>
  <div class="tarjeta">
    <h1>Hola desde mi servidor Node.js</h1>
    <p>Esta pagina la genero mi propio codigo.</p>
    <p>Visita <a href="/acerca">Acerca de</a></p>
  </div>
</body>
</html>
`;

const servidor = http.createServer((req, res) => {
  res.writeHead(200, { "Content-Type": "text/html; charset=utf-8" });
  res.end(pagina);
});

servidor.listen(3000, () => {
  console.log("Servidor corriendo en http://localhost:3000");
});

```

Abrelo en el navegador y verás una página con estilos, no solo texto. Todo ese HTML viaja como una cadena desde tu servidor hasta el navegador. Y lo mejor: puedes GENERARLO dinámicamente. Cómo es solo una cadena de JavaScript, puedes meterle variables, fechas, datos del sistema:

```

const os = require("os");
const http = require("http");

const servidor = http.createServer((req, res) => {
  const pagina = `
    <!DOCTYPE html>
    <html lang="es">
    <head><meta charset="UTF-8"><title>Info del sistema</title></head>
    <body>
      <h1>Mi servidor corre en:</h1>
      <ul>
        <li>Maquina: ${os.hostname()}</li>
        <li>Sistema: ${os.platform()}</li>
        <li>Memoria total: ${os.totalmem() / 1024 / 1024 / 1024}.toFixed(2)} GB</li>
        <li>Hora del servidor: ${new Date().toLocaleString()}</li>
      </ul>
    </body>
    </html>`;

  res.writeHead(200, { "Content-Type": "text/html; charset=utf-8" });
  res.end(pagina);
});

servidor.listen(3000, () => {
  console.log("Servidor corriendo en http://localhost:3000");
});

```

Cada vez que alguien abre la página, el servidor ejecuta el código, consulta os, arma el HTML fresco con los datos ACTUALES y lo manda. Eso es una página DINÁMICA: no es un archivo estático guardado en disco, se genera al momento. Esta es la diferencia entre un folleto impreso (página estática) y un mesero que te cuenta el menu del día (página dinámica). Todo lo interesante de la web (redes sociales, tiendas, paneles) son páginas dinamicas: contenido generado al momento según datos. Y acabas de hacer una.

También puedes servir archivos HTML reales desde disco combinando http con fs, que ya conoces:

```
const http = require("http");
const fs = require("fs");

const servidor = http.createServer((req, res) => {
  fs.readFile("pagina.html", "utf8", (err, contenido) => {
    if (err) {
      res.writeHead(404, { "Content-Type": "text/plain; charset=utf-8" });
      res.end("Archivo no encontrado");
      return;
    }
    res.writeHead(200, { "Content-Type": "text/html; charset=utf-8" });
    res.end(contenido);
  });
});

servidor.listen(3000, () => {
  console.log("Servidor corriendo en http://localhost:3000");
});
```

Fíjate cómo se juntan las unidades: fs lee el archivo, http lo sirve. El callback de readFile decide la respuesta: si hubo error, 404; si no, 200 con el contenido. Este es el puente entre "programa que maneja archivos" y "servidor web": servir archivos estaticos (HTML, CSS, imagenes) es una de las tareas más comunes de un servidor, y ya sabes hacerla con piezas que conoces.

El método listen y el callback de confirmación: encendiendo el restaurante

Ya vimos listen en las cuatro líneas magicas, pero vamos a profundizar porque tiene detalles que importan. Su forma completa es:

```
servidor.listen(puerto, host, callback);
```

- puerto: el número de puerta (3000 en desarrollo). Obligatorio.
- host: en que dirección IP escucha. Si lo omites, escucha en todas las disponibles, incluyendo localhost. Casi siempre lo omites al inicio.
- callback: función que corre UNA vez cuando el servidor quedó encendido. Opcional pero MUY recomendado.

El callback de confirmación es tu mejor amigo al inicio. Sin el, corres el programa y no sabes si encendio o falló en silencio. Con el, ves el mensaje y tienes certeza. Hazlo un habito: todo listen lleva su console.log. Y pon la URL completa en el mensaje ("http://localhost:3000") para

poder copiarla y pegarla en el navegador. Los buenos hábitos al inicio te ahorran minutos todos los días después.

listen también puede recibir el puerto como variable de entorno, un patrón profesional que vas a ver en todas partes:

```
const PUERTO = process.env.PORT || 3000;
servidor.listen(PUERTO, () => {
  console.log("Servidor corriendo en el puerto " + PUERTO);
});
```

process.env.PORT es una variable que muchos servicios de hosting (donde subes tu app a internet) te dan para decirte "oye, aquí el puerto disponible es este". Con el || 3000 dices: "usa el que me den, y si no me dan ninguno (como en mi máquina), usa el 3000". Así el mismo código funciona en tu laptop y en el servidor de producción. Es una línea que separa al código de juguete del código desplegable. Ya conoces process de unidades anteriores; aquí ves otro de sus usos.

Errores comunes y como resolverlos: lo que te va a pasar (y no es grave)

Vamos a ver los tres errores clásicos del principiante con servidores, para que cuando te pasen los reconozcas en lugar de asustarte.

ERROR 1: EADDRINUSE, el puerto ocupado. El mensaje se ve así:

```
Error: listen EADDRINUSE: address already in use :::3000
```

Traducción: "quieres usar el puerto 3000 pero ya tiene dueño". Causas: dejaste corriendo otro servidor en otra terminal y se te olvidó, o el anterior no se detuvo bien, o dos programas quieren el mismo puerto. Soluciones, en orden: uno, busca la terminal donde corre el otro servidor y deténlo con Ctrl+C; dos, si no encuentras la terminal, cambia tu servidor a otro puerto (3001, 8080) y sigue trabajando; tres, en casos extremos, busca que proceso usa el puerto y terminalo (en Linux/macOS: lsof -i :3000 y luego kill; en Windows: netstat -ano y taskkill). Este error te va a pasar MUCHAS veces en tu carrera. Ya no te va a asustar: es solo "la puerta ya está ocupada".

ERROR 2: cambias el código y no ves los cambios. Modificas el HTML, recargas el navegador, y ves lo mismo de antes. ¿Qué paso? Node.js NO recarga tu código solo: el servidor que está corriendo usa la versión del código de cuando lo ENCENDISTE. Los cambios que haces después no existen para él hasta que lo reinicies. Solución: Ctrl+C para detener, node

servidor.js para encender de nuevo, recargar el navegador. Este ciclo (editar -> detener -> encender -> probar) lo vas a repetir cientos de veces. Existen herramientas que lo automatizan (nodemon reinicia solo cuando detecta cambios; lo verás cuando hablemos de herramientas de desarrollo), pero entiende primero el ciclo manual: te enseña que el servidor corre una "foto" de tu código, no el código vivo.

ERROR 3: olvidar `res.end` y el navegador "colgado". Tu servidor recibe la petición, hace algo, pero nunca llama `res.end`. El navegador se queda con la ruedita girando eternamente, "cargando...". No hay mensaje de error en ningún lado, solo silencio y espera. Diagnóstico: revisa TODOS los caminos de tu callback y asegúrate de que cada uno termina en `res.end`. El caso típico es un `if` sin `else`: atiendes dos rutas pero se te olvida la tercera, y cuando piden la tercera no hay respuesta. La rama `else` con el 404 no es solo buena práctica: es tu seguro contra el colgado. Regla de oro: cada petición que entra, sale con un `res.end`. Sin excepciones.

ERROR BONUS: el error dentro del callback. Si tu código dentro de `createServer` lanza una excepción no capturada (por ejemplo, lees una variable que no existe), el servidor se CAE: el programa termina y la terminal regresa. No es el fin del mundo (lo reinicias), pero en producción es grave. La defensa es `try/catch` alrededor del código riesgoso y, sobre todo, probar bien antes. Cuando el servidor "se apaga solo", lo primero que miras es la terminal: ahí está el mensaje de error con la línea culpable. La terminal es la caja negra del servidor: todo lo que le pasa queda registrado ahí.

Buenas prácticas: los hábitos del buen servidor

Cierra la explicación con hábitos que distinguen un servidor de juguete de uno profesional. No son reglas complicadas; son disciplina.

Uno: siempre responde algo a cada petición. Cada camino del código termina en `res.end`. La rama `else` con 404 es obligatoria, no opcional.

Dos: el código de estado debe decir la verdad. 200 para éxito, 404 para no encontrado, 500 para errores tuyos. No devuelvas 200 con un mensaje de error adentro: confundes a todos los que consuman tu servidor.

Tres: fija el `Content-Type` correcto siempre, con `charset=utf-8` en textos. Es una línea y evita una categoría entera de problemas (acentos rotos, navegadores confundidos).

Cuatro: usa el callback de confirmación en `listen` con la URL completa. Tu yo del futuro te lo va a agradecer cada vez que enciendas el servidor.

Cinco: el puerto como variable de entorno con valor por defecto (`process.env.PORT || 3000`). Prepara tu código para salir de tu máquina desde el día uno.

Seis: separa la lógica en funciones. En lugar de un callback gigante con todo el HTML adentro, crea funciones como `paginaInicio()`, `pagina404()`, `manejarRuta(req, res)`. El callback queda corto y legible; las funciones se pueden probar por separado. Es la lección de la unidad de funciones aplicada a servidores.

Siete: no pongas secretos en el código del servidor (claves, tokens). Al inicio no tendrás secretos, pero forma el hábito desde ya: los secretos van en variables de entorno, nunca escritos en el código. Cuando subas tu código a internet, todo lo escrito ahí se vuelve visible.

Ocho: registra lo importante con `console.log`. Cada petición importante, cada error, cada evento raro: un log con fecha y, si tienes varios servidores, con hostname. Los logs son los ojos del servidor cuando tu no estas mirando.

Con esto tienes la teoría completa: sabes que máquina tienes (os), sabes como hablan cliente y servidor (HTTP), sabes por donde entran (puertos), sabes construir respuestas (`req/res`), sabes el idioma de los códigos y contenidos, sabes enrutar, sabes encender y sabes que errores te esperan. Ahora vamos a los ejemplos completos, donde todo esto se vuelve código que puedes correr.

Ejemplos paso a paso

Ejemplo 1: tu primer servidor "hola mundo"

El ritual de iniciación. Un servidor que responde "Hola mundo" en texto simple. Copialo, correlo, ábrelo en el navegador.

Paso 1: crea el archivo `servidor-hola.js` con este contenido:


```
const http = require("http");

const servidor = http.createServer((req, res) => {
  res.writeHead(200, { "Content-Type": "text/plain; charset=utf-8" });
  res.end("Hola mundo desde mi primer servidor");
});

const PUERTO = 3000;
servidor.listen(PUERTO, () => {
  console.log("Servidor corriendo en http://localhost:" + PUERTO);
});
```

Paso 2: en la terminal, corre `node servidor-hola.js`. Debes ver "Servidor corriendo en `http://localhost:3000`". Si ves `EADDRINUSE`, otro programa tiene el 3000: deténlo o cambia a 3001.

Paso 3: abre `http://localhost:3000` en tu navegador. Verás el texto "Hola mundo desde mi primer servidor", sin formato, como bloc de notas. Eso es `text/plain`.

Paso 4: prueba pedir otra URL, como `http://localhost:3000/loquesea`. Verás EL MISMO texto, porque este servidor no tiene rutas: responde igual a todo. En el ejemplo 2 arreglamos eso.

Paso 5: regresa a la terminal y presiona `Ctrl+C`. El mensaje de confirmación desaparece y recuperas tu terminal: el servidor se apaga. Si recargas el navegador ahora, verás "no se puede conectar": ya no hay nadie escuchando en el 3000.

Que aprendiste: el ciclo completo de vida de un servidor (crear, encender, probar, detener), y que sin rutas todo responde igual. Es poco, pero es TODO el mecanismo: lo demás son variaciones.

Ejemplo 2: servidor con 3 rutas y página 404

Ahora un mini sitio de verdad: página principal, acerca de, contacto, y un 404 para lo desconocido. Cada ruta con su propio HTML.

Paso 1: crea el archivo `mini-sitio.js`:

```
const http = require("http");

function pagina(titulo, contenido) {
  return `
<!DOCTYPE html>
<html lang="es">
<head>
```

```

    <meta charset="UTF-8">
    <title>${titulo}</title>
    <style>
      body { font-family: Arial, sans-serif; max-width: 600px; margin: 40px auto;
padding: 20px; }
      nav a { margin-right: 15px; }
      h1 { color: #2c3e50; }
    </style>
  </head>
  <body>
    <nav>
      <a href="/">Inicio</a>
      <a href="/acerca">Acerca de</a>
      <a href="/contacto">Contacto</a>
    </nav>
    <hr>
    <h1>${titulo}</h1>
    <p>${contenido}</p>
  </body>
</html>`;
}

```

```

const servidor = http.createServer((req, res) => {
  res.setHeader("Content-Type", "text/html; charset=utf-8");

  if (req.url === "/") {
    res.statusCode = 200;
    res.end(pagina("Bienvenido", "Esta es la pagina principal de mi mini sitio hecho
con Node.js."));
  } else if (req.url === "/acerca") {
    res.statusCode = 200;
    res.end(pagina("Acerca de", "Soy un servidor creado a mano con el modulo http. Sin
frameworks."));
  } else if (req.url === "/contacto") {
    res.statusCode = 200;
    res.end(pagina("Contacto", "Escribeme a hola@misitio.com y te respondo pronto."));
  } else {
    res.statusCode = 404;
    res.end(pagina("404 - No encontrado", "La pagina '" + req.url + "' no existe en
este sitio."));
  }
});

servidor.listen(3000, () => {
  console.log("Mini sitio corriendo en http://localhost:3000");
}

```

```
});
```

Paso 2: correlo con node mini-sitio.js y abre <http://localhost:3000>. Verás una página con menu de navegación funcional: los enlaces llevan a /acerca y /contacto porque son etiquetas <a> normales apuntando a tus rutas.

Paso 3: navega con el menu. Cada página tiene el mismo diseno (gracias a la función página que genera el HTML) pero distinto título y contenido. Fíjate como la función página evita repetir el esqueleto HTML tres veces: es la lección de reutilización aplicada.

Paso 4: escribe una URL inventada, como <http://localhost:3000/secreto>. Verás la página 404 que ADEMÁS te dice que URL pediste (req.url dentro del mensaje). Es un 404 útil, no solo un número frío.

Paso 5 (opcional, para curiosos): abre las herramientas de desarrollo del navegador (F12), ve a la pestaña Red (Network), recarga la página y haz clic en la petición. Ahí verás el código de estado 200, los encabezados (incluido tu Content-Type) y el cuerpo HTML. Para el 404 verás el código 404. Es la mejor forma de VER la conversación HTTP con tus propios ojos.

Que aprendiste: enrutamiento real con if/else, la rama 404 obligatoria, generar HTML con funciones reutilizables, y a inspeccionar la conversación HTTP en el navegador. Esto ya es un sitio web funcional, pequeno pero completo.

Ejemplo 3: servidor que responde JSON (tu primera API)

Las APIs son servidores que en lugar de páginas devuelven DATOS en JSON, para que otros programas (apps de celular, otras páginas, tu propio frontend) los consuman. Vamos a hacer una mini API de usuarios.

Paso 1: crea el archivo api-usuarios.js:

```

const http = require("http");

const usuarios = [
  { id: 1, nombre: "Ana", edad: 25, ciudad: "Ciudad de Mexico" },
  { id: 2, nombre: "Luis", edad: 30, ciudad: "Guadalajara" },
  { id: 3, nombre: "Maria", edad: 28, ciudad: "Monterrey" }
];

const servidor = http.createServer((req, res) => {
  res.setHeader("Content-Type", "application/json; charset=utf-8");

  if (req.url === "/api/usuarios" && req.method === "GET") {
    res.statusCode = 200;
    res.end(JSON.stringify(usuarios));
  } else if (req.url === "/api/usuarios/total" && req.method === "GET") {
    res.statusCode = 200;
    res.end(JSON.stringify({ total: usuarios.length }));
  } else {
    res.statusCode = 404;
    res.end(JSON.stringify({ error: "Ruta no encontrada" }));
  }
});

servidor.listen(3000, () => {
  console.log("API corriendo en http://localhost:3000");
});

```

Paso 2: correlo y abre <http://localhost:3000/api/usuarios>. Verás el arreglo de usuarios en JSON. El navegador lo muestra como texto, pero es DATOS estructurados: un programa podría leerlos y usarlos.

Paso 3: abre <http://localhost:3000/api/usuarios/total>. Verás `{"total":3}`. Una ruta que no devuelve datos crudos sino un CALCULO sobre los datos. Así funcionan las APIs reales: algunas rutas dan datos, otras dan resúmenes.

Paso 4: abre una ruta inventada, como <http://localhost:3000/api/perros>. Verás `{"error":"Ruta no encontrada"}` con código 404. Fíjate que hasta el error viene en JSON: en una API, TODO es JSON, incluyendo los errores, para que el programa cliente siempre sepa que formato esperar. Esa consistencia es una buena práctica de APIs.

Paso 5 (para que veas la diferencia con HTML): compara este ejemplo con el anterior. La estructura es IDENTICA (createServer, if sobre req.url, 404 al final). Lo único que cambió es el Content-Type y el contenido (JSON en lugar de HTML). Un servidor es un servidor: lo que

cambia es QUE sirves. Cuando alguien dice "voy a hacer una API", significa "voy a hacer un servidor que sirve JSON en lugar de páginas".

Que aprendiste: qué es una API en su forma más simple, JSON.stringify para convertir objetos a JSON, que los errores también van en el formato de la API, y que puedes filtrar por método (req.method === "GET") además de por URL. Este ejemplo es el embrión de todo backend que construyas después.

Ejemplo 4: servidor que muestra información del sistema (os + http juntos)

El ejemplo que une las dos mitades de la unidad: el módulo os recolecta los datos, el módulo http los sirve como página web. Un mini panel de monitoreo del sistema.

Paso 1: crea el archivo panel-sistema.js:

```
const http = require("http");
const os = require("os");

function bytesAGB(bytes) {
  return (bytes / 1024 / 1024 / 1024).toFixed(2);
}

function formatearUptime(segundos) {
  const dias = Math.floor(segundos / 86400);
  const horas = Math.floor((segundos % 86400) / 3600);
  const minutos = Math.floor((segundos % 3600) / 60);
  return dias + "d " + horas + "h " + minutos + "m";
}

const servidor = http.createServer((req, res) => {
  res.setHeader("Content-Type", "text/html; charset=utf-8");

  if (req.url === "/") {
    const total = os.totalmem();
    const libre = os.freemem();
    const usada = total - libre;
    const porcentaje = ((usada / total) * 100).toFixed(1);

    const pagina = `
    <!DOCTYPE html>
    <html lang="es">
    <head>
      <meta charset="UTF-8">
      <title>Panel del sistema</title>
      <style>
```

```

    body { font-family: Arial; max-width: 600px; margin: 40px auto; padding: 20px;
background: #f5f5f5; }
    .dato { background: white; padding: 12px; margin: 8px 0; border-radius: 8px; }
    .etiqueta { font-weight: bold; color: #555; }
</style>
</head>
<body>
    <h1>Panel del sistema</h1>
    <div class="dato"><span class="etiqueta">Maquina:</span> ${os.hostname()}</div>
        <div class="dato"><span class="etiqueta">Sistema operativo:</span>
${os.platform()} (${os.arch()})</div>
            <div class="dato"><span class="etiqueta">Usuario:</span>
${os.userInfo().username}</div>
                <div class="dato"><span class="etiqueta">Memoria total:</span>
${bytesAGB(total)} GB</div>
                    <div class="dato"><span class="etiqueta">Memoria libre:</span>
${bytesAGB(libre)} GB</div>
                        <div class="dato"><span class="etiqueta">Memoria usada:</span>
${porcentaje}%</div>
                            <div class="dato"><span class="etiqueta">Procesadores:</span>
${os.cpus().length} nucleos (${os.cpus()[0].model})</div>
                                <div class="dato"><span class="etiqueta">Encendida hace:</span>
${formatearUptime(os.uptime())}</div>
                                    <p><a href="/json">Ver estos datos en JSON</a></p>
</body>
</html>`;

res.statusCode = 200;
res.end(pagina);
} else if (req.url === "/json") {
res.setHeader("Content-Type", "application/json; charset=utf-8");
res.statusCode = 200;
res.end(JSON.stringify({
    hostname: os.hostname(),
    plataforma: os.platform(),
    arquitectura: os.arch(),
    memoriaTotalGB: bytesAGB(os.totalmem()),
    memoriaLibreGB: bytesAGB(os.freemem()),
    nucleos: os.cpus().length,
    uptimeSegundos: os.uptime()
}));
} else {
res.statusCode = 404;
res.end("<h1>404</h1><p>Ruta no encontrada. Prueba <a href='/'>el
panel</a>.</p>");

```

```
    }  
  });  
  
  servidor.listen(3000, () => {  
    console.log("Panel corriendo en http://localhost:3000");  
  });
```

Paso 2: correlo y abre <http://localhost:3000>. Verás un panel con los datos REALES de tu máquina: su nombre, su sistema, su memoria, sus procesadores, cuanto lleva encendida. Todo generado al momento por tu código.

Paso 3: abre <http://localhost:3000/json>. Verás LOS MISMOS datos pero en JSON. Un solo servidor, dos formatos: HTML para humanos, JSON para programas. Así trabajan los sistemas reales: la misma información servida según quien la pida.

Paso 4: deja el servidor corriendo unos minutos, abre otra cosa pesada (un video, un juego), y recarga el panel. Verás como cambian la memoria libre y el porcentaje usado. Tu panel esta VIVO: refleja la máquina en tiempo real porque los datos se consultan en cada petición, no una vez al inicio.

Que aprendiste: a combinar módulos (os + http), a generar páginas dinamicas con datos reales, a servir el mismo contenido en dos formatos, y la idea de un panel de monitoreo, que es una aplicación real y útil. Este ejemplo junta TODA la unidad en un solo programa.

Lo que los 4 ejemplos te enseñaron (el patrón universal)

Miralo bien: los cuatro ejemplos tienen el MISMO esqueleto. Importar http, createServer con un callback, if/else sobre req.url adentro, res.end en cada rama, listen con confirmación. Lo único que cambia es QUE respondes: texto simple, páginas HTML, JSON, datos del sistema. Ese esqueleto es el patrón universal del servidor Node.js básico, y ya lo tienes en las manos. Cuando después aprendas Express, vas a reconocer este esqueleto debajo de su sintaxis elegante, y vas a entender qué hace cada parte porque tu ya la hiciste a mano.

Práctica

Ejercicio 1: detective del sistema

Escribe un programa (NO un servidor, un script normal) que imprima un reporte completo de la máquina: sistema operativo, arquitectura, nombre de la máquina, usuario actual, carpeta personal, memoria total y libre en GB con porcentaje usado, número de nucleos y modelo del procesador, y tiempo encendida en formato días/horas/minutos. Usa solo el módulo os.

SOLUCIÓN explicada paso a paso:

Paso 1: importa os. Todo sale de ahí.

Paso 2: para la memoria, convierte bytes a GB dividiendo entre 1024 tres veces, y calcula el porcentaje usado como $(usada / total) * 100$. Usa `toFixed(2)` para GB y `toFixed(1)` para el porcentaje, para que se lea limpio.

Paso 3: para el uptime, convierte segundos a días, horas y minutos con divisiones y `Math.floor`, como vimos en la explicación.

Paso 4: el modelo del procesador está en `os.cpus()[0].model` (todos los nucleos suelen ser del mismo modelo, basta el primero).

Paso 5: imprime todo con etiquetas claras. El orden lógico: identificación de la máquina primero (quien es), recursos después (que tiene), tiempo al final.

```
const os = require("os");

function bytesAGB(bytes) {
  return (bytes / 1024 / 1024 / 1024).toFixed(2);
}

const total = os.totalmem();
const libre = os.freemem();
const usada = total - libre;

const seg = os.uptime();
const dias = Math.floor(seg / 86400);
const horas = Math.floor((seg % 86400) / 3600);
const minutos = Math.floor((seg % 3600) / 60);

console.log("=== REPORTE DEL SISTEMA ===");
console.log("Sistema operativo: " + os.platform());
console.log("Arquitectura: " + os.arch());
console.log("Máquina: " + os.hostname());
console.log("Usuario: " + os.userInfo().username);
console.log("Carpeta personal: " + os.userInfo().homedir);
console.log("Memoria total: " + bytesAGB(total) + " GB");
console.log("Memoria libre: " + bytesAGB(libre) + " GB");
console.log("Memoria usada: " + ((usada / total) * 100).toFixed(1) + "%");
console.log("Núcleos: " + os.cpus().length);
console.log("Procesador: " + os.cpus()[0].model);
console.log("Encendida hace: " + dias + "d " + horas + "h " + minutos + "m");
```


Correlo en tu máquina y luego, si puedes, en otra (la de un amigo, una virtual). Compara los reportes: ahí ves con tus ojos por qué un programa profesional pregunta en lugar de asumir. Cada máquina cuenta una historia distinta.

Ejercicio 2: servidor de frases motivadoras

Crea un servidor que tenga un arreglo de 5 frases motivadoras en español. Debe tener estas rutas: / muestra una página HTML bonita con una frase ALEATORIA cada vez que se recarga; /frases muestra la lista completa en HTML; /api/frase devuelve una frase aleatoria en JSON con formato { "frase": "..." }; cualquier otra ruta devuelve 404 con una página HTML amable. Usa el puerto 3000.

SOLUCIÓN explicada paso a paso:

Paso 1: el arreglo de frases vive FUERA del callback (es configuración, se crea una vez). La elección aleatoria vive DENTRO (se hace en cada petición): `Math.floor(Math.random() * frases.length)` te da un índice al azar.

Paso 2: la ruta / genera el HTML con la frase elegida metida en el template. Cada recarga del navegador = una petición nueva = una frase nueva. Así se siente la "dinámica" de la página.

Paso 3: /frases usa `frases.map` para convertir cada frase en un `<li>` y `join("")` para unirlos. Es una forma elegante de generar listas HTML desde arreglos.

Paso 4: /api/frase cambia el Content-Type a `application/json` y manda `JSON.stringify({ frase: elegida })`. Mismo dato, otro formato.

Paso 5: el else final con 404. Nunca lo olvides.

```
const http = require("http");

const frases = [
  "El que persevera, alcanza.",
  "Haz hoy lo que otros no quieren y mañana vivirás lo que otros no pueden.",
  "No cuentes los días, haz que los días cuenten.",
  "El éxito es la suma de pequeños esfuerzos repetidos cada día.",
  "Cree en ti y todo será posible."
];

function fraseAlAzar() {
  const indice = Math.floor(Math.random() * frases.length);
  return frases[indice];
}
```

```

const servidor = http.createServer((req, res) => {
  if (req.url === "/") {
    res.writeHead(200, { "Content-Type": "text/html; charset=utf-8" });
    res.end(`
      <!DOCTYPE html>
      <html lang="es">
      <head><meta charset="UTF-8"><title>Frase del dia</title>
      <style>body{font-family:Arial;text-align:center;padding:60px;background:#eef;}
      blockquote{font-size:28px;color:#2c3e50;}</style></head>
      <body><h1>Tu frase motivadora</h1>
      <blockquote>"${fraseAlAzar()}"</blockquote>
      <p><a href="/">Otra frase</a> | <a href="/frases">Ver todas</a></p>
      </body></html>`);
  } else if (req.url === "/frases") {
    res.writeHead(200, { "Content-Type": "text/html; charset=utf-8" });
    const lista = frases.map(f => "<li>" + f + "</li>").join("");
    res.end("<!DOCTYPE html><html lang='es'><head><meta charset='UTF-8'>" +
      "<title>Todas las frases</title></head><body><h1>Todas las frases</h1><ul>" +
      lista + "</ul><a href='/'>Volver</a></body></html>");
  } else if (req.url === "/api/frase") {
    res.writeHead(200, { "Content-Type": "application/json; charset=utf-8" });
    res.end(JSON.stringify({ frase: fraseAlAzar() }));
  } else {
    res.writeHead(404, { "Content-Type": "text/html; charset=utf-8" });
    res.end("<h1>404</h1><p>Aquí no hay frases. Prueba <a href='/'>el inicio</a>.</p>");
  }
});

servidor.listen(3000, () => {
  console.log("Servidor de frases en http://localhost:3000");
});

```

Pruebalo recargando / varias veces: cada recarga trae una frase distinta. Ese es el poder de generar al momento. Y abre /api/frase: la misma frase aleatoria, pero en JSON lista para que una app la consuma.

Ejercicio 3: servidor con método POST (recibir datos)

Hasta ahora todos los ejemplos usan GET (pedir). En este ejercicio el servidor también RECIBE: tendrá un formulario HTML en / (método GET que muestra el formulario) y una ruta /saludar que recibe el POST con el nombre y responde "Hola, [nombre]". Vas a aprender a leer el cuerpo de una petición, que llega POR PARTES.

SOLUCIÓN explicada paso a paso:

Paso 1: en la ruta / con GET, sirve un formulario HTML con `method="POST"` y `action="/saludar"`, con un campo de texto llamado "nombre". Cuando el usuario lo envía, el navegador manda un POST a /saludar con los datos.

Paso 2: en /saludar, el cuerpo NO llega de golpe: llega en pedazos (chunks) por el evento "data" de req. Los vas acumulando en una variable: `req.on("data", pedazo => cuerpo += pedazo)`.

Paso 3: cuando termina de llegar (evento "end"), parseas el cuerpo. Un formulario simple llega como "nombre=Ana&edad=25". Para este ejercicio, extrae el nombre buscando después de "nombre=" y decodificando con `decodeURIComponent` (convierte los %20 en espacios y los acentos codificados).

Paso 4: responde el saludo personalizado. IMPORTANTE: el `res.end` va DENTRO del evento "end", porque solo ahí ya tienes todos los datos. Si respondes antes, el cuerpo aun no llega completo.

Paso 5: verifica también el método: si alguien abre /saludar con GET (escribiendo la URL a mano), no hay datos que leer; redirígelo al formulario o muestra un mensaje.

```

const http = require("http");

const servidor = http.createServer((req, res) => {
  if (req.url === "/" && req.method === "GET") {
    res.writeHead(200, { "Content-Type": "text/html; charset=utf-8" });
    res.end(`
<!DOCTYPE html>
<html lang="es">
<head><meta charset="UTF-8"><title>Saludame</title></head>
<body>
  <h1>Como te llamas?</h1>
  <form method="POST" action="/saludar">
    <input type="text" name="nombre" placeholder="Tu nombre" required>
    <button type="submit">Saludar</button>
  </form>
</body>
</html>`);
  } else if (req.url === "/saludar" && req.method === "POST") {
    let cuerpo = "";
    req.on("data", (pedazo) => {
      cuerpo += pedazo;
    });
    req.on("end", () => {
      // El cuerpo viene como "nombre=Ana"; extraemos el valor
      const nombreCodificado = cuerpo.split("nombre=")[1] || "";
      const nombre = decodeURIComponent(nombreCodificado.replace(/\+/g, " "));
      res.writeHead(200, { "Content-Type": "text/html; charset=utf-8" });
      res.end("<h1>Hola, " + nombre + "!</h1><p><a href='/'>Volver</a></p>");
    });
  } else {
    res.writeHead(404, { "Content-Type": "text/html; charset=utf-8" });
    res.end("<h1>404</h1><p>Ruta no encontrada.</p>");
  }
});

servidor.listen(3000, () => {
  console.log("Servidor con formulario en http://localhost:3000");
});

```

Pruebalo: abre /, escribe tu nombre, envía, y el servidor te saluda. Acabas de completar el ciclo web fundamental: mostrar formulario (GET) -> recibir datos (POST) -> responder personalizado. Casi todo lo interactivo de la web (registros, comentarios, búsquedas) es una variación de este ciclo. El detalle clave que debes recordar: el cuerpo llega por partes y se lee con eventos; la respuesta se manda cuando el "end" confirma que llegó todo.

Ejercicio 4: servidor de archivos estaticos

Crea un servidor que sirva archivos HTML desde una carpeta llamada "público". Si piden /, sirve público/index.html. Si piden /página, sirve público/página.html. Si el archivo no existe, responde 404. Si piden cualquier otra cosa que no sea HTML (por seguridad, solo permite archivos .html), responde 403 Forbidden. Usa fs.readFile y path.join.

SOLUCIÓN explicada paso a paso:

Paso 1: crea la carpeta público con dos archivos: index.html ("Bienvenido") y página.html ("Otra página"). El servidor los lee del disco, no los trae escritos en el código.

Paso 2: traduce la URL a archivo: "/" -> "index.html", "/página" -> "página.html". La regla general: quita la diagonal inicial y agrega ".html". Pero OJO con la seguridad: un atacante podría pedir "../secretos/clave" para salirse de la carpeta. La defensa de este ejercicio: solo acepta nombres sin diagonales ni puntos raros; si la URL contiene ".." o no termina en algo esperado, responde 403.

Paso 3: usa path.join(__dirname, "público", archivo) para construir la ruta de forma segura en cualquier sistema (__dirname es la carpeta donde vive tu script).

Paso 4: fs.readFile con callback: si hay error, 404; si no, 200 con el contenido y Content-Type text/html. Recuerda: el res.end va DENTRO del callback de readFile, porque la lectura es asincrona.

```

const http = require("http");
const fs = require("fs");
const path = require("path");

const CARPETA = path.join(__dirname, "publico");

const servidor = http.createServer((req, res) => {
  // Traducir URL a nombre de archivo
  let nombre = req.url === "/" ? "index.html" : req.url.slice(1) + ".html";

  // Seguridad basica: nada de ".." ni diagonales en el nombre
  if (nombre.includes("..") || nombre.includes("/") || nombre.includes("\\")) {
    res.writeHead(403, { "Content-Type": "text/plain; charset=utf-8" });
    res.end("403 - Prohibido");
    return;
  }

  const rutaArchivo = path.join(CARPETA, nombre);

  fs.readFile(rutaArchivo, "utf8", (err, contenido) => {
    if (err) {
      res.writeHead(404, { "Content-Type": "text/html; charset=utf-8" });
      res.end("<h1>404</h1><p>El archivo '" + nombre + "' no existe.</p>");
      return;
    }
    res.writeHead(200, { "Content-Type": "text/html; charset=utf-8" });
    res.end(contenido);
  });
});

servidor.listen(3000, () => {
  console.log("Servidor de archivos en http://localhost:3000");
});

```

Que aprendiste: a servir archivos reales del disco (así funcionan los servidores de páginas estaticas), a traducir URLs a rutas de archivo, y tu primera lección de SEGURIDAD web: nunca confíes en lo que pide el cliente sin validarlo. El chequeo de ".." es simple pero real: sin el, cualquiera podría leer archivos fuera de tu carpeta publica. La seguridad empieza validando entradas, y las URLs son entradas.

Ejercicio 5: mini API REST de tareas (junta todo)

El ejercicio final junta toda la unidad: un servidor que es una API de lista de tareas con varias rutas y métodos. Debe cumplir: GET /api/tareas devuelve todas las tareas en JSON; GET /api/tareas/total devuelve { "total": N, "pendientes": M }; POST /api/tareas agrega una tarea que viene en el cuerpo como JSON { "texto": "..." } y devuelve la tarea creada con código 201; cualquier otra ruta o método devuelve 404 en JSON. Las tareas viven en un arreglo en memoria (se pierden al reiniciar, y está bien para este ejercicio).

SOLUCIÓN explicada paso a paso:

Paso 1: el arreglo de tareas con 2 tareas de ejemplo, cada una con id, texto y hecha (booleano). Un contador para el siguiente id.

Paso 2: GET /api/tareas: responde el arreglo completo. Es el ejemplo 3 aplicado.

Paso 3: GET /api/tareas/total: calcula con filter cuantas tienen hecha === false. Es lógica de JavaScript normal dentro del servidor: el servidor puede calcular lo que sea antes de responder.

Paso 4: POST /api/tareas: lee el cuerpo por partes (como en el ejercicio 3), pero ahora el cuerpo es JSON, así que lo conviertes con JSON.parse dentro de un try/catch (si el cliente manda JSON mal formado, responde 400 Bad Request en lugar de caerse). Crea la tarea con el siguiente id, la agrega al arreglo, responde 201 con la tarea creada. El 201 dice "todo bien y cree algo nuevo": es el código correcto para creaciones.

Paso 5: todo lo demás (rutas desconocidas, métodos no soportados como PUT o DELETE) cae al else con 404 en JSON. Consistencia: en una API, hasta los errores son JSON.

```
const http = require("http");

let tareas = [
  { id: 1, texto: "Aprender el modulo os", hecha: true },
  { id: 2, texto: "Crear mi primer servidor", hecha: false }
];
let siguienteId = 3;

const servidor = http.createServer((req, res) => {
  res.setHeader("Content-Type", "application/json; charset=utf-8");

  if (req.url === "/api/tareas" && req.method === "GET") {
    res.statusCode = 200;
    res.end(JSON.stringify(tareas));
  } else if (req.url === "/api/tareas/total" && req.method === "GET") {
```

```

    const pendientes = tareas.filter(t => !t.hecha).length;
    res.statusCode = 200;
    res.end(JSON.stringify({ total: tareas.length, pendientes: pendientes }));
  } else if (req.url === "/api/tareas" && req.method === "POST") {
    let cuerpo = "";
    req.on("data", (pedazo) => { cuerpo += pedazo; });
    req.on("end", () => {
      try {
        const datos = JSON.parse(cuerpo);
        if (!datos.texto) {
          res.statusCode = 400;
          res.end(JSON.stringify({ error: "Falta el campo 'texto'" }));
          return;
        }
        const nueva = { id: siguienteId++, texto: datos.texto, hecha: false };
        tareas.push(nueva);
        res.statusCode = 201;
        res.end(JSON.stringify(nueva));
      } catch (e) {
        res.statusCode = 400;
        res.end(JSON.stringify({ error: "El cuerpo no es JSON valido" }));
      }
    });
  } else {
    res.statusCode = 404;
    res.end(JSON.stringify({ error: "Ruta o metodo no encontrado" }));
  }
});

servidor.listen(3000, () => {
  console.log("API de tareas en http://localhost:3000");
});

```

Para probar el POST necesitas mandar JSON en el cuerpo. La forma más fácil sin instalar nada: abre la consola de desarrollo del navegador (F12) y ejecuta:

```

fetch("http://localhost:3000/api/tareas", {
  method: "POST",
  headers: { "Content-Type": "application/json" },
  body: JSON.stringify({ texto: "Probar mi API" })
}).then(r => r.json()).then(console.log);

```

Verás la tarea creada con su id. Luego recarga GET /api/tareas y ahí estara. Acabas de hacer un CRUD parcial (crear y leer) sobre una API REST hecha a mano. Las letras que faltan

(actualizar y borrar) usan PUT y DELETE con la misma técnica. Cuando llegues a Express y a bases de datos, este ejercicio te va a parecer familiar: es el mismo patrón, con mejores herramientas.

Resumen de la unidad

Llegaste al final de una de las unidades más importantes del curso. Vamos a amarrar todo lo que aprendiste, porque es mucho y todo se conecta.

El módulo `os` es la ventanilla de información de la máquina: `os.platform()` te dice el sistema operativo, `os.arch()` la arquitectura del procesador, `os.totalmem()` y `os.freemem()` la memoria en bytes, `os.cpus()` el arreglo de procesadores (su `.length` es el número de núcleos), `os.hostname()` el nombre de la máquina, `os.uptime()` los segundos encendida, `os.userInfo()` los datos del usuario actual (incluida su carpeta personal en `homedir`), y `os.EOL` el fin de línea correcto para cada sistema. La filosofía que los une: un programa profesional PREGUNTA en lugar de asumir, y así funciona en cualquier máquina, no solo en la tuya.

HTTP es el protocolo de la web: un modelo de petición-respuesta donde el cliente pide y el servidor responde. Cada petición trae un método (GET para pedir, POST para enviar), una URL (que cosa quiere), encabezados (metadatos) y a veces un cuerpo (datos). Cada respuesta trae un código de estado (como le fue), encabezados (el más importante: `Content-Type`) y un cuerpo (lo pedido). HTTP no tiene memoria: cada petición es independiente. La analogía que te acompaña: el restaurante, con clientes que piden, meseros que atienden y cocina que prepara.

Los puertos son las puertas numeradas del edificio que es tu computadora: la IP dice cual edificio, el puerto dice cual programa. Los conocidos: 80 para HTTP, 443 para HTTPS, 3306 para MySQL, entre otros. En desarrollo usamos el 3000 por convención. `localhost` significa "esta misma computadora". Un servidor "escucha" en un puerto y mientras corre, ese puerto es suyo: dos programas no pueden compartirlo (error `EADDRINUSE`).

El módulo `http` crea servidores con cuatro líneas: `require("http")`, `http.createServer(callback)` donde el callback recibe (`req`, `res`) y se ejecuta una vez por petición, y `servidor.listen(puerto, callbackDeConfirmacion)` que enciende el servidor. El callback de confirmación con la URL completa es un hábito obligatorio. El puerto puede venir de `process.env.PORT || 3000` para que el código funcione aquí y en producción. Y recuerda: un servidor no termina solo; se queda vivo hasta que lo detienes con `Ctrl+C`.

El objeto `req` describe lo que pidió el cliente: `req.url` (que ruta), `req.method` (GET, POST...), `req.headers` (metadatos como el navegador). El objeto `res` construye tu respuesta en orden:

encabezados primero (`res.writeHead` o `res.setHeader`), contenido después (`res.write` una o varias veces), y `res.end` SIEMPRE al final para cerrar. Cada petición que entra sale con un `res.end`: sin excepciones.

Los códigos de estado son el semáforo: 2xx éxito (200 todo bien, 201 creado), 3xx redirección (301 se mudo), 4xx error del cliente (404 no encontrado, 400 petición mal formada), 5xx error del servidor (500 se rompió algo mío). El código debe decir la verdad siempre. El `Content-Type` dice que tipo de contenido mandas: `text/plain` para texto simple, `text/html` para páginas, `application/json` para datos; con `charset=utf-8` para que los acentos se vean bien.

El enrutamiento es un `if/else` sobre `req.url`: cada URL responde diferente, y la rama `else` final con 404 es obligatoria. Así nacen las rutas de tu sitio, y así funcionan por dentro todos los frameworks: cuando uses Express después, reconocerás este patrón debajo de su sintaxis. Puedes servir HTML armado como cadena (páginas dinámicas generadas al momento con datos reales) o leer archivos del disco con `fs` (páginas estáticas). Combinar os con `http` te da paneles de monitoreo reales.

Los tres errores que ya no te asustan: `EADDRINUSE` (puerto ocupado: detiene el otro servidor o cambia de puerto), cambios que no se ven (Node.js no recarga solo: `Ctrl+C` y vuelve a encender), y el navegador colgado (olvidaste `res.end` en algún camino: revisa que cada rama responda). Y las buenas prácticas: responde siempre, códigos honestos, `Content-Type` correcto, confirmación en `listen`, puerto por variable de entorno, lógica en funciones, secretos en variables de entorno (nunca en el código), y logs de lo importante.

Lo más importante de todo: ya hiciste un servidor que responde "hola mundo", un mini sitio con rutas, una API que sirve JSON, un panel con datos reales de tu máquina, un formulario que recibe datos por POST y una mini API REST de tareas. El esqueleto fue siempre el mismo; lo que cambió fue QUE serviste. Ese esqueleto es el patrón universal del servidor Node.js, y lo hiciste a mano, sin frameworks. Cuando el curso avance a Express, a APIs REST completas y a bases de datos, no vas a estar aprendiendo magia nueva: vas a estar poniendo herramientas cómodas sobre un cimiento que ya entiendes. Esa es la diferencia entre usar un framework y dominarlo.

En la próxima unidad vas a subir de nivel: vas a tomar este servidor hecho a mano y a conocer Express, que hace todo esto con menos código y más poder. Vas a ver rutas elegantes, middlewares y APIs REST de verdad. Y cada vez que Express haga algo "mágico", vas a sonreír, porque tu ya sabes que por dentro hay un `createServer`, un `req.url` y un `res.end` trabajando. Nos vemos ahí.

UNIDAD 5: NPM y herramientas de desarrollo

Introducción

Bienvenido a la unidad que separa a los que "juegan con Node.js" de los que de verdad construyen con Node.js. Hasta aquí ya sabes qué es Node, cómo instalarlo, cómo correr archivos con el comando node, cómo funciona el sistema de módulos con require y module.exports, y cómo Node maneja las operaciones asíncronas sin bloquearse. Con eso ya puedes escribir programas que funcionan. Pero hay una verdad incómoda que todo programador descubre tarde o temprano: en el mundo real, casi nadie escribe todo desde cero. Los programadores profesionales no reinventan la rueda cada vez que necesitan algo; usan piezas ya hechas, probadas por miles de personas, y las ensamblan. Esta unidad es sobre eso: sobre el ecosistema de piezas de Node.js y las herramientas que te permiten usarlo como un profesional.

Piensa en lo siguiente. Imagina que quieres armar una computadora. Podrías, en teoría, fabricar tú mismo el procesador, soldar la memoria, diseñar la tarjeta madre. En la práctica, nadie hace eso: vas a una tienda, compras las piezas que ya existen y las ensamblas según tu necesidad. La programación moderna funciona igual. Cuando necesitas que tu servidor web maneje rutas, no escribes tu propio framework desde cero: instalas Express. Cuando necesitas trabajar con fechas complicadas, no escribes tu propia librería de fechas: instalas una que ya existe. Cuando necesitas colores en la terminal, validación de formularios, conexión a bases de datos, encriptación, lectura de archivos Excel, envío de correos... para todo eso ya existe una pieza hecha por alguien más, gratis, lista para instalarse con un solo comando. Ese universo de piezas se llama npm, y esta unidad te enseña a moverte en él como pez en el agua.

npm son las siglas de Node Package Manager, el administrador de paquetes de Node.js. Un "paquete" es simplemente código que alguien más escribió, empaquetó y publicó para que cualquiera lo use. Y cuando digo "cualquiera", hablo en serio: el registro de npm es el registro de software más grande del mundo, con millones de paquetes disponibles. Hay paquetes para prácticamente todo lo que se te ocurra, desde cosas serias como frameworks web completos hasta cosas curiosas que solo existen por diversión. Aprender npm no es un tema "extra" del curso: es el tema que multiplica por cien lo que puedes construir con lo que ya sabes. Un programador que domina npm construye en una tarde lo que a otro le tomaría un mes escribiendo todo a mano.

En esta unidad vas a aprender qué es npm y de dónde salió, cómo inicializar un proyecto con npm init, y vas a desmenuzar el archivo package.json campo por campo hasta que lo leas como si fuera tu acta de nacimiento: name, versión, description, main, scripts, dependencies y todos los demás. Vas a entender el versionado semántico, ese sistema de números como 4.18.2 que parece decoración pero que en realidad es un contrato entre el autor del paquete y tú. Vas a dominar npm install: qué pasa exactamente cuando lo ejecutas, qué es la carpeta node_modules y por qué jamás debes subirla a git, y qué es el archivo package-lock.json y por qué ese sí debes subirlo. Vas a distinguir entre dependencias normales y dependencias de desarrollo, entre instalación local e instalación global, y vas a aprender a crear tus propios scripts con npm run para automatizar tareas. Vas a conocer npx, el primo de npm que te permite ejecutar paquetes sin instalarlos. Vas a aprender a desinstalar, actualizar y auditar paquetes. Y vas a dominar nodemon, la herramienta que reinicia tu servidor automáticamente cada vez que guardas un cambio, para que desarrolles sin parar y arrancar a mano.

El panorama de la unidad es el siguiente. Primero, en la explicación detallada, vamos a construir tu entendimiento desde los cimientos: qué es npm, su historia breve, por qué el registro de npm es el más grande del planeta, y la analogía de la refaccionaria que te va a acompañar toda tu carrera. Luego pasamos a lo práctico: npm init paso a paso, cada campo del package.json explicado con ejemplos reales, el versionado semántico desmenuzado (major, minor, patch, y los símbolos ^ y ~ que tanta confusión causan), y el ciclo completo de instalación de paquetes. Veremos la diferencia entre dependencies y devDependencies, entre instalar local y global, y cómo los scripts de npm te convierten en un programador más rápido. Después conocerás npx, aprenderás a mantener tus paquetes al día y a quitar los que ya no uses, y le dedicaremos una sección completa a nodemon: instalación, uso diario y configuración avanzada con nodemon.json. Cerraremos la explicación con temas que separan al aficionado del profesional: el archivo .gitignore esencial para Node, los errores más comunes con npm y sus soluciones paso a paso, y las buenas prácticas de seguridad, incluyendo npm audit.

Después, en los ejemplos paso a paso, vamos a construir un proyecto real desde cero: lo inicializas con npm init, instalas lodash y colors, los usas en código de verdad, creas scripts personalizados de start y dev, y configuras nodemon para desarrollar a gusto. En la práctica tendrás ejercicios graduados, cada uno con su solución explicada paso a paso, para que no solo copies comandos sino que entiendas qué hace cada uno. Y cerraremos con un resumen de los puntos clave de la unidad.

Un consejo antes de empezar, y te lo digo directo: esta unidad tiene muchos comandos y muchos conceptos nuevos, y la tentación va a ser leerla como si fuera una lista de recetas. No lo hagas. Cada comando de npm existe porque resuelve un problema real que alguien sufrió

antes que tú. Cuando leas sobre `package-lock.json`, piensa en el programador que desplegó su app un viernes y el lunes ya no funcionaba porque una dependencia se actualizó sola. Cuando leas sobre `.gitignore`, piensa en el que subió `node_modules` a GitHub y su repositorio pesó dos gigabytes. Las historias detrás de estas herramientas son las que hacen que se te queden grabadas. Vamos a darle con todo.

Explicación detallada

Qué es npm: la refaccionaria más grande del mundo

Empecemos por lo básico pero bien entendido. npm es el administrador de paquetes que viene incluido cuando instalas Node.js. Es dos cosas al mismo tiempo, y conviene no confundirlas. Por un lado, npm es una herramienta de línea de comandos: el programa que ejecutas en tu terminal cuando escribes `npm install` o `npm init`. Por otro lado, npm es un registro en línea, una bodega gigante en internet donde viven millones de paquetes listos para descargar. La herramienta y el registro trabajan juntos: cuando escribes `npm install lodash`, la herramienta se conecta al registro, descarga el paquete `lodash` y lo deja en tu proyecto.

La analogía que mejor funciona, y la que usan muchos maestros, es la de la refaccionaria. Imagina que eres mecánico. Un cliente llega con su coche descompuesto. Tú podrías fabricar cada pieza desde cero: fundir el metal, moldear el pistón, calibrarlo. Pero eso sería una locura: vas a la refaccionaria, pides la pieza exacta para ese modelo, te la dan en una caja, la instalas y listo. npm es la refaccionaria del programador de Node.js. Necesitas manejar fechas: hay pieza para eso. Necesitas un servidor web: hay pieza para eso. Necesitas leer un archivo PDF, mandar un correo, conectar una base de datos, hacer gráficas: para todo hay pieza, y la "compras" (gratis) con un comando.

Un poco de historia para que entiendas por qué npm es como es. npm nació en 2010, creado por Isaac Schlueter, casi al mismo tiempo que Node.js despegaba. En ese entonces, compartir código de JavaScript era un dolor de cabeza: la gente copiaba y pegaba archivos, o los descargaba a mano de páginas web. npm resolvió eso con una idea simple pero poderosa: un lugar central donde publicar código y una herramienta que lo descarga y lo organiza por ti. La idea prendió como reguero de pólvora. Hoy el registro de npm contiene millones de paquetes y se descargan miles de millones de veces por semana. Es, por mucho, el registro de paquetes de software más grande que ha existido. Para que te des una idea de la escala: hay más paquetes en npm que libros en muchas bibliotecas universitarias. Y la inmensa mayoría son gratis y de código abierto.

Aquí va un dato que te va a volar la cabeza y que explica por qué npm es tan importante: cuando instalas un paquete popular como Express, no estás instalando un solo paquete. Express depende de otros paquetes, y esos dependen de otros más. Un proyecto típico de Node.js puede tener cientos o hasta miles de paquetes instalados aunque tú solo hayas pedido cinco o seis directamente. Esto se llama el árbol de dependencias, y es tanto la mayor fortaleza de npm (reutilizas el trabajo de miles de personas) como su mayor riesgo (si un paquete pequeño falla, puede afectar a miles de proyectos). Más adelante veremos cómo npm maneja esto y cómo protegerte.

Otra cosa que debes entender desde ya: npm no es la única opción que existe. Hay otros administradores como yarn y pnpm que hacen básicamente lo mismo con algunas diferencias de velocidad o de manejo del disco. En este curso usamos npm porque es el que viene con Node.js, el que más documentación tiene y el que te van a pedir en el noventa por ciento de los trabajos y tutoriales. Lo que aprendas de npm te sirve como base para entender los demás en una tarde.

npm init: el acta de nacimiento de tu proyecto

Todo proyecto serio de Node.js empieza con el mismo ritual: npm init. Este comando crea el archivo package.json, que es algo así como el acta de nacimiento y la receta de tu proyecto al mismo tiempo. Sin package.json, tu carpeta es solo una carpeta con archivos JavaScript sueltos. Con package.json, tu carpeta se convierte en un proyecto de Node.js de verdad: tiene nombre, versión, lista de piezas que necesita y comandos para trabajar con él.

Vamos a hacerlo paso a paso. Abre tu terminal, crea una carpeta para practicar y entra en ella:

```
mkdir mi-proyecto
cd mi-proyecto
npm init
```

Cuando ejecutas npm init sin más, npm te hace una serie de preguntas, una por una, como un formulario. Te pregunta el nombre del paquete (por defecto usa el nombre de la carpeta), la versión (por defecto 1.0.0), la descripción, el punto de entrada (por defecto index.js), el comando de prueba, el repositorio git, las palabras clave, el autor y la licencia. Puedes responder cada una o simplemente apretar Enter para aceptar el valor por defecto. Al final te muestra el JSON resultante y te pregunta si está bien; escribes "yes" y listo: nace tu package.json.

Si las preguntas te dan flojera o ya sabes que vas a aceptar todo por defecto, existe el atajo: npm init -y. La bandera -y significa "yes", sí a todo. En un segundo tienes tu package.json

creado con valores razonables por defecto, y luego lo editas a mano si necesitas cambiar algo. Los programadores experimentados usan `npm init -y` casi siempre, porque es más rápido ajustar el archivo después que responder ocho preguntas cada vez.

Un detalle importante: el nombre del proyecto tiene reglas. Debe estar en minúsculas, no puede llevar espacios (usa guiones si necesitas separar palabras, como `mi-proyecto`), y no puede ser igual al nombre de un paquete famoso del registro a menos que de verdad sea tuyo. Si algún día quieres publicar tu propio paquete en npm, el nombre debe ser único en todo el registro. Para proyectos personales que nunca vas a publicar, esto no importa mucho, pero es buena costumbre poner nombres limpios desde el principio.

Otro detalle: `npm init` crea el `package.json` en la carpeta donde estás parado. Asegúrate de estar en la raíz de tu proyecto antes de ejecutarlo. Si lo ejecutas en la carpeta equivocada, el archivo nace en el lugar equivocado, y luego andas batallando con rutas. Parece obvio, pero es un error más común de lo que crees, sobre todo cuando tienes varias terminales abiertas.

package.json campo por campo: la receta de tu proyecto

El `package.json` es el archivo más importante de tu proyecto después de tu propio código. Es un archivo JSON, es decir, un texto con estructura de llaves, comillas y comas que tanto JavaScript como npm saben leer. Piensa en él como la receta de tu proyecto: dice qué se llama, qué versión tiene, qué piezas necesita y qué comandos sabe ejecutar. La magia está en esto: si le das tu `package.json` a cualquier programador del mundo, esa persona puede reconstruir tu proyecto idéntico ejecutando un solo comando: `npm install`. Sin `package.json`, reconstruir tu proyecto sería adivinar qué piezas usaste y en qué versiones. Con `package.json`, es automático.

Vamos a desmenuzar cada campo importante con un ejemplo real. Supón que este es el `package.json` de un proyecto llamado "mi-tienda":

```
{
  "name": "mi-tienda",
  "version": "1.0.0",
  "description": "Tienda en línea de ejemplo para el curso",
  "main": "index.js",
  "scripts": {
    "start": "node index.js",
    "dev": "nodemon index.js",
    "test": "echo \"Error: no hay pruebas todavía\" && exit 1"
  },
  "keywords": ["tienda", "ejemplo", "curso"],
  "author": "Tu Nombre",
  "license": "MIT",
  "dependencies": {
    "express": "^4.18.2",
    "colors": "^1.4.0"
  },
  "devDependencies": {
    "nodemon": "^3.0.1"
  }
}
```

Campo "name": el nombre de tu proyecto, en minúsculas y sin espacios. Es obligatorio. Si algún día publicas el paquete, este será su nombre en el registro de npm. Para proyectos privados no se publica nada, pero el nombre sigue siendo obligatorio porque npm lo usa para identificar el proyecto internamente.

Campo "versión": la versión de tu proyecto, usando versionado semántico: tres números separados por puntos (major.minor.patch), empezando normalmente en 1.0.0. Cada vez que haces un cambio importante en tu proyecto, subes la versión. Más adelante dedicaremos una sección completa al versionado semántico porque es uno de esos temas que parecen triviales y luego te salvan de desastres.

Campo "description": una frase corta que describe qué hace tu proyecto. No es obligatorio, pero es buena práctica llenarlo. Cuando alguien abre tu proyecto por primera vez, la descripción es lo primero que lee. Además, si publicas el paquete, la descripción aparece en la página del registro de npm.

Campo "main": el archivo principal de tu proyecto, el punto de entrada. Por defecto es index.js. Esto le dice a Node.js (y a otros programadores) cuál archivo se ejecuta cuando alguien hace require de tu paquete. Si tu archivo principal se llama app.js o server.js, debes cambiar este campo para que apunte al archivo correcto. Un error clásico de principiante: renombrar index.js a app.js y olvidar actualizar el "main", y luego nada funciona y no saben por qué.

Campo "scripts": esta es una de las partes más poderosas del package.json y una de las menos aprovechadas por los principiantes. Los scripts son atajos de comandos que defines tú. En vez de escribir node index.js cada vez, defines "start": "node index.js" y luego ejecutas npm start. En vez de acordarte de una cadena larga de comandos para desarrollo, defines "dev": "nodemon index.js" y ejecutas npm run dev. Los scripts convierten comandos largos y propensos a errores en palabras cortas que todo tu equipo puede usar igual. Veremos esto a fondo en su propia sección.

Campo "keywords": una lista de palabras clave que describen tu proyecto. Sirven para que la gente encuentre tu paquete si lo publicas en el registro. No afectan el funcionamiento; son metadatos útiles.

Campo "author": tu nombre (y si quieres, tu correo entre paréntesis angulares, como "Tu Nombre <tu@correo.com>"). Es tu firma en el proyecto.

Campo "license": la licencia de tu código, por defecto "ISC". La más común en proyectos abiertos es "MIT", que básicamente dice "úsalo como quieras". Si tu proyecto es privado y nunca lo vas a publicar, puedes poner "UNLICENSED" o dejar el valor por defecto; no pasa nada. Este campo existe por razones legales, no técnicas.

Campo "dependencies": el corazón operativo del package.json. Aquí vive la lista de paquetes que tu proyecto necesita para funcionar, cada uno con la versión que debe instalarse. Cuando ejecutas npm install lodash, npm descarga lodash y automáticamente agrega una línea en "dependencies". Cuando alguien más clona tu proyecto y ejecuta npm install sin argumentos, npm lee esta sección y descarga todo lo necesario. Es la lista de compras de tu refaccionaria.

Campo "devDependencies": igual que "dependencies", pero para paquetes que solo necesitas mientras desarrollas, no cuando el proyecto ya está en producción. El ejemplo clásico es nodemon: lo usas para desarrollar cómodo, pero el servidor en producción no lo necesita. Separar dependencias de desarrollo de dependencias de producción hace tus despliegues más ligeros y más rápidos. Se instalan con npm install --save-dev.

Fíjate en algo importante: tú casi nunca escribes a mano las secciones "dependencies" y "devDependencies". npm las escribe por ti cada vez que instalas o desinstalas un paquete. Tu trabajo es entender qué significan y saber leerlas, no memorizar sintaxis. El package.json es

un archivo que se edita solo en gran medida; tú lo configuras al inicio y npm lo mantiene actualizado.

Versionado semántico: los tres números que son un contrato

Esos números como 4.18.2 que ves en todas las dependencias no son decoración: son un contrato. Se llaman versionado semántico, o "semver" en inglés, y son un acuerdo entre el autor del paquete y tú sobre qué tipo de cambios trae cada actualización. La versión se compone de tres números separados por puntos: MAJOR.MINOR.PATCH. Cada número tiene un significado preciso, y entenderlo te evita una cantidad industrial de dolores de cabeza.

El primer número es el MAJOR (mayor). Cambia cuando el autor hace cambios que ROMPEN la compatibilidad con versiones anteriores. Ejemplo: pasas de la versión 3.5.0 a la 4.0.0. Ese salto de 3 a 4 te está diciendo: "cuidado, algo cambió en la forma de usar este paquete y tu código viejo podría dejar de funcionar". Cuando veas que un paquete sube su número mayor, actualiza con precaución: lee qué cambió antes de hacerlo. Un ejemplo famoso: cuando Express pasó de la versión 3 a la 4, varias formas de configurar el servidor cambiaron y muchos tutoriales viejos dejaron de funcionar.

El segundo número es el MINOR (menor). Cambia cuando se agregan funciones nuevas pero sin romper nada de lo que ya existía. Ejemplo: de 4.17.0 a 4.18.0. El mensaje es: "hay cosas nuevas que puedes usar si quieres, pero tu código actual sigue funcionando igual". Estas actualizaciones son seguras en general: puedes tomarlas sin miedo.

El tercer número es el PATCH (parche). Cambia cuando se corrigen errores sin agregar nada nuevo. Ejemplo: de 4.18.1 a 4.18.2. El mensaje es: "arreglamos un bug, nada más". Estas actualizaciones son las más seguras de todas; de hecho, no actualizarlas es más riesgoso, porque te quedas con el bug.

Ahora, lo interesante: cuando npm instala un paquete, no escribe la versión exacta en tu package.json, sino un rango. Y ahí entran los símbolos ^ y ~, que confunden a todo principiante. Vamos a desarmarlos.

El símbolo ^ (caret, "sombbrero") es el que npm usa por defecto. Cuando ves "express": "^4.18.2", eso significa: "instala la versión 4.18.2 o cualquier versión posterior COMPATIBLE, es decir, cualquiera que no cambie el número mayor". En la práctica: puede instalar 4.18.3, 4.19.0, 4.99.99, pero JAMÁS la 5.0.0. La lógica es: mientras el mayor sea el mismo, el autor promete no romper nada, así que es seguro tomar parches y minors nuevos automáticamente. Este es el comportamiento sensato para la mayoría de los casos.

El símbolo ~ (tilde) es más conservador. Cuando ves "express": "~4.18.2", significa: "instala la 4.18.2 o cualquier PATCH posterior, pero no subas el minor". Es decir: puede instalar 4.18.3, 4.18.9, pero no 4.19.0. Se usa cuando quieres ser extra cuidadoso y solo aceptar correcciones de errores, no funciones nuevas.

También puedes escribir la versión exacta sin ningún símbolo: "express": "4.18.2". Eso significa: "quiero exactamente esta versión, ni una más ni una menos". Se usa cuando necesitas reproducibilidad total, por ejemplo en sistemas críticos donde ni un minor nuevo es aceptable sin pruebas.

Y hay más símbolos para casos especiales: ">=4.18.2" (esta versión o mayor), "4.18.x" (cualquier patch del minor 18), "*" (cualquiera, no lo uses jamás en serio). Pero con ^, ~ y la versión exacta tienes cubierto el 99 por ciento de la vida real.

Un consejo de veterano: no le tengas miedo a actualizar, pero tampoco actualices a lo loco. La combinación ganadora es: usa ^ en tu package.json (el comportamiento por defecto), deja que npm tome parches y minors automáticamente, y revisa con calma cuando un paquete anuncie un cambió de mayor. Y para blindarte del todo, existe el package-lock.json, que veremos en un momento: ese archivo congela las versiones exactas que se instalaron, para que tu proyecto sea reproducible al cien por ciento.

npm install: qué pasa realmente cuando lo ejecutas

npm install es el comando que más vas a usar en tu vida con Node.js, así que vale la pena entender qué hace por dentro y no tratarlo como una caja negra. Tiene dos modos, y es crucial distinguirlos.

Modo uno: npm install lodash (con nombre de paquete). Esto hace cuatro cosas en orden. Primero, npm busca el paquete "lodash" en el registro y averigua cuál es la última versión compatible según las reglas que ya vimos. Segundo, descarga el paquete y todas sus dependencias (los paquetes de los que lodash depende, y los paquetes de los que esos dependen, y así). Tercero, crea la carpeta node_modules en tu proyecto (si no existe) y descomprime ahí todo lo descargado, organizando el árbol de dependencias. Cuarto, actualiza tu package.json agregando la línea "lodash": "^4.17.21" en la sección "dependencies", y actualiza el package-lock.json con las versiones exactas instaladas.

Modo dos: `npm install` (sin ningún nombre, a secas). Este es el modo "reconstrucción". `npm` lee tu `package.json`, mira la lista de "dependencias" y "devDependencies", y descarga e instala todo lo necesario para que el proyecto funcione. Es el comando que ejecutas cuando clonas un proyecto de GitHub, cuando cambias de computadora, o cuando un compañero te pasa su código. Un solo comando y el proyecto queda idéntico al original. Esta es la magia de la que hablábamos: la receta (`package.json`) más el comando reconstruyen todo.

Hay banderas útiles que debes conocer. `npm install --save-dev nodemon` instala el paquete como dependencia de desarrollo (lo anota en "devDependencies" en vez de "dependencies"). En versiones modernas de `npm`, el `--save` ya no es necesario para dependencias normales: `npm install lodash` solito ya lo guarda en "dependencies". Pero el `--save-dev` sí sigue siendo necesario para marcar algo como dependencia de desarrollo. Otra bandera útil: `npm install --no-save` instala el paquete sin anotarlo en el `package.json`; sirve para pruebas rápidas que no quieres que queden registradas.

Un detalle que confunde: a veces verás tutoriales viejos que escriben `npm i` en vez de `npm install`. Son lo mismo: "i" es el alias corto de "install". Igual que "`npm t`" es alias de "`npm test`" y "`npm run`" tiene sus atajos. Úsalos cuando te sientas cómodo; al principio mejor escribe los comandos completos para que se te graben.

node_modules: la bodega que no se sube a git

La carpeta `node_modules` es donde `npm` guarda todos los paquetes descargados. Es la bodega de tu refaccionaria personal. Cada paquete vive en su propia subcarpeta, con su código, su propio `package.json` y sus propias dependencias. Cuando en tu código escribes `const _ = require("lodash")`, Node.js busca la carpeta `lodash` dentro de `node_modules` y carga su archivo principal. Así de directo es el mecanismo.

Ahora, la regla de oro que todo programador de Node.js aprende (a veces a la mala): `node_modules` NO se sube a git, NO se manda por correo, NO se respalda. ¿Por qué? Tres razones. Primera: pesa muchísimo. Un proyecto pequeño puede tener cientos de megabytes en `node_modules`; uno grande, gigabytes. Subir eso a un repositorio es un desperdicio obsceno de espacio y tiempo. Segunda: es reconstruible. Con tu `package.json` y un `npm install`, cualquier persona regenera la carpeta idéntica en segundos o minutos. Guardar algo reconstruible es como guardar fotocopias de un libro que ya tienes: ocupa espacio sin aportar nada. Tercera: contiene código compilado específico de tu sistema operativo en algunos casos; lo que funciona en tu Windows puede no servir en el Linux del servidor. Cada máquina debe generar su propio `node_modules` con `npm install`.

¿Cómo evitas subirla a git por accidente? Con un archivo `.gitignore` que le diga a git "ignora esta carpeta". Le dedicaremos una sección completa más adelante, pero adelanto la idea: un archivo de texto llamado `.gitignore` en la raíz de tu proyecto, con la línea `node_modules` adentro, y git jamás volverá a molestarte con esa carpeta.

¿Qué haces si `node_modules` se corrompe o algo deja de funcionar misteriosamente? El remedio clásico, el "apágalo y préndelo" del mundo Node, es: borra la carpeta `node_modules` por completo, borra también el `package-lock.json` si sospechas que el problema está en las versiones, y ejecuta `npm install` de nuevo para reconstruir todo desde cero. Suena drástico, pero funciona el noventa por ciento de las veces y es totalmente seguro, porque todo es reconstruible desde tu `package.json`. Grábate este remedio: te va a salvar más de una vez.

package-lock.json: el candado de las versiones exactas

Si el `package.json` es la receta, el `package-lock.json` es la fotografía exacta de la cocina después de preparar el platillo. Mientras el `package.json` dice "express": "`^4.18.2`" (un rango: cualquier 4.x compatible), el `package-lock.json` dice "se instaló exactamente la versión 4.18.2, descargada de tal URL, con tal huella digital, y estas fueron las versiones exactas de sus 50 dependencias". Es un archivo largo, generado automáticamente, que congela el árbol completo de dependencias con precisión quirúrgica.

¿Para qué sirve? Para la reproducibilidad. Imagina este escenario de terror: tú desarrollas tu app un lunes con Express 4.18.2 y todo funciona. El viernes, tu compañero clona el proyecto y ejecuta `npm install`. Pero entre el lunes y el viernes salió Express 4.19.0, y como tu `package.json` permite `^4.x`, a tu compañero se le instala la 4.19.0. Y resulta que la 4.19.0 trae un cambio sutil que rompe tu código. Tú dices "a mí me funciona", él dice "a mí no", y pierden una tarde entera hasta descubrir que tienen versiones distintas. El `package-lock.json` evita exactamente eso: cuando tu compañero ejecuta `npm install` y existe un `package-lock.json`, `npm` instala LAS VERSIONES EXACTAS que dice el candado, no "la última compatible". Los dos quedan con el mismo árbol, byte por byte.

Por eso la regla es clara y no se discute: el `package-lock.json` SÍ se sube a git, SIEMPRE. Es parte del proyecto, como el `package.json`. Cuando haces cambios en dependencias (instalas, actualizas, desinstalas), el `package-lock.json` se actualiza solo; tú solo te aseguras de subirlo al repositorio junto con tus demás cambios.

Hay un comando primo que debes conocer: `npm ci`. Mientras `npm install` puede actualizar el `package-lock.json` si encuentra versiones nuevas compatibles, `npm ci` ("clean install") es estricto: borra `node_modules` e instala EXACTAMENTE lo que dice el `package-lock.json`, y si el candado no coincide con el `package.json`, falla en vez de adivinar. `npm ci` es lo que se usa

en servidores de producción y en sistemas de integración continua, donde quieres certeza total y cero sorpresas. En tu máquina de desarrollo usa `npm install`; en despliegues automáticos, `npm ci`.

Dependencias vs devDependencies: lo que viaja a producción y lo que se queda en casa

Ya vimos los campos en el `package.json`; ahora entendamos la filosofía detrás de la separación. Las "dependencies" son los paquetes que tu aplicación necesita para FUNCIONAR: Express para el servidor, lodash para utilidades, colors para la terminal si tu app los usa en producción. Sin estos paquetes, tu programa no corre. Las "devDependencies" son los paquetes que necesitas para DESARROLLAR: nodemon para reiniciar el servidor automáticamente, herramientas de pruebas, formateadores de código. Sin estos paquetes tu programa sí corre; solo que desarrollar es menos cómodo.

¿Por qué importa la distinción? Por el despliegue. Cuando subes tu app a un servidor de producción, quieres que la instalación sea rápida y ligera. Con `npm install --production` (o configurando la variable de entorno `NODE_ENV=production`), npm instala solo las "dependencies" y se salta las "devDependencies". Si tienes 200 megabytes de herramientas de desarrollo, el servidor no tiene por qué descargarlas. Además, menos paquetes en producción significa menos superficie de ataque para problemas de seguridad.

¿Cómo decides dónde va cada paquete? La pregunta mágica es: "si borró este paquete, ¿mi aplicación sigue funcionando para el usuario final?" Si la respuesta es no, va en "dependencies". Si la respuesta es sí (solo afecta tu comodidad al programar), va en "devDependencies". Ejemplos claros: express, lodash, colors van en dependencies. nodemon, las librerías de testing, los linters van en devDependencies.

Para instalar como dependencia de desarrollo usas `npm install --save-dev nombre-paquete`. Para mover un paquete de un lado a otro después, lo más limpio es desinstalarlo y reinstalarlo con la bandera correcta: `npm uninstall nodemon` y luego `npm install --save-dev nodemon`. También puedes editar el `package.json` a mano moviendo la línea de una sección a otra y ejecutando `npm install`, pero el camino de desinstalar y reinstalar es menos propenso a errores.

Instalación local vs global: la caja de herramientas personal y la del taller

Cuando ejecutas `npm install lodash`, el paquete se instala LOCAL: dentro de la carpeta `node_modules` de tu proyecto actual. Solo ese proyecto puede usarlo. Esto es lo correcto el 95 por ciento de las veces: cada proyecto tiene sus propias piezas, en sus propias versiones, sin

interferir con otros proyectos. Es como tener tu caja de herramientas en tu casa: tus desarmadores no se mezclan con los del vecino.

Pero hay paquetes que no pertenecen a ningún proyecto en particular, sino que son herramientas que quieres usar desde cualquier carpeta de tu computadora. Para esos existe la instalación GLOBAL con la bandera -g: `npm install -g nodemon`. Un paquete global se instala en una carpeta especial del sistema y su comando queda disponible en toda tu terminal, sin importar en qué carpeta estés parado. Es como las herramientas del taller comunal: están ahí para quien las necesite.

¿Qué se instala global y qué local? La regla práctica: global van las herramientas de línea de comandos que usas en muchos proyectos (nodemon es el ejemplo clásico de este curso; otros comunes son herramientas de construcción o de despliegue). Local va TODO lo que tu código importa con `require`. Esta distinción es importante porque es un error típico de principiante instalar global algo que el código necesita con `require`: el `require` busca en el `node_modules` local, no encuentra el paquete global, y el programa truena con "Cannot find module". Si tu código hace `require("lodash")`, `lodash` debe estar instalado local en ese proyecto, punto.

Otro detalle: los paquetes globales a veces dan problemas de permisos en Linux y Mac, porque la carpeta global del sistema requiere privilegios de administrador. Si `npm install -g` te responde con errores de "EACCES" o "permission denied", NO lo resuelvas usando `sudo` a lo loco (eso crea más problemas de permisos después). La solución limpia es configurar `npm` para que use una carpeta global dentro de tu directorio personal, sin privilegios especiales. En la sección de errores comunes veremos el procedimiento exacto paso a paso.

Para ver qué tienes instalado globalmente, usa `npm list -g --depth=0`: te muestra la lista de paquetes globales sin el árbol completo de sus dependencias. Y para saber dónde viven los paquetes globales en tu sistema, `npm root -g` te muestra la ruta. Son comandos de diagnóstico que te sacan de apuros cuando algo "debería funcionar y no funciona".

npm run y scripts personalizados: tus atajos de poder

La sección "scripts" del `package.json` es una de las herramientas más subestimadas por los principiantes y una de las más amadas por los profesionales. Un script es simplemente un atajo: le pones un nombre corto a un comando largo (o a una secuencia de comandos) y luego lo ejecutas con `npm run nombre`. Parece poca cosa hasta que trabajas en un proyecto real donde los comandos son largos, tienen banderas específicas y todo el equipo debe ejecutarlos exactamente igual.

Empecemos con lo básico. Cuando npm init te crea el package.json, ya trae un script de ejemplo llamado "test". Tú puedes agregar los tuyos editando el archivo. Un conjunto típico para un proyecto Node.js se ve así:

```
"scripts": {  
  "start": "node index.js",  
  "dev": "nodemon index.js",  
  "test": "echo \"Error: no hay pruebas todavía\" && exit 1"  
}
```

Para ejecutar un script escribes npm run seguido del nombre: npm run dev ejecuta nodemon index.js. Hay dos excepciones famosas con atajo directo: npm start es lo mismo que npm run start, y npm test es lo mismo que npm run test. Son los únicos dos scripts que puedes correr sin escribir "run". Por convención de toda la industria, "start" significa "corre la aplicación en modo producción" y "dev" significa "corre la aplicación en modo desarrollo" (típicamente con nodemon). Si respetas esa convención, cualquier programador que abra tu proyecto sabrá cómo arrancarlo sin preguntar.

¿Por qué usar scripts en vez de escribir los comandos directamente? Tres razones de peso. Primera: los comandos reales suelen ser largos y fáciles de escribir mal. Compara npm run dev contra ./node_modules/.bin/nodemon --watch src --ext js,json --ignore src/datos.json src/index.js. El script esconde la complejidad. Segunda: consistencia del equipo. Si el comando vive en el package.json, todos ejecutan exactamente lo mismo; nadie tiene que adivinar banderas. Tercera: los scripts pueden encadenarse. Puedes definir "deploy": "npm run build && npm start", y con un solo comando haces dos pasos en orden. El operador && significa "ejecuta el segundo comando solo si el primero tuvo éxito".

Un truco poderoso: dentro de los scripts puedes usar variables de entorno. Por ejemplo, "dev": "NODE_ENV=development nodemon index.js" (en Windows la sintaxis varía un poco, pero la idea es la misma). Y puedes definir scripts que se ejecutan automáticamente antes o después de otros usando los prefijos "pre" y "post": un script llamado "prestart" se ejecuta automáticamente antes de "start", y "poststart" después. Esto sirve para tareas como "antes de arrancar, verifica que la base de datos esté disponible".

Otro uso genial de los scripts: documentar tu proyecto. Un package.json bien escrito con scripts claros ("start", "dev", "test", "build", "lint") es como un manual de instrucciones integrado. Un programador nuevo clona el repo, lee los scripts y ya sabe qué comandos existen. No necesita un documento aparte ni preguntarle a nadie. Esa es la marca de un proyecto profesional.

npx: el primo que ejecuta sin instalar

npx es una herramienta que viene incluida con npm (desde la versión 5.2) y que resuelve un problema muy concreto: ¿qué haces cuando necesitas ejecutar un paquete UNA sola vez, sin querer instalarlo en tu proyecto ni globalmente? La respuesta es npx.

El caso clásico: quieres probar una herramienta o ejecutar un generador de proyectos. Sin npx, tendrías que instalarlo global (ensuciando tu sistema con algo que usarás una vez) o instalarlo local (ensuciando tu proyecto). Con npx, simplemente escribes `npx nombre-paquete` y listo: npx descarga el paquete a una caché temporal, lo ejecuta y luego lo olvida. Ejemplo real: `npx create-react-app mi-app` descarga el generador de proyectos de React, lo ejecuta para crear tu proyecto y no deja rastro en tu sistema.

Otro uso cotidiano: ejecutar los programas que vienen dentro de tus paquetes locales. Cuando instalas un paquete local que trae un ejecutable (como nodemon instalado localmente), ese ejecutable queda en `node_modules/.bin/`. Para correrlo tendrías que escribir la ruta completa `./node_modules/.bin/nodemon`, que es un horror. Con `npx nodemon index.js`, npx busca primero en tu `node_modules/.bin` local y lo ejecuta desde ahí. Es el atajo limpio para usar herramientas locales sin instalarlas globalmente.

La diferencia entre npm y npx, en una frase: npm INSTALA paquetes, npx EJECUTA paquetes. Se complementan. Y un consejo práctico: si vas a usar una herramienta todos los días en muchos proyectos (como nodemon), instálala global con `npm install -g`. Si la vas a usar una vez o solo dentro de un proyecto, usa npx o instálala local. Así mantienes tu sistema limpio y tus proyectos autocontenidos.

Desinstalar y actualizar: mantenimiento de tu refaccionaria

Instalar paquetes es solo la mitad del trabajo; la otra mitad es mantenerlos. Los proyectos viven meses o años, los paquetes sacan nuevas versiones, y de vez en cuando un paquete deja de servirte. Estos son los comandos de mantenimiento que debes dominar.

Para desinstalar un paquete: `npm uninstall lodash`. Esto hace tres cosas: borra la carpeta del paquete dentro de `node_modules`, quita su línea de la sección "dependencies" (o "devDependencies") de tu `package.json`, y actualiza el `package-lock.json`. Limpio y completo. Si solo borras la carpeta de `node_modules` a mano pero no actualizas el `package.json`, la próxima vez que alguien ejecute `npm install` el paquete regresará como si nada, porque la receta sigue pidiéndolo. Siempre desinstala con el comando, no a mano.

Para ver qué paquetes tienen actualizaciones disponibles: `npm outdated`. Este comando compara las versiones que tienes instaladas contra las últimas versiones publicadas en el

registro y te muestra una tabla con tres columnas clave: la versión actual instalada, la versión "wanted" (la última compatible con tu rango ^ o ~) y la versión "latest" (la última absoluta, aunque rompa compatibilidad). Es tu tablero de control: de un vistazo sabes qué se puede actualizar sin riesgo y qué requiere cuidado.

Para actualizar: `npm update`. Este comando sube cada paquete a la última versión permitida por tu rango (la columna "wanted" de `npm outdated`). Es decir: respeta tu ^ y tu ~, no salta a un mayor nuevo sin tu permiso. Después de actualizar, revisa que tu proyecto siga funcionando y sube el `package-lock.json` actualizado a git. Si quieres actualizar UN paquete específico a su última versión aunque eso signifique saltar de mayor, lo instalas de nuevo pidiendo la versión: `npm install express@latest`. El `@latest` le dice a npm "quiero la última, sin importar el rango actual", y npm actualiza el `package.json` con el nuevo rango.

La estrategia sana de actualización, la que usan los equipos profesionales, es esta: ejecuta `npm outdated` una vez al mes (o cuando el proyecto lo pida), actualiza primero los parches y minors con `npm update`, prueba que todo siga funcionando, y planifica aparte las actualizaciones de mayor, leyéndote los cambios importantes antes de aplicarlas. Actualizar todo a lo loco un viernes a las seis de la tarde es la receta perfecta para un fin de semana arruinado. Y siempre, siempre, después de actualizar, ejecuta tu proyecto y verifica que arranque y responda bien antes de dar el tema por cerrado.

Nodemon a fondo: desarrolla sin parar y arrancar a mano

Llegamos a la estrella de esta unidad según el temario del curso: nodemon. Si has estado desarrollando con Node.js hasta aquí, conoces este ritual tedioso: editas tu código, guardas, vas a la terminal, presionas `Ctrl+C` para matar el servidor, presionas la flecha arriba para recuperar el comando `node index.js`, presionas `Enter`, vuelves al editor... y repites eso CIENTOS de veces al día. Es un impuesto al desarrollo que pagas con tu tiempo y tu concentración. Nodemon elimina ese impuesto por completo.

Nodemon (de "node monitor") es una herramienta que envuelve tu aplicación Node.js y la vigila: cada vez que detecta que guardaste un cambio en tus archivos, reinicia tu servidor automáticamente. Tú solo editas, guardas y ves el resultado. Sin `Ctrl+C`, sin flecha arriba, sin `Enter`. El flujo se vuelve: escribes código, guardas, pruebas en el navegador, repites. Es una de esas herramientas que, una vez que la pruebas, te preguntas cómo vivías sin ella.

Instalación: como nodemon es una herramienta de desarrollo que quieres disponible en todos tus proyectos, la instalación recomendada es global:

```
npm install -g nodemon
```

Una sola vez en tu computadora y queda disponible en cualquier carpeta. Después de instalarlo, verifica que quedó bien con `nodemon --versión`: debe mostrarte el número de versión sin errores.

Uso básico: en vez de arrancar tu app con `node index.js`, la arrancas con `nodemon index.js`. Eso es todo. Nodemon ejecuta tu archivo igual que lo haría node, pero se queda vigilando. Verás en la terminal un mensaje como "[nodemon] starting node index.js" y luego la salida normal de tu programa. Edita tu archivo, guárdalo, y verás aparecer "[nodemon] restarting due to changes..." seguido del reinicio. Magia pura.

Nodemon por defecto vigila todos los archivos JavaScript de tu carpeta actual y sus subcarpetas, pero ignora `node_modules` (obviamente: si vigilara los miles de archivos de tus dependencias, tu computadora se derretiría). También puedes decirle qué extensiones vigilar: por defecto son `js`, `mjs`, `json` y algunas más. Si tu proyecto usa otras extensiones (por ejemplo, archivos de plantillas), se las indicas con la bandera `--ext`: `nodemon --ext js,json,html index.js`.

Hay banderas útiles para el día a día. `nodemon --ignore "datos/*.json"` le dice que ignore cambios en cierta carpeta o archivos (útil si tu app escribe archivos de log o de datos que no deberían provocar reinicios). `nodemon --delay 2` le dice que espere 2 segundos después de detectar cambios antes de reiniciar (útil si tu editor guarda varias veces seguidas o si usas herramientas que escriben archivos en ráfaga). `nodemon --exec "node --inspect"` te permite pasarle banderas al propio node, por ejemplo para depurar.

Integración con tus scripts: la forma elegante de usar nodemon en un proyecto es a través del `package.json`, no escribiendo el comando a mano cada vez. Agregas el script "dev": `"nodemon index.js"` y a partir de entonces arrancas tu entorno de desarrollo con `npm run dev`. Todo tu equipo usa el mismo comando, y si algún día necesitas agregar banderas (como `--ignore`), las agregas una vez en el script y todos las heredan. Recuerda la convención: "start" para producción con node, "dev" para desarrollo con nodemon.

Configuración avanzada con nodemon.json

Cuando tu proyecto crece, las banderas en la línea de comandos se vuelven incómodas: el script "dev" se alarga, nadie se acuerda de qué banderas lleva y cada quien inventa las suyas. Para eso existe `nodemon.json`: un archivo de configuración en la raíz de tu proyecto donde defines el comportamiento de nodemon una vez y para siempre.

Creas un archivo llamado `nodemon.json` en la raíz de tu proyecto con un contenido como este:

```
{
  "watch": ["src", "config"],
  "ext": "js,json",
  "ignore": ["src/datos/*.json", "src/logs/*"],
  "delay": 1500,
  "verbose": true
}
```

Vamos campo por campo. "watch" es la lista de carpetas o archivos que nodemon debe vigilar; si no lo defines, vigila todo el proyecto. Limitarlo a tus carpetas de código hace que nodemon sea más rápido y evita reinicios por archivos que no importan. "ext" define las extensiones que disparan reinicios. "ignore" es la lista de patrones a ignorar: archivos de datos que tu app escribe, logs, archivos temporales. "delay" es la espera en milisegundos antes de reiniciar (1500 ms = 1.5 segundos), útil contra guardados en ráfaga. "verbose": true hace que nodemon te cuente con más detalle qué está haciendo, útil mientras configuras y depuras.

Con nodemon.json en su lugar, tu script "dev" vuelve a ser simple: "dev": "nodemon index.js". Nodemon lee automáticamente el nodemon.json de la carpeta actual y aplica la configuración. Y como el archivo vive en tu proyecto (y se sube a git), todo el equipo comparte la misma configuración sin tener que memorizar nada.

Un par de trucos finales de nodemon que te van a servir. Si en algún momento necesitas reiniciar manualmente sin haber cambiado archivos, escribe "rs" y presiona Enter en la terminal donde corre nodemon: son las siglas de "restart" y fuerza un reinicio inmediato. Es útil cuando tu app depende de algo externo (una base de datos, una API) que cambió sin que nodemon lo notara. Y si nodemon se vuelve loco reiniciando sin parar (pasa cuando tu app escribe archivos dentro de una carpeta vigilada, creando un ciclo infinito de cambios y reinicios), la solución es agregar esos archivos al "ignore" del nodemon.json. Ese ciclo infinito es el problema más común con nodemon y siempre se arregla igual: identifica qué archivo está cambiando solo y exclúyelo.

.gitignore esencial para Node.js: lo que nunca se sube

Si usas git (y deberías: es el estándar para guardar y compartir código), hay archivos de un proyecto Node.js que jamás deben llegar al repositorio. El mecanismo para lograrlo es el archivo .gitignore: un archivo de texto en la raíz de tu proyecto donde escribes, línea por línea, los patrones de archivos y carpetas que git debe ignorar. Es la lista negra de tu repositorio.

El .gitignore mínimo y esencial para cualquier proyecto Node.js se ve así:

```
node_modules/  
.env  
.DS_Store  
*.log  
dist/  
coverage/
```

Vamos línea por línea. `node_modules/` es la obvia: la bodega reconstruible que pesa cientos de megabytes y que cada quien regenera con `npm install`. La diagonal al final indica que es una carpeta. Si ya la subiste por accidente alguna vez, agregarla al `.gitignore` no la borra del historial; en ese caso hay que sacarla con `git rm -r --cached node_modules`, pero eso es tema de un curso de git.

`.env` es el archivo donde viven tus secretos: contraseñas de bases de datos, claves de APIs, tokens. Por convención se llama `.env` y NUNCA se sube a git, porque un repositorio (sobre todo si es público en GitHub) con tus contraseñas es una invitación abierta al desastre. En su lugar se sube un archivo `.env.example` con los nombres de las variables pero sin los valores reales, para que cada programador sepa qué configurar. Este punto no es opcional: es seguridad básica.

`.DS_Store` es un archivo basura que macOS crea en cada carpeta para guardar preferencias de visualización. No tiene nada que ver con tu proyecto y ensucia el repositorio; se ignora siempre. `*.log` ignora todos los archivos de registro que tu app o tus herramientas generen. `dist/` es la carpeta donde muchos proyectos ponen el código ya construido para producción (si usas herramientas de compilación); se genera sola, no se sube. `coverage/` es donde las herramientas de pruebas guardan sus reportes; también se genera sola.

La regla general para decidir qué va en el `.gitignore`: si el archivo se puede REGENERAR con un comando (`node_modules` con `npm install`, `dist` con la compilación), se ignora. Si el archivo contiene SECRETOS (`.env`), se ignora. Si el archivo lo genera tu SISTEMA OPERATIVO o tu EDITOR (`.DS_Store`, carpetas como `.vscode` en algunos equipos), se ignora. Todo lo demás (tu código, tu `package.json`, tu `package-lock.json`, tu `nodemon.json`, tu `README`) se sube.

Un consejo de oro: crea el `.gitignore` AL INICIO del proyecto, antes del primer commit. Es mucho más fácil ignorar desde el principio que limpiar un repositorio ya contaminado. Muchos generadores y plantillas ya lo incluyen; si el tuyo no, estos seis renglones te cubren el 95 por ciento de los casos.

Errores comunes con npm y sus soluciones paso a paso

Esta sección te va a ahorrar horas de frustración. Son los errores que TODO programador de Node.js comete al menos una vez, con su causa y su solución explicadas sin tecnicismos innecesarios.

Error 1: "Cannot find module 'lodash'" (o el nombre de cualquier paquete). Causa: tu código hace require de un paquete que no está instalado en el `node_modules` local del proyecto. Pasa cuando clonas un proyecto y olvidas ejecutar `npm install`, o cuando instalaste el paquete global en vez de local. Solución: primero verifica que estás en la carpeta del proyecto y ejecuta `npm install`. Si el paquete no está en tu `package.json`, instálalo local: `npm install lodash`. Recuerda: si tu código lo requiere, debe estar instalado local, no global.

Error 2: versiones incompatibles, el famoso "infierno de dependencias". Síntoma: instalaste algo y de pronto otro paquete dejó de funcionar, o npm te muestra advertencias de "peer dependencies" no satisfechas. Causa: dos paquetes necesitan versiones distintas de un tercer paquete y chocan. Solución paso a paso: primero, lee el mensaje de error completo, que suele decir qué paquete pide qué versión. Segundo, prueba `npm install nombre-paquete@versión-especifica` para alinear las versiones. Tercero, si nada cuadra, el remedio nuclear: borra `node_modules` y `package-lock.json`, y reinstala todo con `npm install` para partir de un árbol limpio. En casos extremos, busca si alguno de los paquetes tiene una versión más nueva que ya resolvió el conflicto.

Error 3: "EACCES: permission denied" al instalar paquetes globales en Linux o Mac. Causa: npm intenta escribir en la carpeta global del sistema, que pertenece al administrador. Solución (la correcta, no el sudo a lo loco): configura npm para usar una carpeta global dentro de tu directorio personal. Los pasos son: crea la carpeta con `mkdir ~/.npm-global`, dile a npm que la use con `npm config set prefix '~/.npm-global'`, y agrega esa carpeta a tu PATH agregando la línea `export PATH=~/.npm-global/bin:$PATH` a tu archivo `~/.bashrc` o `~/.zshrc` (según tu terminal), y recarga con `source ~/.bashrc`. A partir de ahí, `npm install -g` funciona sin permisos especiales y sin ensuciar tu sistema. Hazlo una sola vez y olvídate del problema para siempre.

Error 4: el caché corrupto. Síntoma: errores extraños y aleatorios al instalar, descargas que fallan a la mitad, mensajes sobre archivos corruptos. Solución: `npm cache clean --force` limpia la caché de descargas de npm y la reconstruye desde cero. Es seguro y no toca tus proyectos; solo borra archivos temporales descargados. Después de limpiarla, vuelve a intentar la instalación.

Error 5: "npm ERR! code ERESOLVE" en versiones modernas de npm. Causa: npm detectó un conflicto de dependencias entre lo que pide tu proyecto y lo que piden los paquetes, y en vez de instalar algo roto, se niega. Solución: no lo fuerces con `--force` a menos que sepas

exactamente qué haces; mejor investiga el conflicto con `npm ls nombre-paquete` para ver quién pide qué, y resuelve alineando versiones. Si es un proyecto heredado con dependencias viejas y necesitas salir del paso, existe `npm install --legacy-peer-deps`, que le dice a npm "ignora los conflictos de peer dependencies e instala como antes". Úsalo como último recurso y documenta por qué lo usaste.

Error 6: nodemon no reinicia o reinicia en ciclo infinito. Si no reinicia: verifica que lo arrancaste con `nodemon index.js` (no con `node`) y que estás editando archivos dentro de las carpetas vigiladas. Si reinicia sin parar: tu app está escribiendo archivos dentro de una carpeta vigilada (logs, datos, subidas), y cada escritura dispara otro reinicio, que escribe de nuevo... ciclo infinito. Solución: agrega esos archivos o carpetas al "ignore" de tu `nodemon.json`, como vimos en la sección de configuración.

Error 7: "command not found: nodemon" después de instalarlo global. Causa: la carpeta de paquetes globales no está en tu PATH, así que la terminal no encuentra el comando. Solución: verifica dónde instala npm lo global con `npm root -g`, verifica que la carpeta bin correspondiente esté en tu PATH con `echo $PATH`, y si no está, agrégala siguiendo el mismo procedimiento del error 3. En Windows, esto casi nunca pasa porque el instalador de Node.js configura el PATH solo; en Linux y Mac es más común.

La metodología general cuando algo falla con npm, y te la regalo porque sirve para todo: uno, lee el mensaje de error completo, no solo la primera línea; npm suele decir exactamente qué pasó cerca del final del mensaje. Dos, verifica lo básico: ¿estás en la carpeta correcta? ¿existe el `package.json`? Tres, prueba el remedio nuclear (borrar `node_modules` y reinstalar) antes de perder una hora investigando. Cuatro, si el error menciona un paquete específico, búscalo con `npm ls` para ver de dónde viene.

Buenas prácticas y seguridad básica: programa como profesional

Instalar paquetes es fácil; hacerlo con criterio es lo que distingue al profesional. Esta sección son las reglas no escritas que los equipos serios siguen todos los días.

Primera regla: revisa antes de instalar. Cada vez que vas a agregar un paquete a tu proyecto, tómate dos minutos para verificar: ¿cuántas descargas semanales tiene? (miles o millones es buena señal; unas pocas docenas es bandera roja), ¿cuándo fue su última actualización? (un paquete sin actualizar en tres años puede estar abandonado), ¿tiene buena documentación? Puedes ver todo esto en la página del paquete en npmjs.com. Instalar paquetes desconocidos sin revisar es como aceptar dulces de extraños: la mayoría de las veces no pasa nada, pero el día que pasa, pasa en grande.

Segunda regla: menos es más. Cada paquete que agregas es código que no escribiste, que no auditaste y que ahora es tu responsabilidad cuando algo falla. Antes de instalar un paquete pregúntate: "¿de verdad lo necesito, o puedo resolverlo con diez líneas de código propio?" Para cosas triviales (como una función que convierte texto a mayúsculas con formato especial), a veces es mejor escribirlo tú que depender de un paquete externo. El árbol de dependencias crece rápido; mantenlo bajo control instalando solo lo necesario.

Tercera regla: usa npm audit. Este comando revisa todos tus paquetes instalados contra una base de datos de vulnerabilidades de seguridad conocidas y te reporta qué paquetes tienen problemas, qué tan graves son (low, moderate, high, critical) y cómo arreglarlos. Ejecútalo periódicamente, sobre todo antes de desplegar a producción. Si encuentra vulnerabilidades, npm audit fix intenta corregirlas automáticamente actualizando a versiones seguras. No es magia: a veces la corrección requiere un cambio de mayor que debes hacer a mano. Pero es tu primera línea de defensa, gratis e incluida con npm. Acostúmbrate a correrlo como quien revisa el aceite del coche.

Cuarta regla: congela tus versiones en producción. Ya lo vimos con package-lock.json y npm ci: en desarrollo puedes ser flexible, pero cuando despliegas, quieres certeza. Usa el candado, súbelo a git y despliega con npm ci. "En mi máquina sí funciona" no es una estrategia de despliegue.

Quinta regla: no instales paquetes con sudo. Ya vimos por qué en la sección de errores: usar sudo con npm crea archivos con permisos de administrador dentro de tu proyecto y tu caché, y después todo lo demás falla por permisos. Si te pide sudo, el problema es tu configuración de carpetas globales, no la falta de privilegios. Arregla la causa, no el síntoma.

Sexta regla: mantén separado lo global de lo local. Global solo herramientas de línea de comandos (nodemon y similares). Todo lo que tu código importe con require va local en cada proyecto. Esta disciplina evita el clásico "pero si yo lo tengo instalado" cuando el proyecto truena en otra máquina.

Séptima regla: documenta tus scripts. Un README.md corto que explique qué hace cada script de tu package.json (qué hace npm start, qué hace npm run dev, cómo se corre en producción) vale oro para tu yo del futuro y para cualquiera que colabore contigo. Te toma cinco minutos escribirlo y te ahorra horas de "¿cómo se corría esto?".

Ejemplos paso a paso

Ejemplo 1: proyecto desde cero con npm init

Vamos a construir un proyecto real desde cero, paso a paso, como lo harías en tu computadora. Abre tu terminal y crea una carpeta de práctica:

```
mkdir proyecto-npm
cd proyecto-npm
npm init -y
```

Con npm init -y aceptas todos los valores por defecto y en un segundo tienes tu package.json. Ábrelo en tu editor y verás algo así:

```
{
  "name": "proyecto-npm",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no hay pruebas todavía\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

Ahora personalízalo un poco, como haría un profesional. Cambia la descripción, pon tu nombre como autor y agrega los scripts que usaremos en esta unidad. Déjalo así:

```
{
  "name": "proyecto-npm",
  "version": "1.0.0",
  "description": "Proyecto de práctica de la Unidad 5 del curso de Node.js",
  "main": "index.js",
  "scripts": {
    "start": "node index.js",
    "dev": "nodemon index.js",
    "test": "echo \"Error: no hay pruebas todavía\" && exit 1"
  },
  "keywords": ["practica", "nodejs", "npm"],
  "author": "Tu Nombre",
  "license": "MIT"
}
```

Guarda el archivo. Acabas de hacer lo que todo proyecto Node.js serio hace en su primer minuto de vida. Ahora crea el archivo `index.js` con un saludo simple:

```
console.log("Mi primer proyecto con npm funciona.");
```

Ejecútalo con `node index.js` para verificar que todo está en orden. Si ves el mensaje en la terminal, tu proyecto nació sano.

Ejemplo 2: instalar y usar lodash

lodash es una de las librerías de utilidades más famosas del mundo JavaScript. Trae decenas de funciones para trabajar con arreglos, objetos, textos y números: cosas que harías a mano con mucho código, resueltas en una línea. Es el ejemplo perfecto de "pieza de refaccionaria": en vez de escribir tu propia función para mezclar un arreglo o eliminar duplicados, usas la de lodash, probada por millones de programadores.

Instálalo en tu proyecto:

```
npm install lodash
```

Observa lo que pasa en la terminal: npm descarga el paquete, lo guarda en `node_modules` y actualiza tu `package.json`. Ábrelo y verás que apareció una sección nueva:

```
"dependencies": {  
  "lodash": "^4.17.21"  
}
```

Esa línea es tu recibo de la refaccionaria: "este proyecto necesita lodash, versión 4.17.21 o compatible". También se creó el archivo `package-lock.json` con el detalle exacto de lo instalado.

Ahora úsalo en tu `index.js`. Vamos a reescribir el archivo para probar varias funciones de lodash:

```

const _ = require("lodash");

// _.chunk divide un arreglo en grupos del tamaño indicado
const numeros = [1, 2, 3, 4, 5, 6, 7];
console.log("En grupos de 3:", _.chunk(numeros, 3));

// _.uniq elimina los duplicados de un arreglo
const repetidos = [1, 2, 2, 3, 3, 3, 4];
console.log("Sin duplicados:", _.uniq(repetidos));

// _.shuffle mezcla un arreglo al azar
console.log("Mezclados:", _.shuffle(numeros));

// _.capitalize pone en mayúscula la primera letra
console.log(_.capitalize("hola mundo desde lodash"));

```

Por convención, lodash se importa con el nombre `_` (guion bajo); es una tradición de la librería desde sus inicios. Ejecuta con `node index.js` y verás cada función en acción. Fíjate en algo: no escribiste la lógica de dividir en grupos ni de eliminar duplicados; la trajiste hecha, probada y documentada. Eso es exactamente el poder de npm: en un comando pasaste de "tendría que programarlo" a "ya está resuelto".

Ejemplo 3: instalar y usar colors

El paquete `colors` es pequeño pero muy ilustrativo: le agrega colores a los mensajes de tu terminal. Parece un lujo, pero en proyectos reales los colores sirven para distinguir errores (rojo), advertencias (amarillo) y éxitos (verde) de un vistazo, sin tener que leer cada línea.

Instálalo:

```
npm install colors
```

Tu `package.json` ahora tiene dos dependencias: `lodash` y `colors`. Úsalo en tu código:

```

const colors = require("colors");

console.log("Todo salió bien".green);
console.log("Cuidado con esto".yellow);
console.log("Algo falló".red);
console.log("Mensaje informativo".blue);

```

La magia de colors es que extiende los textos de JavaScript con propiedades de color: "texto".green, "texto".red, etcétera. Ejecuta con node index.js y verás tu terminal a colores. Combínalo con lodash en un ejemplo más realista, como un pequeño reporte:

```
const _ = require("lodash");
require("colors");

const ventas = [1200, 800, 1500, 300, 2000, 450];
const total = _.sum(ventas);
const promedio = _.mean(ventas);

console.log("=== REPORTE DE VENTAS ===".cyan);
console.log("Total: $" + total);
console.log("Promedio: $" + promedio.toFixed(2));
console.log("Mejor venta: $" + _.max(ventas));

if (total > 5000) {
  console.log("Meta superada, felicidades".green);
} else {
  console.log("Meta no alcanzada".red);
}
```

Con dos paquetes instalados en dos comandos construiste un reporte con cálculos estadísticos y salida a colores. Intenta imaginar cuánto código te habría tomado hacerlo todo a mano. Esa es la economía de npm.

Ejemplo 4: scripts start y dev personalizados

Ya tienes los scripts "start" y "dev" definidos en tu package.json desde el ejemplo 1. Vamos a usarlos y a entender la diferencia. El script "start" corre tu app en modo normal con node; el script "dev" la corre con nodemon para desarrollo cómodo.

Primero asegúrate de tener nodemon instalado globalmente:

```
npm install -g nodemon
```

Ahora, en vez de escribir node index.js, ejecuta:

```
npm start
```

Hace exactamente lo mismo que `node index.js`, porque así lo definiste en el script. La ventaja no se nota con comandos cortos, pero imagina un comando real de producción con variables de entorno y banderas: el script lo esconde todo detrás de dos palabras.

Ahora ejecuta:

```
npm run dev
```

Nodemon arranca tu `index.js` y se queda vigilando. Verás el mensaje "[nodemon] starting node index.js". Deja la terminal abierta, ve a tu editor, cambia algún `console.log` de tu `index.js`, guarda el archivo y mira la terminal: nodemon detectó el cambio y reinició tu programa solo, mostrando la salida actualizada. Edita, guarda, observa. Repite tres o cuatro veces hasta que el flujo se te haga natural. Este es el ciclo de desarrollo moderno: escribes, guardas, ves el resultado, sin tocar la terminal.

Para detener nodemon, presiona `Ctrl+C` en la terminal, igual que con cualquier programa de Node.

Un detalle profesional: si trabajas en equipo, documenta en tu README qué hace cada script, con una pequeña sección de comandos. Algo tan simple como esto basta:

```
Comandos del proyecto:  
- npm start: corre la app en modo producción  
- npm run dev: corre la app con nodemon (reinicio automático)
```

Tu yo del futuro te lo va a agradecer cuando regreses al proyecto después de tres meses.

Ejemplo 5: configurar nodemon con nodemon.json

Para proyectos pequeños, nodemon sin configuración es suficiente. Pero supón que tu proyecto crece: tienes una carpeta `src` con tu código, una carpeta `datos` donde tu app guarda archivos JSON, y no quieres que cada vez que la app escriba un dato se reinicie el servidor. Ahí entra `nodemon.json`.

Primero, reorganiza tu proyecto de ejemplo para simular un proyecto real:

```
mkdir src  
mkdir datos
```

Mueve tu `index.js` a `src/index.js` (o crea uno nuevo ahí). Actualiza tu script "dev" en el `package.json` para que apunte a la nueva ubicación:

```
"dev": "nodemon src/index.js"
```

Ahora crea el archivo nodemon.json en la raíz del proyecto:

```
{
  "watch": ["src"],
  "ext": "js,json",
  "ignore": ["src/datos/*.json"],
  "delay": 1000,
  "verbose": true
}
```

Con esta configuración le dices a nodemon: vigila solo la carpeta src, reacciona a cambios en archivos .js y .json, ignora los JSON dentro de src/datos (donde tu app escribe datos), espera un segundo antes de reiniciar (por si tu editor guarda en ráfaga) y cuéntame con detalle lo que haces.

Pruébalo: ejecuta `npm run dev`, edita `src/index.js` y observa el reinicio automático. Luego crea un archivo JSON dentro de `src/datos` y edítalo varias veces: verás que nodemon NO se reinicia, porque lo excluiste. Esa es la configuración trabajando para ti: reinicios cuando cambias código, silencio cuando cambian datos.

Con estos cinco ejemplos ya recorriste el flujo completo de un proyecto Node.js moderno: inicializar, instalar dependencias, usarlas, automatizar con scripts y desarrollar con nodemon. Todo lo que sigue en la práctica es ejercitar estos mismos movimientos hasta que te salgan sin pensar.

Práctica

Ejercicio 1: inicializa un proyecto y descríbelo

Crea una carpeta llamada "calculadora-pro", entra en ella, inicializa un proyecto npm aceptando todo por defecto y luego edita el `package.json` para que quede con estos datos: descripción "Calculadora de línea de comandos", autor con tu nombre, licencia MIT, y agrega los scripts "start" ("`node index.js`") y "dev" ("`nodemon index.js`"). Crea el archivo `index.js` con un `console.log` de prueba y verifica que `npm start` lo ejecute.

Solución paso a paso. Primero, los comandos en la terminal:

```
mkdir calculadora-pro
cd calculadora-pro
npm init -y
```

Con `npm init -y` nace el `package.json` con valores por defecto. Ahora ábrelo en tu editor y edítalo para que quede así:

```
{
  "name": "calculadora-pro",
  "version": "1.0.0",
  "description": "Calculadora de línea de comandos",
  "main": "index.js",
  "scripts": {
    "start": "node index.js",
    "dev": "nodemon index.js",
    "test": "echo \"Error: no hay pruebas todavía\" && exit 1"
  },
  "keywords": [],
  "author": "Tu Nombre",
  "license": "MIT"
}
```

Los cambios que hiciste: llenaste `"description"` para que cualquiera sepa qué es el proyecto, pusiste tu nombre en `"author"`, cambiaste la licencia a `"MIT"` (la más común en proyectos abiertos) y agregaste los scripts `"start"` y `"dev"` siguiendo la convención de la industria. Ahora crea el archivo `index.js`:

```
console.log("Calculadora Pro lista.");
```

Y verifica con `npm start`. Si ves el mensaje `"Calculadora Pro lista."` en la terminal, el ejercicio está completo. Lo que practicaste aquí es el ritual de inicio de todo proyecto Node.js: es tan fundamental que después de hacerlo diez veces te saldrá sin pensar.

Ejercicio 2: instala `lodash` y construye un analizador de calificaciones

En tu proyecto `"calculadora-pro"`, instala `lodash` con `npm install lodash`. Luego crea un archivo llamado `calificaciones.js` que haga lo siguiente: tenga un arreglo con 10 calificaciones (algunas repetidas), use `_.uniq` para obtener las calificaciones únicas, use `_.mean` para calcular el promedio, use `_.max` y `_.min` para la más alta y la más baja, y use `_.sortBy` para mostrarlas ordenadas. Imprime todo con `console.log` y ejecútalo con `node calificaciones.js`.

Solución paso a paso. Primero instala el paquete:

```
npm install lodash
```

Verifica en tu package.json que apareció la sección "dependencies" con la línea de lodash. Ahora crea calificaciones.js:

```
const _ = require("lodash");

const calificaciones = [8, 9, 7, 10, 8, 6, 9, 7, 10, 8];

console.log("Calificaciones:", calificaciones);
console.log("Únicas:", _.uniq(calificaciones));
console.log("Promedio:", _.mean(calificaciones).toFixed(2));
console.log("Más alta:", _.max(calificaciones));
console.log("Más baja:", _.min(calificaciones));
console.log("Ordenadas:", _.sortBy(calificaciones));
```

Ejecuta con node calificaciones.js. Verás el análisis completo calculado en seis líneas gracias a lodash. Fíjate en lo que NO tuviste que escribir: la lógica para eliminar duplicados, para sumar y promediar, para ordenar. Cada una de esas funciones te habría tomado varios minutos y posibles errores; lodash te las dio probadas. Este ejercicio te enseña el patrón mental clave de npm: ante un problema común, primero pregunta si ya existe una pieza que lo resuelva.

Ejercicio 3: crea un script personalizado que combine tareas

Agrega a tu package.json un script llamado "reporte" que ejecute node calificaciones.js, y otro script llamado "todo" que primero ejecute el reporte y luego arranque la app principal, usando el operador &&. Pruébalos con npm run reporte y npm run todo. Explica con tus palabras qué hace el operador &&.

Solución paso a paso. Edita la sección "scripts" de tu package.json para que quede así:

```
"scripts": {
  "start": "node index.js",
  "dev": "nodemon index.js",
  "reporte": "node calificaciones.js",
  "todo": "npm run reporte && npm start",
  "test": "echo \"Error: no hay pruebas todavía\" && exit 1"
}
```


El script "reporte" es directo: corre el analizador de calificaciones. El script "todo" es más interesante: ejecuta `npm run reporte` y, si ese comando termina con éxito, ejecuta `npm start`. El operador `&&` encadena comandos con la condición de que cada uno debe tener éxito para que continúe la cadena; si el primero falla, el segundo no se ejecuta. Es una forma simple de decir "haz esto y luego esto otro, pero solo si lo primero salió bien".

Pruébalos:

```
npm run reporte
npm run todo
```

Con el primero verás solo el reporte de calificaciones. Con el segundo verás primero el reporte y después el mensaje de la app principal, todo con un solo comando. Lo que practicaste es la automatización de flujos de trabajo: en proyectos reales, los scripts encadenan pasos como "compilar, probar y arrancar", y todo el equipo los usa igual sin memorizar comandos largos.

Ejercicio 4: configura nodemon para un proyecto con datos

Simula un proyecto real: crea las carpetas `src` y `datos` dentro de "calculadora-pro", mueve `calificaciones.js` a `src/`, crea un archivo `nodemon.json` que vigile solo `src`, ignore todo lo que esté en `datos`, espere 1 segundo antes de reiniciar y muestre salida detallada. Actualiza el script "dev" para que apunte a `src/calificaciones.js`. Luego prueba: ejecuta `npm run dev`, modifica `src/calificaciones.js` y confirma el reinicio; después crea un archivo en `datos/`, modifícalo y confirma que NO hay reinicio.

Solución paso a paso. Primero la reorganización:

```
mkdir src
mkdir datos
```

Mueve `calificaciones.js` a `src/` (córtalo y pégalo con tu editor o tu administrador de archivos). Crea `nodemon.json` en la raíz del proyecto:

```
{
  "watch": ["src"],
  "ext": "js,json",
  "ignore": ["datos/*"],
  "delay": 1000,
  "verbose": true
}
```

Actualiza el script "dev" en tu package.json:

```
"dev": "nodemon src/calificaciones.js"
```

Ahora la prueba en dos partes. Parte uno: ejecuta `npm run dev` en la terminal y déjala abierta. Ve a tu editor, cambia algo en `src/calificaciones.js` (por ejemplo agrega un `console.log`), guarda y observa la terminal: nodemon muestra que detectó el cambio y reinicia. Parte dos: crea el archivo `datos/registro.json` con cualquier contenido, guárdalo, modifícalo varias veces y observa la terminal: nodemon no se reinicia, porque la carpeta `datos` está en la lista de ignorados.

Este ejercicio te enseña la configuración de nodemon que usarás en proyectos reales: vigilar el código, ignorar los datos que la app escribe sola. Sin esta configuración, muchas aplicaciones caen en el ciclo infinito de reinicios que vimos en la sección de errores. Saber configurar `nodemon.json` es lo que separa al que "usa nodemon" del que "domina nodemon".

Ejercicio 5: auditoría y limpieza de dependencias

En tu proyecto, ejecuta `npm outdated` para ver el estado de tus paquetes, luego ejecuta `npm audit` para revisar vulnerabilidades. Después práctica el ciclo de desinstalación: desinstala `colors` si lo habías instalado (`npm uninstall colors`), verifica que desapareció del `package.json` y de `node_modules`, y vuelve a instalarlo. Finalmente, borra la carpeta `node_modules` por completo y reconstrúyela con `npm install`, verificando que todo vuelve a funcionar.

Solución paso a paso. Primero el diagnóstico:

```
npm outdated
```

Si tus paquetes están recién instalados, probablemente no muestre nada (todo está al día). Si muestra alguna línea, verás las columnas de versión actual, "wanted" y "latest" que estudiamos. Luego la auditoría:

```
npm audit
```

Te mostrará un reporte de vulnerabilidades encontradas (con suerte, cero) y sugerencias de corrección. Acostúmbrate a correrlo antes de cada despliegue importante.

Ahora la desinstalación. Si instalaste `colors` en algún ejemplo previo:

```
npm uninstall colors
```

Verifica tres cosas: que la carpeta `node_modules/colors` ya no existe, que la línea de `colors` desapareció de la sección "dependencies" de tu `package.json`, y que el `package-lock.json` se actualizó. Si tu código todavía hace `require("colors")`, fallará con "Cannot find module": esa es la prueba de que la desinstalación fue completa y de que el `require` siempre busca en el `node_modules` local.

Vuelve a instalarlo:

```
npm install colors
```

Y ahora el ejercicio más valioso de todos, el remedio nuclear:

```
rm -rf node_modules
npm install
```

Borraste toda la bodega y la reconstruiste desde la receta con un solo comando. Verifica que tu `package.json` sigue intacto, que `node_modules` regresó con `lodash` y `colors` adentro, y que `node src/calificaciones.js` (o tu archivo principal) sigue funcionando igual que antes. Lo que acabas de comprobar con tus propias manos es la idea central de esta unidad: `node_modules` es desechable y reconstruible; el `package.json` es lo valioso. Quien entiende esto jamás entra en pánico cuando algo falla con las dependencias.

Ejercicio 6: instala un paquete global y úsalo desde cualquier carpeta

Instala `nodemon` de forma global con `npm install -g nodemon` (si aún no lo tienes), verifica la instalación con `nodemon --versión`, y luego comprueba que funciona fuera de tu proyecto: crea una carpeta temporal en otro lugar de tu computadora, escribe un archivo `hola.js` con un `console.log`, y ejecútalo con `nodemon hola.js` desde esa carpeta. Finalmente, lista tus paquetes globales con `npm list -g --depth=0`.

Solución paso a paso. La instalación global:

```
npm install -g nodemon
nodemon --version
```

El primer comando instala `nodemon` en la carpeta global del sistema; el segundo confirma que quedó disponible como comando. Si te da error de permisos (EACCES) en Linux o Mac, aplica la solución de la sección de errores: configura la carpeta `~/.npm-global` y actualiza tu `PATH`, sin usar `sudo`.

Ahora la prueba de que es global de verdad:

```
mkdir /tmp/prueba-global  
cd /tmp/prueba-global
```

Crea el archivo hola.js con este contenido:

```
console.log("Nodemon global funcionando.");
```

Y ejecútalo:

```
nodemon hola.js
```

Funciona aunque esta carpeta no tiene package.json ni node_modules propio: esa es exactamente la gracia de lo global. Nodemon vive en tu sistema, no en tu proyecto. Deténlo con Ctrl+C.

Finalmente, el inventario:

```
npm list -g --depth=0
```

Verás la lista de tus herramientas globales (nodemon entre ellas) sin el árbol de dependencias. Este ejercicio te deja clara la distinción más práctica de la unidad: lo global son tus herramientas personales del sistema; lo local son las piezas de cada proyecto. Si algún día tu código hace require de algo y truena con "Cannot find module", tu primer sospechoso ya sabes cuál es: lo instalaste global cuando debía ir local.

Resumen de la unidad

Llegaste al final de la unidad más práctica del curso hasta ahora. Vamos a recapitular los puntos clave, porque son muchos y todos importantes.

npm es dos cosas: la herramienta de línea de comandos que instalas paquetes y el registro en línea más grande del mundo donde viven millones de piezas de código gratis. La analogía que te acompaña desde hoy: npm es la refaccionaria del programador; en vez de fabricar cada pieza, la pides con un comando y la instalas.

Todo proyecto serio empieza con npm init (o npm init -y para aceptar todo por defecto). Ese comando crea el package.json, el acta de nacimiento y la receta de tu proyecto. Sus campos principales son: "name" (nombre en minúsculas sin espacios), "versión" (versión semántica

empezando en 1.0.0), "description" (qué hace el proyecto), "main" (archivo de entrada, por defecto index.js), "scripts" (tus atajos de comandos), "keywords", "author", "license", "dependencies" (piezas necesarias para funcionar) y "devDependencies" (piezas solo para desarrollar).

El versionado semántico usa tres números: MAJOR.MINOR.PATCH. El major sube cuando se rompe compatibilidad (actualiza con cuidado), el minor cuando se agregan funciones sin romper nada, el patch cuando se corrigen errores. En el package.json verás rangos: ^ permite subir minor y patch pero no major, ~ permite solo patch, y la versión exacta sin símbolo congela todo.

npm install tiene dos modos: con nombre de paquete (npm install lodash) descarga la pieza y la anota en tu package.json; sin nombre (npm install a secas) reconstruye todo el proyecto leyendo el package.json. Es el comando que usas al clonar un proyecto o cambiar de máquina.

node_modules es la bodega donde viven los paquetes descargados. Es pesada, reconstruible y específica de tu sistema: NUNCA se sube a git, nunca se manda por correo, nunca se respalda. Si se corrompe, la borras y la reconstruyes con npm install. El package-lock.json, en cambio, SÍ se sube siempre: congela las versiones exactas instaladas para que todo el equipo y el servidor tengan el mismo árbol. Para instalaciones estrictas en producción existe npm ci.

Las dependencias normales van en "dependencies" (el proyecto no funciona sin ellas) y las de desarrollo en "devDependencies" (solo comodidad al programar; se instalan con npm install --save-dev). La instalación local (sin bandera) pone el paquete en el node_modules de tu proyecto y es lo correcto para todo lo que tu código importe con require. La instalación global (npm install -g) pone herramientas de línea de comandos disponibles en todo tu sistema, como nodemon.

Los scripts del package.json convierten comandos largos en atajos cortos: npm start y npm run dev son los más comunes, con la convención de que "start" es para producción y "dev" para desarrollo. Se pueden encadenar con && y documentan tu proyecto. npx, el primo de npm, ejecuta paquetes sin instalarlos: ideal para herramientas de un solo uso o para correr ejecutables locales sin escribir rutas largas.

El mantenimiento se hace con npm uninstall (quita el paquete de node_modules y del package.json), npm outdated (muestra qué se puede actualizar) y npm update (actualiza respetando tus rangos ^ y ~). Actualiza con estrategia: parches y minors sin miedo, majors con lectura previa de cambios.

Nodemon es tu compañero de desarrollo: se instala global con `npm install -g nodemon`, se usa con `nodemon index.js` (o mejor, con el script `npm run dev`) y reinicia tu servidor automáticamente cada vez que guardas. Se configura con `nodemon.json`: "watch" para elegir qué vigilar, "ext" para las extensiones, "ignore" para excluir archivos de datos y evitar ciclos infinitos de reinicio, "delay" para esperar antes de reiniciar y "verbose" para más detalle. Con "rs" + Enter fuerzas un reinicio manual.

El `.gitignore` esencial de Node ignora `node_modules/`, `.env` (tus secretos nunca se suben), `.DS_Store`, `*.log`, `dist/` y `coverage/`. Créalo antes del primer commit.

Los errores más comunes ya los conoces con su solución: "Cannot find module" se arregla instalando el paquete local; los conflictos de versiones se investigan con `npm ls` y se resuelven alineando versiones o reinstalando limpio; los permisos con `-g` en Linux/Mac se arreglan configurando `~/.npm-global` en vez de usar `sudo`; la caché corrupta se limpia con `npm cache clean --force`; y nodemon en ciclo infinito se calma agregando archivos al "ignore".

Las buenas prácticas que te hacen profesional: revisa un paquete antes de instalarlo (descargas, mantenimiento, documentación), instala solo lo necesario, corre `npm audit` periódicamente, despliega con versiones congeladas usando `npm ci`, nunca uses `sudo` con `npm`, separa lo global de lo local y documenta tus scripts en el README.

Con esta unidad completaste las clases 13, 14 y 15 del curso: sabes instalar paquetes como `lodash` con `npm install`, entiendes el `package.json` como la receta de tu proyecto con su nombre, sus scripts y sus dependencias, y dominas nodemon para desarrollar sin parar y arrancar a mano. A partir de aquí, cada curso o proyecto que toques te pedirá estos mismos movimientos: inicializar, instalar, configurar scripts y desarrollar con reinicio automático. Ya no son comandos misteriosos: son tus herramientas de todos los días. En la próxima unidad seguiremos construyendo sobre esta base. Nos vemos ahí.

UNIDAD 6: El corazón asíncrono: eventos, streams y buffers

Hasta este punto del curso has aprendido a encender el motor de Node.js: sabes qué es, cómo instalarlo, cómo correr un archivo, cómo usar módulos con `require` y `module.exports`, cómo organizar tu código, cómo instalar paquetes con `npm` y cómo leer y escribir archivos con el módulo `fs`. También levantaste tu primer servidor HTTP y respondiste peticiones de un navegador. Todo eso es la carrocería del coche. En esta unidad vamos a abrir el cofre y a mirar el motor de verdad: el mecanismo interno que hace que Node.js sea tan rápido y tan distinto de otros lenguajes del lado del servidor. Ese mecanismo tiene tres piezas, y las tres trabajan juntas como un solo organismo: los eventos, los streams y los buffers.

La idea central de esta unidad se puede decir en una frase: Node.js no espera, reacciona. Cuando un programa tradicional necesita leer un archivo gigante o recibir datos de una red, lo normal es que se quede parado esperando a que los datos lleguen completos para después procesarlos. Node.js hace lo contrario: en cuanto llega un pedacito de datos, avisa con un evento y sigue trabajando con lo que ya tiene. Esa filosofía de "avísame cuando pase algo" en lugar de "me quedé aquí esperando" es lo que le permite a un solo proceso de Node atender miles de conexiones al mismo tiempo sin ahogarse. Los eventos son el sistema nervioso, los streams son las tuberías por donde fluyen los datos, y los buffers son los pedacitos de datos crudos que viajan por esas tuberías. Cuando entiendas cómo se conectan las tres piezas, vas a entender por qué Node.js se comporta como se comporta, y vas a poder escribir código que aproveche esa arquitectura en lugar de pelear contra ella.

Esta unidad cubre las clases 16, 17 y 18 del curso. La clase 16 está dedicada a `EventEmitter`, la clase del módulo `events` que es, literalmente, el corazón del estilo de programación de Node. Vas a aprender a crear tus propios emisores de eventos, a suscribir oyentes con `on`, a disparar eventos con `emit`, y a usar variantes como `once` y `removeListener`. La clase 17 entra a los streams: vas a entender qué es un stream, por qué existe, cuáles son sus tipos (legible, escribible, dúplex y de transformación), cómo se encadenan con `pipe`, y vas a ver el ejemplo estrella de la unidad: servir un archivo de video con `fs.createReadStream("grande.mp4").pipe(res)`, explicado línea por línea para que veas por qué los datos viajan por pedazos sin cargar todo en memoria. La clase 18 cierra con los buffers: qué son los datos binarios, cómo crearlos con `Buffer.from` y `Buffer.alloc`, cómo convertirlos a texto con distintos encodings como `utf8`, `hex` y `base64`, y por qué existen si ya tenemos los strings. Al final de la unidad, los tres temas se juntan en una sola imagen mental: los streams mueven buffers, y los eventos te avisan de todo lo que pasa en el camino.

¿Por qué importa todo esto para ti como desarrollador? Por tres razones muy prácticas. Primera: casi todo lo que haces en Node pasa por eventos aunque no lo veas. El servidor que creaste con `http.createServer` reacciona al evento de "llegó una petición". Los sockets, las conexiones a bases de datos y los temporizadores funcionan igual. Si entiendes el patrón emisor/oyente, dejas de programar "a ciegas" y empiezas a leer el código de Node como quien lee su propio idioma. Segunda: los streams son la diferencia entre un programa que se cae con un archivo de 2 GB y uno que lo procesa sin despeinarse. Cualquier aplicación seria (subir videos, procesar logs, comprimir respaldos, transmitir en vivo) usa streams por debajo, y saber usarlos te pone en otro nivel. Tercera: los buffers aparecen en cuanto trabajas con archivos binarios, imágenes, video, cifrado o protocolos de red. Si alguna vez te topaste con un objeto raro lleno de números hexadecimales en la consola y no supiste qué hacer con él, esta unidad te da la respuesta.

El panorama de la unidad es el siguiente. En la explicación detallada empezaremos con el patrón emisor/oyente usando la analogía de la radio: una estación emite y muchos radios sintonizan, y nadie le pregunta a la estación "¿ya transmitiste?", simplemente la escuchan. Después desmenuzaremos `EventEmitter` método por método: `on`, `emit`, `once`, `removeListener`, cómo pasar argumentos a los eventos, los eventos de error y por qué siempre hay que escucharlos (si no lo haces, el proceso truena), y cómo crear tus propias clases que heredan de `EventEmitter` para que tus objetos "hablen". Veremos también cómo Node usa eventos por dentro: los servidores y los sockets son emisores, y tu código de todos los días ya es código orientado a eventos sin que te dieras cuenta. Luego pasaremos a los streams con la analogía de la tubería contra la cubeta: por qué mover agua con una tubería (poco a poco, sin parar) le gana a cargarla en cubetas (todo de golpe, con esfuerzo). Cubriremos los cuatro tipos de streams, el método `pipe` y el encadenamiento, los eventos `data`, `end` y `error`, y por qué los streams salvan la memoria con archivos gigantes. El ejemplo de servir video con `createReadStream().pipe(res)` lo desarmaremos línea por línea. Después entraremos a los buffers: qué son los datos binarios, `Buffer.from`, `Buffer.alloc`, la conversión a texto con encodings, la concatenación de buffers, y la pregunta honesta de por qué existen si ya hay strings. Cerraremos la explicación con la relación entre los tres conceptos y con los errores comunes (no escuchar "error", fugas de memoria con demasiados listeners, confundir encodings) y sus soluciones. En los ejemplos paso a paso construiremos cuatro programas completos: un sistema de notificaciones con `EventEmitter`, una clase `Tienda` que emite eventos, un servidor que transmite video con streams, y la conversión de una imagen a base64 con buffers. La práctica te traerá cinco ejercicios con solución explicada paso a paso, y el resumen cerrará con los puntos clave que debes llevarte grabados.

Un consejo de colega antes de empezar: esta unidad es conceptual, pero no te quedes solo en la lectura. Cada ejemplo que veas, cópialo, ejecútalo, rompe algo a propósito y vuelve a

arreglarlo. Los eventos se entienden escuchándolos, los streams se entienden viéndolos fluir, y los buffers se entienden tocándolos. Si algo no te cuadra a la primera, sigue leyendo: los tres temas se iluminan entre sí, y lo que parezca abstracto en la sección de eventos se volverá concreto cuando veas los streams mover buffers de verdad. Vamos a darle.

Explicación detallada

El patrón emisor/oyente: la radio como maestra

Antes de tocar una sola línea de código, necesitas grabarte un patrón mental, porque todo lo demás de esta unidad cuelga de él. El patrón se llama emisor/oyente, en inglés emitter/listener, y también lo vas a ver con el nombre publicador/suscriptor. La idea es ridículamente simple: hay un objeto que emite avisos cuando algo pasa, y hay otros objetos (o funciones) que se suscriben para enterarse. El emisor no sabe quién lo escucha, no le importa cuántos son, y no espera respuesta. Simplemente grita al aire: "¡pasó esto!". Quien esté sintonizando, reacciona. Quien no, sigue con su vida.

La mejor analogía es la radio. Una estación de radio transmite su señal al aire las 24 horas. No llama por teléfono a cada oyente para preguntarle "¿quieres escuchar la siguiente canción?". No espera a que alguien conteste. Simplemente emite. Del otro lado, tú enciendes tu radio, sintonizas la frecuencia y escuchas. Si apagas la radio, la estación sigue transmitiendo; no se entera ni le importa. Si cien mil personas sintonizan a la vez, la estación no trabaja cien mil veces más: emite una sola vez y todos reciben. Esa es la magia del patrón: el emisor hace su trabajo una vez, y cualquier cantidad de oyentes puede reaccionar, cada uno a su manera, sin que el emisor tenga que conocerlos.

Ahora contrasta eso con el estilo tradicional de "preguntar y esperar". Imagina que en lugar de radio tuvieras que marcarle por teléfono a la estación cada cinco segundos para preguntar "¿ya pusieron mi canción? ¿ya? ¿ya?". Eso es el polling, y es agotador e ineficiente. Mucho código de principiantes funciona así: un ciclo que revisa una y otra vez si algo cambió. El patrón emisor/oyente invierte la responsabilidad: en lugar de que tú preguntes, el sistema te avisa. Tú te suscribes una vez y te olvidas; cuando el evento ocurra, tu función se ejecuta sola. Esa inversión es exactamente la filosofía de Node.js: no te quedes esperando bloqueado, suscríbete y sigue con tu vida.

Llevemos la analogía al código que ya conoces. Cuando escribiste tu primer servidor HTTP, hiciste algo como esto:

```
const http = require("http");

const servidor = http.createServer((req, res) => {
  res.end("Hola mundo");
});

servidor.listen(3000);
```

Esa función que le pasas a `createServer` es, en el fondo, un oyente. Le estás diciendo al servidor: "cuando llegue una petición, ejecuta esto". Tú no estás en un ciclo preguntando "¿ya llegó una petición? ¿ya? ¿ya?". El servidor emite el aviso y tu función reacciona. Sin saberlo, ya estabas programando con eventos desde tu primer servidor. Lo que haremos en esta unidad es bajar un nivel y manejar ese mecanismo directamente, con tus propias manos, para que deje de ser magia y se vuelva herramienta.

El patrón emisor/oyente tiene tres ingredientes que debes distinguir con claridad. Primero, el evento mismo, que es simplemente un nombre: "saludo", "click", "datos", "error", "fin". El nombre lo eliges tú (salvo algunos nombres reservados como "error", que tienen un trato especial y los veremos a fondo). Segundo, el emisor, el objeto que tiene el poder de disparar eventos con el método `emit`. Tercero, los oyentes, las funciones que se registran con el método `on` para reaccionar cuando el evento ocurra. Un emisor puede tener cero, uno o cien oyentes para el mismo evento, y cada oyente recibe los mismos datos. Además, los eventos pueden llevar información: cuando el emisor dispara, puede mandar argumentos (un nombre, un objeto, un pedazo de datos), y cada oyente los recibe como parámetros de su función. Piensa en la radio otra vez: la estación no solo dice "¡atención!", también transmite la canción completa; los oyentes no solo se enteran de que algo pasó, reciben el contenido.

¿Por qué este patrón es "el corazón del estilo de Node"? Porque resuelve el problema central de un servidor: hay miles de cosas pasando al mismo tiempo (peticiones que llegan, archivos que se terminan de leer, datos que arriban por la red, temporizadores que se cumplen) y un solo hilo de ejecución para atenderlas. Si cada operación bloqueara el hilo hasta terminar, el servidor se quedaría congelado atendiendo a un solo cliente. Con eventos, cada operación avisa cuando termina y el hilo sigue libre para lo demás. El event loop de Node, ese motor que ya mencionamos en unidades anteriores, es en el fondo una máquina de repartir eventos: "se terminó de leer el archivo, avísale a quien lo pidió", "llegó una petición, avísale al servidor", "llegó un pedazo de datos, avísale al stream". Todo Node es una gran estación de radio con miles de frecuencias, y tu código son los radios sintonizados.

Hay una sutileza importante que te va a ahorrar confusiones: los eventos en Node son síncronos en su despacho. Cuando llamas a `emisor.emit("saludo")`, todos los oyentes

registrados se ejecutan en ese mismo momento, uno tras otro, en el orden en que se registraron, antes de que la siguiente línea de tu código corra. Emitir no es "mandar un mensaje que llegará después", es "llamar ahorita mismo a todos los suscritos". Esto sorprende a muchos principiantes que esperan un comportamiento asíncrono. La asincronía real viene de cuándo se dispara el evento (cuando el archivo termina de leerse, cuando llegan los datos), no del disparo en sí. Guárdate esa distinción: el aviso puede llegar en cualquier momento del futuro, pero una vez que llega, los oyentes corren de inmediato y en orden.

Otra sutileza: el emisor y los oyentes están desacoplados. El código que emite no necesita importar ni conocer el código que escucha. Esto permite una arquitectura muy limpia: un módulo se encarga de detectar cosas (llegó un pedido, se cayó la conexión, se completó la descarga) y emite eventos con nombres claros; otros módulos se suscriben a los eventos que les interesan y reaccionan. Si mañana agregas un módulo nuevo (por ejemplo, mandar un correo cada vez que llega un pedido), no tocas el código del emisor: solo agregas un oyente más. Esa es la base de los sistemas extensibles, y es la razón por la que tantas librerías de Node exponen su funcionalidad como eventos en lugar de como funciones que hay que llamar en cierto orden.

Para cerrar esta sección, un mapa mental que te servirá todo el curso: cada vez que veas en Node un objeto con métodos `on` y `emit`, estás viendo un `EventEmitter` o algo que se comporta como uno. Servidores HTTP, peticiones, respuestas, sockets TCP, streams de archivos, procesos hijos: todos hablan el idioma de los eventos. Aprender `EventEmitter` no es aprender "una clase más del manual", es aprender el idioma materno de Node.js. Y como todo idioma, se aprende usándolo, así que en la siguiente sección lo vamos a desarmar pieza por pieza con las manos en el código.

EventEmitter a fondo: on, emit, once y todo el arsenal

Vamos al código. La clase `EventEmitter` vive en el módulo `events`, que es parte del núcleo de Node, así que no necesitas instalar nada. La forma clásica de usarla es esta:

```
const EventEmitter = require("events");

const emisor = new EventEmitter();

emisor.on("saludo", (nombre) => {
  console.log("Hola, " + nombre + ". Bienvenido.");
});

emisor.emit("saludo", "Esteban");
```

Analicemos cada línea porque aquí está todo el patrón en miniatura. La primera línea importa la clase. La tercera crea una instancia: un emisor vacío, sin oyentes, esperando. La quinta registra un oyente con `on`: le dice al emisor "cuando ocurra el evento llamado saludo, ejecuta esta función". Nota que en este punto nada se ha ejecutado todavía; solo quedó anotada la suscripción, como sintonizar la radio antes de que empiece tu programa favorito. La última línea dispara el evento con `emit`: en ese instante, el emisor busca todos los oyentes registrados para "saludo" y los ejecuta uno por uno, pasándoles los argumentos que acompañan al `emit`. El resultado en consola es "Hola, Esteban. Bienvenido.". Si llamaras a `emit` tres veces, verías el saludo tres veces. Si nunca llamaras a `emit`, el oyente jamás se ejecutaría, aunque esté perfectamente registrado.

El método `on` es el caballo de batalla: registra un oyente permanente. Permanente significa que seguirá reaccionando cada vez que el evento se dispare, hasta que tú lo quites o el programa termine. Puedes registrar varios oyentes para el mismo evento, y se ejecutan en el orden en que los registraste:

```
emisor.on("saludo", (nombre) => {
  console.log("Oyente 1: hola " + nombre);
});

emisor.on("saludo", (nombre) => {
  console.log("Oyente 2: qué gusto verte, " + nombre);
});

emisor.emit("saludo", "María");
```

Esto imprime primero el mensaje del oyente 1 y luego el del oyente 2. El orden de registro es el orden de ejecución, garantizado. Y fíjate en algo poderoso: cada oyente hace algo distinto con el mismo evento. En una aplicación real, el evento "pedido-recibido" podría tener un oyente que guarda en la base de datos, otro que manda un correo y otro que actualiza el inventario. Tres reacciones independientes, un solo disparo. Esa es la extensibilidad de la que hablábamos: agregar comportamiento nuevo es agregar un `on`, no modificar código existente.

Los eventos pueden llevar tantos argumentos como quieras. El `emit` acepta el nombre del evento seguido de cualquier cantidad de valores, y cada oyente los recibe en el mismo orden:

```
emisor.on("compra", (producto, precio, cliente) => {
  console.log(cliente + " compró " + producto + " por $" + precio);
});

emisor.emit("compra", "laptop", 15000, "Ana");
```

Por convención, cuando el evento representa "algo salió mal", el primer argumento suele ser un objeto `Error`, siguiendo el estilo de los callbacks de Node que ya conoces. Y cuando el evento representa datos, es común mandar un solo objeto con toda la información, porque así el oyente no depende del orden de diez parámetros sueltos:

```
emisor.on("pedido", (datos) => {
  console.log("Nuevo pedido #" + datos.id + " de " + datos.cliente);
});

emisor.emit("pedido", { id: 1024, cliente: "Luis", total: 850 });
```

Ahora, el método `once`: registra un oyente que se ejecuta una sola vez y luego se desuscribe solo. Es perfecto para eventos que por naturaleza ocurren una vez, como "el servidor ya arrancó" o "la conexión se estableció":

```
emisor.once("listo", () => {
  console.log("El sistema está listo. Esto solo se imprime una vez.");
});

emisor.emit("listo");
emisor.emit("listo");
emisor.emit("listo");
```

Aunque dispares el evento tres veces, el mensaje aparece una sola vez. Por dentro, Node registra el oyente, y en cuanto se ejecuta la primera vez, lo elimina automáticamente. Es un detalle elegante que te ahorra escribir la lógica de "ya lo atendí, ignora los siguientes".

Para quitar oyentes manualmente tienes `removeListener` (y su alias `off`, que hace exactamente lo mismo). Necesitas pasarle el nombre del evento y la referencia exacta a la función que quieres quitar, por eso cuando planeas remover un oyente conviene guardar la función en una variable en lugar de usar una flecha anónima:

```
function manejarSaludo(nombre) {
  console.log("Hola " + nombre);
}

emisor.on("saludo", manejarSaludo);
emisor.emit("saludo", "Pedro");

emisor.removeListener("saludo", manejarSaludo);
emisor.emit("saludo", "Pedro");
```

El primer emit imprime el saludo; después de removerlo, el segundo emit no imprime nada porque ya no hay oyentes. Si hubieras registrado la función como flecha anónima directamente dentro del on, no tendrías cómo referirte a ella después para quitarla. Regla práctica: si un oyente va a vivir poco tiempo (por ejemplo, solo durante una petición), guárdalo en variable y remuévelo cuando termine; si va a vivir todo el programa (como los manejadores de un servidor), la flecha anónima está bien.

También existe `removeAllListeners("saludo")`, que quita todos los oyentes de un evento de un jalón, y sin argumentos quita todos los de todos los eventos. Úsalo con cuidado: es el botón de "apagar todas las radios", útil en limpieza de recursos pero peligroso si otro módulo dependía de esos oyentes.

Hablemos ahora del evento más importante y más traicionero: "error". En `EventEmitter`, "error" tiene un trato especial a nivel de lenguaje. Si un emisor dispara un evento "error" y no hay ningún oyente registrado para él, Node no lo ignora en silencio: lanza la excepción y tu proceso truena por completo. Mira:

```
const emisor2 = new EventEmitter();
emisor2.emit("error", new Error("Algo salió muy mal"));
```

Ese programa se muere con el error en la cara. En cambio, si hay al menos un oyente, todo sigue normal:

```
emisor2.on("error", (err) => {
  console.log("Se capturó el error: " + err.message);
});
emisor2.emit("error", new Error("Algo salió muy mal"));
```

Aquí el programa imprime el mensaje y continúa vivo. Esta regla existe por diseño: los errores no deben perderse en silencio. En Node, muchísimas operaciones asíncronas reportan sus fallos emitiendo "error" (leer un archivo que no existe, una conexión que se cae, un stream que se rompe). Por eso la regla de oro de esta unidad, grábala con fuego: siempre registra un oyente para "error" en cualquier emisor que uses. Es el equivalente a ponerte el cinturón de seguridad: no lo notas cuando todo va bien, pero el día que algo falla es lo único que te salva de un crash total. Más adelante, en la sección de errores comunes, veremos el desastre típico de olvidar este oyente en un servidor en producción.

Otra herramienta útil es el método `listenerCount`, que te dice cuántos oyentes hay registrados para un evento, y `eventNames`, que te devuelve la lista de eventos que tienen al menos un oyente. Son tus instrumentos de diagnóstico:

```
console.log(emisor.listenerCount("saludo"));
console.log(emisor.eventNames());
```

Y hay un detalle de protección que debes conocer: si registras más de 10 oyentes para el mismo evento en el mismo emisor, Node te muestra una advertencia en consola (`MaxListenersExceededWarning`). No es un error, el programa sigue, pero es Node diciéndote "oye, esto huele a fuga de memoria". El caso típico: registras un oyente nuevo dentro de una función que se ejecuta por cada petición, y nunca los remueves; con mil peticiones tienes mil oyentes acumulados. Si tu diseño realmente necesita más de 10, puedes subir el límite con `emisor.setMaxListeners(20)`, pero primero pregúntate si no estás acumulando oyentes sin querer. Veremos este problema a fondo en la sección de errores comunes.

Ahora subamos de nivel: lo verdaderamente poderoso es crear tus propias clases que heredan de `EventEmitter`, para que tus objetos "hablen" con eventos. Imagina una clase `Alarma` que emite "activada" cuando se dispara, o una clase `Descargador` que emite "progreso" cada vez que avanza. Con la sintaxis de clases de JavaScript es muy natural:

```
const EventEmitter = require("events");

class Temporizador extends EventEmitter {
  iniciar(segundos) {
    console.log("Temporizador iniciado: " + segundos + " segundos");
    setTimeout(() => {
      this.emit("terminado", segundos);
    }, segundos * 1000);
  }
}

const t = new Temporizador();

t.on("terminado", (s) => {
  console.log("¡Tiempo! Pasaron " + s + " segundos.");
});

t.iniciar(3);
```

Fíjate en lo que pasó: la clase `Temporizador` hereda de `EventEmitter`, así que cada instancia ya trae los métodos `on`, `emit` y `once` sin que escribieras nada. El método `iniciar` usa `setTimeout` (asíncrono) y, cuando el tiempo se cumple, dispara el evento "terminado" con `this.emit`. El código que usa el temporizador no necesita saber nada de `setTimeout` ni de la implementación interna: solo se suscribe al evento "terminado" y reacciona. Esa es la belleza del

desacoplamiento: la clase encapsula el "cómo" y expone el "qué pasó" mediante eventos. Cualquier objeto de tu aplicación que represente algo que "pasa en el tiempo" (una descarga, una partida de juego, un sensor, un pedido) es candidato a heredar de EventEmitter.

Un patrón muy común es que tus clases emitan varios eventos para las distintas etapas de un proceso: "inicio", "progreso", "terminado", "error". El consumidor se suscribe solo a los que le interesan:

```
class Descargador extends EventEmitter {
  descargar(archivo) {
    this.emit("inicio", archivo);
    let progreso = 0;
    const intervalo = setInterval(() => {
      progreso += 25;
      this.emit("progreso", progreso);
      if (progreso >= 100) {
        clearInterval(intervalo);
        this.emit("terminado", archivo);
      }
    }, 500);
  }
}

const d = new Descargador();
d.on("inicio", (a) => console.log("Descargando " + a + "..."));
d.on("progreso", (p) => console.log("Avance: " + p + "%"));
d.on("terminado", (a) => console.log(a + " descargado por completo."));
d.on("error", (e) => console.log("Falló: " + e.message));
d.descargar("video.mp4");
```

Este ejemplo, aunque simulado con setInterval, muestra la arquitectura real de muchísimas librerías de Node: el objeto hace su trabajo por dentro y te mantiene informado con eventos. Nota que incluimos el oyente de "error" aunque en este ejemplo nunca se dispare: es el cinturón de seguridad, siempre puesto.

Para cerrar la sección de EventEmitter, un resumen operativo de su API esencial. Para suscribirte: on (permanente) y once (una sola vez). Para disparar: emit(nombre, ...argumentos), que devuelve true si había al menos un oyente y false si nadie escuchaba (útil para saber si tu aviso cayó en el vacío). Para desuscribirte: removeListener u off con la referencia exacta a la función, y removeAllListeners para limpieza total. Para diagnosticar: listenerCount y eventNames, más setMaxListeners si necesitas ajustar el límite. Y la regla de oro: el evento "error" siempre debe tener oyente, porque un "error" sin oyente mata el proceso.

Con esto dominas la primera pieza del motor. En la siguiente sección veremos algo revelador: descubrirás que ya llevabas tiempo usando emisores sin saberlo, porque Node construye sus piezas más importantes sobre esta misma clase.

Cómo Node usa eventos por dentro: ya eras parte del club

Aquí viene una revelación que cambia cómo lees el código de Node: los objetos más importantes de la plataforma son EventEmitters disfrazados. El servidor HTTP que creaste en unidades anteriores, la petición (req), la respuesta (res), los sockets de red, los streams de archivos, los procesos hijos: todos heredan de EventEmitter o implementan su interfaz. Cuando escribiste `servidor.on("request", ...)` o cuando le pasaste un callback a `http.createServer`, estabas usando el patrón emisor/oyente sin que nadie te lo explicara. Esta unidad simplemente te está dando el manual de lo que ya hacías por instinto.

Mira un servidor escrito en estilo explícito de eventos, equivalente al que ya conoces:

```
const http = require("http");

const servidor = http.createServer();

servidor.on("request", (req, res) => {
  res.end("Hola desde el estilo explícito");
});

servidor.on("error", (err) => {
  console.log("Error del servidor: " + err.message);
});

servidor.listen(3000, () => {
  console.log("Servidor escuchando en el puerto 3000");
});
```

La diferencia con la versión que ya conocías es solo de forma: en lugar de pasarle la función a `createServer`, la registras con `on("request", ...)`. Por dentro es lo mismo: cada vez que llega una petición HTTP, el servidor emite el evento "request" con los objetos `req` y `res` como argumentos, y tu oyente reacciona. Y nota el oyente de "error": si el puerto 3000 ya está ocupado por otro programa, el servidor emite "error", y sin ese oyente tu proceso tronaría con un mensaje críptico. Con el oyente, capturas el problema con elegancia. Este es el primer lugar donde la regla de oro ("siempre escucha error") se vuelve real y no teórica.

Los objetos req y res también son emisores. La petición (req) es un stream de lectura: emite "data" cada vez que llega un pedazo del cuerpo de la petición y "end" cuando termina. Eso significa que puedes leer el cuerpo de un POST a mano, evento por evento:

```
servidor.on("request", (req, res) => {
  let cuerpo = "";
  req.on("data", (pedazo) => {
    cuerpo += pedazo;
  });
  req.on("end", () => {
    console.log("Cuerpo completo recibido: " + cuerpo);
    res.end("Recibido");
  });
  req.on("error", (err) => {
    console.log("Error leyendo la petición: " + err.message);
  });
});
```

Este fragmento es una joya pedagógica: muestra eventos anidados (el evento "request" del servidor contiene suscripciones a los eventos "data" y "end" de la petición), muestra cómo los datos llegan por pedazos y no de golpe, y muestra el oyente de "error" en su hábitat natural. Si vienes de otros lenguajes donde "leer el cuerpo de la petición" es una sola llamada bloqueante, esto te parecerá verboso; pero es exactamente este diseño el que permite a Node atender miles de peticiones sin bloquearse: mientras los pedazos van llegando, el hilo sigue libre para otros clientes.

Los sockets TCP del módulo net son otro ejemplo puro. Un socket emite "connect" cuando se conecta, "data" cuando llegan datos, "end" cuando el otro lado cierra, "error" si algo falla y "close" cuando la conexión termina del todo. Si alguna vez programas un chat, un juego multijugador o cualquier protocolo de red, vivirás suscrito a esos eventos. Y los procesos hijos (child_process) emiten "exit" cuando terminan y sus streams de salida emiten "data". El patrón está en todas partes porque es la forma en que Node le dice al mundo exterior: "avísame cuando pase algo, no me tengas esperando".

Entender esto te da un superpoder de lectura: cuando abras la documentación de cualquier módulo de Node y veas una lista de "Events", ya sabes exactamente qué significa y cómo usarlos. No hay que memorizar cada evento de cada clase; con saber el patrón, la documentación se vuelve predecible: buscas el nombre del evento, ves qué argumentos lleva, te suscribes con on, y listo. Esa es la diferencia entre "sé usar el módulo http" y "sé usar cualquier módulo de Node".

Streams a fondo: la tubería contra la cubeta

Cambiamos de pieza del motor. Los streams (flujos) son objetos que manejan datos que llegan poco a poco en el tiempo, en lugar de llegar todos juntos. La analogía que lo explica todo es la tubería contra la cubeta. Imagina que necesitas pasar un millón de litros de agua de un tanque a otro. Opción cubeta: llenas una cubeta gigante con todo el millón de litros (si es que cabe), la cargas con un esfuerzo brutal y la vacías del otro lado. Opción tubería: conectas una manguera entre los tanques y el agua fluye sola, litro por litro, sin que nadie cargue nada pesado. Los streams son la tubería. En lugar de cargar un archivo completo en memoria (la cubeta), los datos fluyen por pedazos (los chorros de agua) desde el origen hasta el destino, y tu programa procesa cada pedazo conforme llega.

¿Por qué importa? Por la memoria. La memoria RAM de tu servidor es finita y compartida entre todos los clientes. Si para servir un video de 2 GB tu programa tuviera que leer los 2 GB completos a memoria antes de empezar a enviarlos, con diez clientes simultáneos necesitarías 20 GB de RAM y el servidor se arrodillaría. Con streams, lees un pedacito (digamos 64 KB), lo envías, lo olvidas, lees el siguiente. La memoria usada se mantiene pequeña y constante sin importar si el archivo mide 2 MB o 200 GB. Esa es la frase que debes grabarte: los streams mantienen el uso de memoria constante frente a datos de tamaño arbitrario. Es la diferencia entre un programa que escala y uno que se cae el día que un usuario sube un archivo grande.

Node tiene cuatro tipos de streams y conviene distinguirlos con claridad. Los streams legibles (readable) son fuentes de datos: de ellos sale información. Ejemplos: `fs.createReadStream` para leer un archivo, la petición `req` en un servidor, la salida de un proceso hijo. Los streams escribibles (writable) son destinos: a ellos entra información. Ejemplos: `fs.createWriteStream` para escribir un archivo, la respuesta `res` en un servidor, la entrada de un proceso hijo. Los streams dúplex (duplex) son las dos cosas a la vez: se puede leer de ellos y escribir en ellos, como un socket de red (recibes y envías por la misma conexión). Y los streams de transformación (transform) son un tipo especial de dúplex que modifica los datos al pasar: entra texto, sale texto comprimido (como `zlib` para `gzip`), o entra un archivo y sale cifrado. Piensa en los transform como estaciones intermedias en la tubería que le hacen algo al agua mientras pasa.

El método estrella para conectar streams es `pipe` (tubería, literalmente). Tomas un stream legible y lo "enchufas" a un stream escribible, y Node se encarga de pasar los datos de uno a otro, pedazo por pedazo, manejando el ritmo para que el destino no se sature:

```
const fs = require("fs");

const lectura = fs.createReadStream("entrada.txt");
const escritura = fs.createWriteStream("salida.txt");

lectura.pipe(escritura);
```

Esas tres líneas copian un archivo del tamaño que sea usando memoria constante. Sin pipe tendrías que suscribirte a "data", escribir cada pedazo a mano, cuidar el evento "end" para cerrar, y manejar los errores de ambos lados. El pipe hace todo eso por ti, incluyendo algo sutil y crucial: el control de flujo (backpressure). Si el disco de escritura es más lento que el de lectura, pipe le dice al lector "aguanta, no me mandes más por ahora" en lugar de acumular pedazos en memoria hasta reventar. Ese control de ritmo automático es una de las razones por las que pipe es preferible a manejar los eventos a mano cuando solo quieres mover datos de un lado a otro.

Y lo mejor: los pipes se pueden encadenar. Como pipe devuelve el stream destino, puedes conectar varios en serie y armar una línea de ensamblaje donde los datos pasan por varias estaciones de transformación:

```
const fs = require("fs");
const zlib = require("zlib");

fs.createReadStream("entrada.txt")
  .pipe(zlib.createGzip())
  .pipe(fs.createWriteStream("entrada.txt.gz"));
```

Lee el archivo, lo comprime con gzip al vuelo y escribe el resultado comprimido, todo en streaming, sin cargar el archivo completo en memoria en ningún momento. Cada estación procesa pedacitos: el lector lee un pedazo, el compresor lo comprime, el escritor lo guarda, y vuelta a empezar. Esta es la arquitectura de las aplicaciones serias de procesamiento de datos: cadenas de streams que transforman información gigante con recursos modestos. Cuando veas herramientas que procesan logs de gigabytes o convierten videos, por debajo casi siempre hay tuberías como esta.

Ahora veamos los eventos de los streams, porque aquí las tres piezas de la unidad se tocan por primera vez. Un stream legible emite "data" cada vez que tiene un pedazo listo, "end" cuando ya no hay más datos, y "error" si algo falla. Puedes consumir un stream a mano suscribiéndote a esos eventos, que es exactamente lo que pipe hace por ti internamente:

```
const fs = require("fs");

const lector = fs.createReadStream("novela.txt", { encoding: "utf8" });

lector.on("data", (pedazo) => {
  console.log("Llegó un pedazo de " + pedazo.length + " caracteres");
});

lector.on("end", () => {
  console.log("Se terminó el archivo");
});

lector.on("error", (err) => {
  console.log("Error leyendo: " + err.message);
});
```

Observa el segundo argumento de `createReadStream`: `{ encoding: "utf8" }`. Sin él, los pedazos llegan como buffers (datos binarios crudos, la tercera pieza de la unidad, que veremos en la siguiente sección). Con el `encoding`, Node convierte cada pedazo a string automáticamente. Este es el primer encuentro entre streams y buffers: los streams mueven buffers por defecto, y los eventos te avisan de cada pedazo. Las tres piezas, trabajando juntas, en cinco líneas.

Los streams escribibles tienen sus propios eventos y métodos: `write(pedazo)` para mandar datos, `end()` para decir "ya no mandó más, cierra", y los eventos "finish" (cuando todo se escribió) y "error". Un patrón manual completo de copia de archivo se vería así:

```
const lector2 = fs.createReadStream("origen.txt");
const escritor = fs.createWriteStream("destino.txt");

lector2.on("data", (pedazo) => {
  const puedeSeguir = escritor.write(pedazo);
  if (!puedeSeguir) {
    lector2.pause();
    escritor.once("drain", () => lector2.resume());
  }
});

lector2.on("end", () => escritor.end());
lector2.on("error", (err) => console.log("Error leyendo: " + err.message));
escritor.on("error", (err) => console.log("Error escribiendo: " + err.message));
```

Este ejemplo es más largo que el pipe de tres líneas a propósito: te muestra el control de flujo manual (`pause/resume` con el evento "drain") que pipe hace automáticamente. No necesitas

memorizarlo, pero entenderlo te hace apreciar lo que pipe te ahorra y te prepara para los casos donde necesites control fino.

El ejemplo estrella: servir video con `createReadStream().pipe(res)`, línea por línea

Llegó el momento del ejemplo más famoso de esta unidad, el que aparece en la clase 17 del curso y que todo programador de Node termina usando. Un servidor que transmite un archivo de video sin cargarlo completo en memoria:

```
const http = require("http");
const fs = require("fs");

const servidor = http.createServer((req, res) => {
  res.writeHead(200, { "Content-Type": "video/mp4" });
  fs.createReadStream("grande.mp4").pipe(res);
});

servidor.listen(3000);
```

Siete líneas que hacen algo que en otros lenguajes requiere configuración especial. Desarmémoslo línea por línea.

Línea 1 y 2: importamos los módulos `http` (para el servidor) y `fs` (para el sistema de archivos). Nada nuevo, son los módulos que ya dominas.

Línea 4: creamos el servidor con `http.createServer` y le pasamos el manejador de peticiones. Recuerda la sección anterior: esa función es un oyente del evento "request"; se ejecuta una vez por cada petición que llegue, con `req` (lo que pide el cliente) y `res` (el canal para responderle).

Línea 5: `res.writeHead(200, { "Content-Type": "video/mp4" })`. Antes de mandar los datos, le decimos al cliente "todo bien (código 200) y lo que viene es un video MP4". El navegador usa esa cabecera para saber que debe reproducir video y no mostrar texto. Es como la etiqueta de la caja antes de empezar a sacar cosas.

Línea 6, la protagonista: `fs.createReadStream("grande.mp4").pipe(res)`. Aquí pasan tres cosas encadenadas. Primero, `fs.createReadStream("grande.mp4")` abre el archivo de video en modo streaming: no lo carga a memoria, solo prepara el lector y empieza a leer por pedazos (típicamente de 64 KB cada uno). Segundo, `.pipe(res)` enchufa ese lector al objeto `res`, que es un stream escribible (el canal de respuesta hacia el navegador). Tercero, a partir de ese momento y sin más código, cada pedazo que se lee del disco se envía automáticamente por la red al cliente, con control de flujo incluido: si la red es lenta, el lector espera; si el disco es

lento, la red espera. Cuando se lee el último pedazo, pipe cierra la respuesta con `end()` y el navegador da por terminada la descarga.

Línea 9: el servidor escucha en el puerto 3000. Cada cliente que pida el video obtiene su propia tubería independiente: diez clientes viendo el video al mismo tiempo son diez streams corriendo en paralelo, cada uno usando apenas unos kilobytes de memoria. Con la técnica de la "cubeta" (`fs.readFile` completo y luego `res.end` con todo), diez clientes de un video de 2 GB habrían requerido 20 GB de RAM. Con streams, el servidor ni se inmuta.

Hay un detalle de producción que debes conocer: en este ejemplo minimalista falta el manejo de errores. Si el archivo "grande.mp4" no existe, el stream emite "error" y, como nadie lo escucha, el proceso truena (la regla de oro otra vez). La versión robusta agrega el oyente:

```
const servidor2 = http.createServer((req, res) => {
  const video = fs.createReadStream("grande.mp4");
  video.on("error", (err) => {
    res.writeHead(404);
    res.end("Video no encontrado");
  });
  res.writeHead(200, { "Content-Type": "video/mp4" });
  video.pipe(res);
});
```

Si el archivo falta, el cliente recibe un 404 limpio en lugar de que tu servidor se muera. Nota un matiz interesante: aquí el `writeHead(200)` se manda antes de saber si el archivo existe, lo cual en producción se refinaría verificando primero; lo importante para esta unidad es el patrón: todo stream que crees, su oyente de "error". Sin excepciones.

Este ejemplo también responde a la pregunta "¿para qué sirven los streams en la vida real?": transmisión de video y audio (Netflix y YouTube usan esta idea a escala masiva), descargas de archivos grandes, subida de archivos (el req es un stream legible que puedes pipear a un archivo en disco), compresión al vuelo, procesamiento de logs gigantes línea por línea, y cualquier ETL (extraer, transformar, cargar) de datos. Donde veas "datos demasiado grandes para la memoria", verás streams.

Buffers a fondo: el idioma crudo de las computadoras

Tercera y última pieza del motor. Para entender los buffers hay que empezar por una verdad incómoda: las computadoras no entienden texto. No entienden letras, ni palabras, ni emojis. Lo único que una computadora entiende son números, y más específicamente, secuencias de bytes (grupos de 8 bits, valores de 0 a 255). Todo lo demás (el texto que lees, las fotos que

miras, la música que escuchas) es una convención: un acuerdo sobre cómo interpretar esos números. La letra "A" es el byte 65 solo porque todos acordamos que así sea (eso es ASCII/UTF-8). Un píxel rojo en una foto es una secuencia de bytes solo porque el formato JPEG dice cómo leerlos. Los datos binarios son eso: secuencias de bytes crudos, sin interpretar todavía. Y el Buffer de Node es la estructura que te permite trabajar con esos bytes directamente desde JavaScript.

¿Por qué JavaScript necesita esto? Porque JavaScript nació para texto: su tipo string maneja caracteres Unicode y es perfecto para palabras y código. Pero en el servidor te topas constantemente con cosas que no son texto: una imagen PNG, un video MP4, un archivo ZIP, datos cifrados, paquetes de red. Ninguno de esos es "texto" y tratarlos como strings los corrompe (un string intentaría interpretar los bytes como caracteres y los destrozaría). El Buffer existe para decir: "estos son bytes crudos, no los interpretes, guárdalos tal cual y déjame manipularlos". Es, como dice la clase 18 del curso, el idioma crudo de las computadoras, y Node te lo pone en las manos.

Crear un buffer es sencillo. La forma más común es Buffer.from, que convierte algo que ya tienes (un string, un array de números) en bytes:

```
const b1 = Buffer.from("Hola");  
console.log(b1);
```

En consola verás algo como <Buffer 48 6f 6c 61>. Esos son los bytes en hexadecimal: 48 es la "H", 6f es la "o", 6c es la "l", 61 es la "a". El string "Hola" se convirtió a su representación cruda según UTF-8. Nada se perdió: es la misma información, vista con otros ojos. Si imprimes b1.length, obtienes 4: cuatro bytes, uno por letra (con caracteres como la ñ o los acentos, un solo carácter puede ocupar 2 bytes en UTF-8, y eso es una fuente clásica de confusiones que veremos en errores comunes).

La otra forma fundamental es Buffer.alloc, que crea un buffer vacío de cierto tamaño, lleno de ceros:

```
const b2 = Buffer.alloc(8);  
console.log(b2);
```

Verás <Buffer 00 00 00 00 00 00 00 00>: ocho bytes en cero, listos para que los llenes. Esto se usa cuando sabes de antemano cuántos bytes necesitas (por ejemplo, al armar un paquete de red de tamaño fijo) o cuando vas a ir escribiendo bytes poco a poco. Existe también Buffer.allocUnsafe(tamaño), que es más rápido porque no limpia la memoria (puede traer basura de usos anteriores), pero su nombre te advierte: úsalo solo si vas a sobrescribir cada

byte de inmediato; de lo contrario puedes leer datos ajenos por accidente. Para este curso, quédate con `alloc` a secas: seguro y predecible.

Un buffer se puede leer y escribir byte por byte como si fuera un array, porque de hecho se comporta como uno:

```
const b3 = Buffer.from("Hola");
console.log(b3[0]);
b3[0] = 104;
console.log(b3.toString());
```

`b3[0]` te da 72 (el código de "H"). Al asignar `b3[0] = 104` (el código de "h" minúscula), el buffer cambia y al convertirlo a texto obtienes "hola". Cada posición es un byte (0 a 255); si intentas guardar un número mayor, se recorta. Esta manipulación directa es lo que hace a los buffers ideales para protocolos binarios: puedes armar byte por byte exactamente la secuencia que un dispositivo o un formato de archivo espera.

Ahora, la operación inversa: convertir bytes a algo legible. El método `toString` acepta un encoding, que es el "acuerdo de interpretación" que mencionamos. Los tres encodings que debes dominar son `utf8` (texto normal, el valor por defecto), `hex` (cada byte como dos dígitos hexadecimales, útil para depurar y para hashes) y `base64` (una representación en texto seguro para transmitir binarios por canales de texto, como JSON o HTML):

```
const b4 = Buffer.from("Hola mundo");

console.log(b4.toString("utf8"));
console.log(b4.toString("hex"));
console.log(b4.toString("base64"));
```

La primera línea devuelve "Hola mundo" (el texto original). La segunda devuelve "486f6c61206d756e646f" (los bytes en hexadecimal, pegados). La tercera devuelve "SG9sYSBtdW5kbw==" (la versión base64). Son tres formas de mirar los mismos bytes. El `base64` merece atención especial porque es el puente entre el mundo binario y el mundo del texto: si necesitas meter una imagen dentro de un JSON, o incrustarla en una página HTML, o mandarla en un correo, la conviertes a `base64` y viaja como texto sin corromperse. En los ejemplos paso a paso haremos exactamente eso con un archivo de imagen real.

Los buffers también se pueden cortar y pegar. El método `subarray` (o `slice` en versiones anteriores) extrae una porción, y `Buffer.concat` une varios buffers en uno solo:

```
const parte1 = Buffer.from("Hola ");
const parte2 = Buffer.from("mundo");
const completo = Buffer.concat([parte1, parte2]);
console.log(completo.toString());
```

Esto imprime "Hola mundo". El concat es protagonista en el mundo de los streams: como los datos llegan por pedazos (cada pedazo un buffer), el patrón clásico para reconstruir el todo es ir acumulando los pedazos en un array y al final hacer Buffer.concat. Lo verás en acción en los ejemplos:

```
const pedazos = [];
lector.on("data", (pedazo) => pedazos.push(pedazo));
lector.on("end", () => {
  const todo = Buffer.concat(pedazos);
  console.log("Bytes totales: " + todo.length);
});
```

Cada evento "data" entrega un buffer; los guardamos; al terminar, los unimos en uno solo. Este patrón (acumular buffers y concatenar al final) aparece en código real de Node más veces de las que imaginas: al descargar un archivo por HTTP, al leer el cuerpo de una petición, al procesar una imagen.

La pregunta honesta que debes hacerte: ¿por qué existen los buffers si ya hay strings? Tres razones. Primera: los strings de JavaScript son inmutables y están pensados para texto Unicode; manipular millones de bytes con strings es lento y peligroso (cada concatenación crea un string nuevo). Los buffers son mutables y están optimizados para bytes. Segunda: los strings no pueden representar datos que no son texto sin corromperlos; un byte con valor 0 o 255 no tiene "letra" equivalente y el string lo destrozaría. Tercera: el mundo exterior (discos, redes, otros programas) habla en bytes, no en strings Unicode; el buffer es la representación fiel de lo que realmente viaja por el cable o vive en el disco. El string es para humanos; el buffer es para máquinas. Y Node, que vive entre ambos mundos, necesita los dos.

Un detalle de rendimiento que te hará apreciar el diseño: los buffers viven fuera del heap normal de JavaScript (en memoria nativa), lo que los hace eficientes para grandes cantidades de datos binarios. No necesitas memorizar la mecánica interna, pero sí la consecuencia práctica: cuando proceses archivos grandes, trabaja con buffers y streams, no conviertas todo a strings "por comodidad", porque esa comodidad se paga en memoria y velocidad.

La relación entre los tres: un solo organismo

Es momento de juntar las piezas, porque por separado son herramientas y juntas son una arquitectura. La relación es esta: los streams mueven buffers, y los eventos avisan de todo lo que pasa en el camino. Cada vez que lees un archivo con `createReadStream`, lo que viaja por la tubería son buffers (pedazos de bytes crudos), y te enteras de su llegada gracias a los eventos "data" y "end" (y te enteras de los problemas gracias a "error"). El `EventEmitter` es el sistema nervioso, el stream es el sistema circulatorio, y el buffer es la sangre. Ninguna pieza tiene sentido completo sin las otras dos.

Mira cómo se entrelazan en un ejemplo que ya conoces, el de la petición POST manual:

```
req.on("data", (pedazo) => { ... });
```

Ese "pedazo" es un buffer. El evento "data" es el aviso. El req es el stream que lo transporta. Si el cuerpo es texto, lo conviertes con `pedazo.toString("utf8")`; si es un archivo binario subido por el usuario, lo mantienes como buffer y lo escribes a disco tal cual. El mismo código, el mismo patrón, dos naturalezas de datos distintas: la diferencia la marca cómo tratas el buffer.

Otro entrelazamiento: cuando haces pipe entre streams, por debajo hay eventos trabajando. El pipe se suscribe a "data" del origen, escribe en el destino, escucha "drain" para el control de flujo, escucha "end" para cerrar, y propaga errores. Tú ves una sola línea elegante; por dentro hay una coreografía de eventos moviendo buffers. Entender esa coreografía es lo que te permite depurar cuando algo falla: si un pipe se queda colgado, piensas en términos de eventos (¿llegó "end"? ¿se emitió "error"?) y de buffers (¿los pedazos están llegando? ¿de qué tamaño?).

Esta visión unificada también explica decisiones de diseño de Node que antes parecían arbitrarias. ¿Por qué los streams emiten "error" en lugar de lanzar excepciones? Porque son asíncronos: cuando el error ocurre, tu código ya siguió adelante hace rato; la única forma de avisarte es un evento. ¿Por qué "error" sin oyente mata el proceso? Porque un falló silencioso en un servidor es peor que una caída ruidosa: la caída te obliga a arreglarlo. ¿Por qué los datos llegan como buffers y no como strings? Porque Node no puede saber si lo que viaja es texto, una imagen o un video; te entrega los bytes crudos y tú decides cómo interpretarlos con el encoding adecuado. Cada rareza tiene su lógica cuando ves el organismo completo.

Errores comunes y sus soluciones

Cierro la explicación con los tropiezos clásicos, porque en este tema los errores no son detalles: son caídas de servidor en producción. Cada uno viene con su solución.

Error 1: no escuchar el evento "error" y que el proceso truene. Es el error número uno de los principiantes con streams y emisores. Creas un `createReadStream` de un archivo que no existe, o un servidor en un puerto ocupado, o una conexión que se cae a la mitad, y como nadie escucha "error", Node lanza la excepción y tu programa muere. La solución es la regla de oro: todo emisor que crees o uses lleva su oyente de "error", sin excepciones. En streams encadenados con pipe, escucha el error en cada eslabón de la cadena, porque un falló en el lector no lo captura el oyente del escritor. Si usas Node moderno, también existe `stream.pipeline`, que conecta streams y maneja los errores de todos con un solo callback; es la forma recomendada cuando la cadena crece.

Error 2: fugas de memoria por acumulación de oyentes. El caso típico: dentro del manejador de cada petición registras un oyente en un emisor global y nunca lo remueves. Con cada petición crece la lista; con miles de peticiones, el proceso se come la memoria y Node te avisa con el `MaxListenersExceededWarning` al pasar de 10. La solución tiene tres niveles: primero, pregúntate si el oyente debe vivir solo durante la petición; si es así, guárdalo en variable y remuévelo con `removeListener` u `off` cuando termine, o usa `once` si solo necesitas la primera ocurrencia. Segundo, si el oyente es permanente por diseño (un manejador global), regístralo una sola vez fuera del manejador de peticiones, no dentro. Tercero, usa `listenerCount` en tus pruebas para verificar que el número de oyentes no crece con el tráfico.

Error 3: confundir encodings y corromper datos. Los síntomas: texto con símbolos raros (el famoso "mojibake"), archivos binarios que se dañan al guardarlos, o un "Hola" que al concatenar pedazos se vuelve "Ho?la". Las causas típicas: convertir cada pedazo de un stream a string por separado cuando un carácter UTF-8 quedó partido entre dos pedazos (la ñ o un emoji ocupan varios bytes y pueden quedar a caballo entre dos buffers); o tratar datos binarios como strings "utf8" en algún paso intermedio. Las soluciones: cuando acumules texto de un stream, concatena los buffers primero con `Buffer.concat` y convierte a string hasta el final; o usa el modo objeto/encoding con cuidado sabiendo del riesgo de caracteres partidos (el módulo `string_decoder` existe justo para eso). Y la regla general: los datos binarios viajan como buffers de principio a fin; solo conviértelos a texto (o a base64) en los extremos, cuando realmente necesites texto.

Error 4: asumir que `emit` es asíncrono. Como los eventos "se sienten" asíncronos (llegan cuando algo pasa en el futuro), muchos creen que `emit` agenda los oyentes para después. No: `emit` ejecuta los oyentes de inmediato, en orden, antes de continuar. El problema aparece cuando un oyente hace algo pesado o lanza una excepción: como corre dentro de tu `emit`, te bloquea o te truena en la cara. La solución es doble: mantén los oyentes ligeros (si hay trabajo pesado, delega con `setImmediate` o a un worker), y si un oyente puede fallar, que capture sus propios errores o que el diseño use el evento "error" del emisor en lugar de lanzar.

Error 5: olvidar que pipe no propaga errores solo. Si encadenas `lector.pipe(compresor).pipe(escritor)` y el lector falla, el error se emite en el lector; si solo escuchas "error" en el escritor, jamás te enteras y el proceso puede morir o quedarse colgado en silencio. La solución: oyentes de "error" en cada stream de la cadena, o mejor, usa pipeline de stream/promises que centraliza el manejo.

Error 6: mezclar el modo fluido y el modo pausado de los streams legibles sin entenderlos. Un stream legible empieza en modo pausado: no emite "data" hasta que te suscribes a "data", llamas a `resume()` o haces pipe. En cuanto te suscribes a "data", entra en modo fluido y los pedazos llegan solos. El error típico es suscribirse a "data" y además llamar a `read()` manualmente, o pausar y reanudar sin control, perdiendo pedazos o duplicando lógica. La solución: elige un modo y quédate en él; para mover datos, pipe; para procesarlos tú, suscríbete a "data"/"end" y deja que fluya; el control manual con `pause/resume/read` solo cuando necesites ritmo personalizado.

Con estos seis errores cubiertos tienes el mapa de minas completo. La buena noticia: todos se evitan con tres hábitos: escucha siempre "error", no acumules oyentes sin removerlos, y trata los bytes como bytes hasta el último momento. Si interiorizas esos hábitos, programarás con eventos, streams y buffers con la tranquilidad de quien conoce el terreno.

Ejemplos paso a paso

Ejemplo 1: sistema de notificaciones con EventEmitter

Vamos a construir un centro de notificaciones como el de una aplicación real: distintos módulos (correo, bitácora, interfaz) reaccionan a lo que pasa en el sistema, sin conocerse entre sí. Paso 1: crea el archivo `notificaciones.js` e importa `EventEmitter`. Paso 2: crea una instancia que será el centro de avisos. Paso 3: registra los oyentes, uno por cada canal de notificación. Paso 4: emite eventos simulando lo que pasa en la app. Paso 5: ejecútalo y observa cómo un solo `emit` despierta a varios oyentes.

```

const EventEmitter = require("events");

const avisos = new EventEmitter();

avisos.on("usuario-nuevo", (usuario) => {
  console.log("[CORREO] Bienvenida enviada a " + usuario.email);
});

avisos.on("usuario-nuevo", (usuario) => {
  console.log("[BITÁCORA] Registro creado para " + usuario.nombre);
});

avisos.on("compra", (datos) => {
  console.log("[CORREO] Recibo enviado por $" + datos.total);
  console.log("[BITÁCORA] Venta #" + datos.id + " guardada");
});

avisos.on("error", (err) => {
  console.log("[ALERTA] Algo falló: " + err.message);
});

avisos.emit("usuario-nuevo", { nombre: "Ana", email: "ana@ejemplo.com" });
avisos.emit("compra", { id: 501, total: 1200 });

```

Al ejecutarlo verás cuatro líneas: dos por el usuario nuevo (correo y bitácora) y dos por la compra. Fíjate en el desacoplamiento: el código que emite "compra" no sabe que existen el correo y la bitácora; si mañana agregas notificaciones por SMS, solo agregas otro on. Ese es el poder del patrón en una aplicación real: el núcleo emite lo que pasa y los módulos se suscriben a lo que les toca. Como ejercicio mental, imagina este mismo sistema sin eventos: una función gigante con diez llamadas en secuencia, donde agregar un canal nuevo significa modificar el núcleo. Con eventos, el núcleo jamás se toca.

Ejemplo 2: clase Tienda que emite eventos de inventario

Ahora subamos de nivel con una clase propia que hereda de EventEmitter. La Tienda vende productos y avisa cuando el inventario baja o cuando algo se agota, para que otros módulos (reabastecimiento, alertas al gerente) reaccionen. Paso 1: define la clase extendiendo EventEmitter. Paso 2: el constructor recibe el inventario inicial. Paso 3: el método vender descuenta existencias y emite eventos según el caso. Paso 4: fuera de la clase, suscribe los oyentes. Paso 5: simula ventas y observa los avisos.

```

const EventEmitter = require("events");

```

```

class Tienda extends EventEmitter {
  constructor() {
    super();
    this.inventario = { laptop: 5, mouse: 12, teclado: 3 };
  }

  vender(producto, cantidad) {
    if (!this.inventario[producto]) {
      this.emit("error", new Error("Producto desconocido: " + producto));
      return;
    }
    if (this.inventario[producto] < cantidad) {
      this.emit("error", new Error("No hay suficientes " + producto));
      return;
    }
    this.inventario[producto] -= cantidad;
    this.emit("venta", { producto, cantidad, quedan: this.inventario[producto] });

    if (this.inventario[producto] === 0) {
      this.emit("agotado", producto);
    } else if (this.inventario[producto] <= 3) {
      this.emit("poco-inventario", producto, this.inventario[producto]);
    }
  }
}

const tienda = new Tienda();

tienda.on("venta", (d) => {
  console.log("Venta: " + d.cantidad + " x " + d.producto + " (quedan " + d.quedan +
  ")");
});
tienda.on("poco-inventario", (p, q) => {
  console.log("AVISO: quedan solo " + q + " " + p + ". Hay que resurtir.");
});
tienda.on("agotado", (p) => {
  console.log("ALERTA: " + p + " se agotó por completo.");
});
tienda.on("error", (err) => {
  console.log("Error de venta: " + err.message);
});

tienda.vender("teclado", 1);
tienda.vender("teclado", 2);

```

```
tienda.vender("laptop", 10);
```

La primera venta deja 2 teclados y dispara "poco-inventario". La segunda los agota y dispara "agotado". La tercera intenta vender 10 laptops habiendo 5 y dispara "error", capturado con elegancia en lugar de tronar el proceso. Este ejemplo muestra el patrón completo de una clase emisora: valida, actúa, y comunica cada situación con el evento adecuado. En una aplicación real, el oyente de "poco-inventario" podría crear una orden de compra automática y el de "agotado" mandar un mensaje al gerente, todo sin que la clase Tienda sepa nada de correos ni de proveedores.

Ejemplo 3: servidor que transmite video con streams

El ejemplo estrella, listo para ejecutar. Necesitas un archivo de video (cualquier mp4 sirve; si no tienes uno a la mano, un archivo grande de texto también demuestra el punto). Paso 1: crea servidor-video.js con el código robusto que incluye manejo de errores. Paso 2: coloca un archivo "grande.mp4" en la misma carpeta (o cambia el nombre en el código). Paso 3: ejecuta con node servidor-video.js. Paso 4: abre <http://localhost:3000> en tu navegador y observa cómo el video empieza a reproducirse antes de descargarse por completo: eso es el streaming en acción.


```

const http = require("http");
const fs = require("fs");

const servidor = http.createServer((req, res) => {
  const video = fs.createReadStream("grande.mp4");

  video.on("error", (err) => {
    console.log("Error con el video: " + err.message);
    if (!res.headersSent) {
      res.writeHead(404, { "Content-Type": "text/plain" });
    }
    res.end("Video no encontrado");
  });

  res.writeHead(200, { "Content-Type": "video/mp4" });
  video.pipe(res);
});

servidor.on("error", (err) => {
  console.log("Error del servidor: " + err.message);
});

servidor.listen(3000, () => {
  console.log("Servidor de video en http://localhost:3000");
});

```

Prueba dos experimentos para sentir la diferencia. Experimento A: pide el video desde dos pestañas del navegador al mismo tiempo; ambas reproducen sin que el servidor se inmute, porque cada una tiene su propia tubería de 64 KB por pedazo. Experimento B (mental o con un archivo muy grande): compara con la versión "cubeta", que sería leer todo con `fs.readFile` y luego `res.end(datos)`; con un video de 4 GB esa versión intentaría meter 4 GB a memoria por cliente. El streaming mantiene la memoria plana sin importar el tamaño. Si quieres ver los pedazos con tus propios ojos, agrega un oyente de "data" al stream antes del pipe que cuente los pedazos: verás decenas o cientos de eventos "data" pasar, cada uno un buffer, cada uno viajando del disco a la red sin acumularse.

Ejemplo 4: convertir una imagen a base64 con buffers

Último ejemplo: tomar una imagen real (binario puro) y convertirla a base64 para incrustarla en HTML o mandarla en un JSON. Necesitas cualquier imagen pequeña (un PNG o JPG). Paso 1: crea `imagen-base64.js`. Paso 2: lee la imagen como buffer con `fs.readFile` (sin encoding, para recibir bytes crudos). Paso 3: convierte el buffer a base64 con `toString("base64")`. Paso 4:

arma un data URL y guárdalo en un HTML de prueba. Paso 5: abre el HTML y comprueba que la imagen se ve sin archivo externo.

```
const fs = require("fs");

fs.readFile("foto.png", (err, datos) => {
  if (err) {
    console.log("No se pudo leer la imagen: " + err.message);
    return;
  }

  console.log("Tipo de datos: " + (Buffer.isBuffer(datos) ? "Buffer" : typeof
datos));
  console.log("Tamaño en bytes: " + datos.length);
  console.log("Primeros bytes (hex): " + datos.subarray(0, 8).toString("hex"));

  const base64 = datos.toString("base64");
  const dataUrl = "data:image/png;base64," + base64;

  const html = '<html><body><h1>Imagen incrustada</h1></body></html>';
  fs.writeFileSync("imagen.html", html);
  console.log("HTML generado. Ábrelo en tu navegador.");
});
```

Tres observaciones importantes. Primera: `fs.readFile` sin `encoding` te entrega un `Buffer`, no un `string`; la comprobación con `Buffer.isBuffer` lo confirma. Segunda: los primeros bytes en hexadecimal de un PNG siempre empiezan con `89504e47`, la "firma" del formato; estás viendo el idioma crudo del archivo con tus propios ojos, sin interpretación. Tercera: el data URL permite que la imagen viaje como texto dentro del HTML; es la técnica que usan los correos con imágenes incrustadas y muchas APIs que devuelven archivos en JSON. Si la imagen fuera grande, el base64 crecería un 33 por ciento (el costo conocido de esta codificación), por eso para archivos pesados se prefieren los streams; pero para imágenes pequeñas y medianas, este puente binario-a-texto es la herramienta correcta.

Práctica

Ejercicio 1: semáforo con eventos

Enunciado: crea una clase Semaforo que herede de EventEmitter y tenga un método iniciar(). Al iniciarse, debe emitir el evento "luz" con el color actual, cambiando de "verde" a "amarillo" a "rojo" y repitiendo el ciclo, cada 2 segundos. Además debe emitir "error" si alguien intenta iniciarlo dos veces. Suscribe oyentes que impriman cada cambio de color.

Solución paso a paso. Paso 1: la clase extiende EventEmitter y en el constructor llama a super() e inicializa el índice del color y una bandera de encendido. Paso 2: el método iniciar verifica la bandera; si ya está encendido, emite "error" y sale. Paso 3: si no, marca la bandera, define el arreglo de colores y arranca un setInterval que emite "luz" con el color actual y avanza el índice en forma circular. Paso 4: fuera, crea la instancia, suscribe los oyentes de "luz" y "error", llama a iniciar dos veces para probar el error, y deja correr unos segundos.

```
const EventEmitter = require("events");

class Semaforo extends EventEmitter {
  constructor() {
    super();
    this.colores = ["verde", "amarillo", "rojo"];
    this.indice = 0;
    this.encendido = false;
  }

  iniciar() {
    if (this.encendido) {
      this.emit("error", new Error("El semáforo ya está en marcha"));
      return;
    }
    this.encendido = true;
    setInterval(() => {
      const color = this.colores[this.indice];
      this.emit("luz", color);
      this.indice = (this.indice + 1) % this.colores.length;
    }, 2000);
  }
}

const s = new Semaforo();
s.on("luz", (color) => console.log("Luz: " + color));
s.on("error", (err) => console.log("Error: " + err.message));
s.iniciar();
s.iniciar();
```

El truco del índice circular (índice + 1) % 3 hace que después del rojo vuelva al verde sin condicionales. La segunda llamada a iniciar demuestra el evento "error" funcionando como cinturón de seguridad. Este ejercicio consolida herencia de EventEmitter, emisión periódica con setInterval y el manejo del evento especial "error".

Ejercicio 2: once contra on

Enunciado: escribe un programa que demuestre la diferencia entre on y once. Registra dos oyentes para el evento "puerta": uno con on y otro con once. Dispara el evento tres veces con distintos nombres de visitante. Explica en comentarios qué se imprime y por qué.

Solución. El oyente registrado con on reacciona a los tres disparos; el registrado con once solo al primero, porque Node lo desuscribe automáticamente después de ejecutarlo una vez.

```
const EventEmitter = require("events");
const casa = new EventEmitter();

casa.on("puerta", (visitante) => {
  console.log("[on] Tocarón la puerta: " + visitante);
});

casa.once("puerta", (visitante) => {
  console.log("[once] Primera visita del día: " + visitante);
});

casa.emit("puerta", "Ana");
casa.emit("puerta", "Luis");
casa.emit("puerta", "María");
```

Salida esperada: en el primer emit aparecen las dos líneas; en el segundo y tercero solo la de [on]. El oyente once es ideal para eventos de una sola vez en la vida del programa (arranque del servidor, primera conexión). Moraleja práctica: si un manejador solo tiene sentido la primera vez, usa once y olvídate de banderas manuales de "ya se hizo".

Ejercicio 3: copiar y comprimir un archivo con streams encadenados

Enunciado: toma un archivo de texto grande (puedes generarlo repitiendo líneas) y escribe un programa que lo lea con un stream, lo comprima con gzip usando un stream de transformación, y lo escriba comprimido en disco, todo con pipe encadenado y manejo de errores en cada eslabón.

Solución paso a paso. Paso 1: genera el archivo de prueba. Paso 2: crea el lector con `createReadStream`. Paso 3: crea el transformador con `zlib.createGzip()`. Paso 4: crea el escritor con `createWriteStream`. Paso 5: encadena con `pipe`. Paso 6: agrega oyente de "error" a cada uno de los tres streams y un oyente de "finish" al escritor para confirmar.

```
const fs = require("fs");
const zlib = require("zlib");

fs.writeFileSync("grande.txt", "Línea de prueba para comprimir.\n".repeat(100000));

const lector = fs.createReadStream("grande.txt");
const compresor = zlib.createGzip();
const escritor = fs.createWriteStream("grande.txt.gz");

lector.on("error", (e) => console.log("Error leyendo: " + e.message));
compresor.on("error", (e) => console.log("Error comprimiendo: " + e.message));
escritor.on("error", (e) => console.log("Error escribiendo: " + e.message));
escritor.on("finish", () => console.log("Compresión terminada.));

lector.pipe(compresor).pipe(escritor);
```

Verifica el resultado comparando tamaños: el `.gz` debe ser mucho más pequeño que el `.txt`. Este ejercicio demuestra el encadenamiento de tres eslabones (legible -> transformación -> escribible), el manejo de errores por eslabón (el punto que `pipe` no resuelve solo) y la magia de procesar megabytes con memoria constante. Como reto extra, agrega un segundo `pipe` que descomprima el `.gz` de vuelta a texto y comprueba que el contenido es idéntico al original.

Ejercicio 4: detective de buffers

Enunciado: dado el buffer de la palabra "Programación", escribe un programa que: a) imprima cuántos bytes ocupa y explique por qué no son 12; b) imprima su representación en hexadecimal; c) lo parta en dos buffers con `subarray` y los vuelva a unir con `Buffer.concat`, comprobando que el resultado es idéntico al original; d) convierta el resultado a `base64`.

Solución. La palabra "Programación" tiene 12 caracteres pero la `ó` ocupa 2 bytes en UTF-8, así que el buffer mide 13 bytes. El resto es aplicar los métodos vistos en la unidad.

```
const original = Buffer.from("Programación", "utf8");

console.log("Bytes: " + original.length);
console.log("La ó ocupa 2 bytes en UTF-8, por eso son 13 y no 12.");
console.log("Hex: " + original.toString("hex"));

const mitad = Math.floor(original.length / 2);
const p1 = original.subarray(0, mitad);
const p2 = original.subarray(mitad);
const reunido = Buffer.concat([p1, p2]);

console.log("Reconstruido igual: " + reunido.equals(original));
console.log("Texto: " + reunido.toString("utf8"));
console.log("Base64: " + reunido.toString("base64"));
```

El método `equals` compara dos buffers byte por byte y es la forma correcta de verificar igualdad (comparar con `===` compararía referencias, no contenido). Nota el peligro didáctico: partimos el buffer justo a la mitad de los bytes, y si el corte cae en medio de la `ó` (que ocupa 2 bytes), cada mitad por separado mostraría un carácter roto al convertirla a string; al reunir las con `concat`, el carácter se recompone perfecto. Esa es exactamente la trampa de los caracteres partidos entre pedazos de stream de la que hablamos en errores comunes, vista aquí en miniatura.

Ejercicio 5: servidor de archivos con rango de streams y conteo de pedazos

Enunciado: escribe un servidor HTTP que sirva cualquier archivo pedido por su nombre (por ejemplo `http://localhost:3000/archivo?nombre=datos.txt`) usando `createReadStream` y `pipe`. Debe: contar cuántos pedazos ("data") viajan por cada descarga e imprimirlo al terminar; responder 404 si el archivo no existe (con oyente de "error"); y nunca cargar el archivo completo en memoria.

Solución paso a paso. Paso 1: crea el servidor y extrae el nombre del archivo de la URL. Paso 2: crea el stream de lectura. Paso 3: registra el oyente de "error" que responde 404 (cuidando no escribir cabeceras dos veces). Paso 4: cuenta los eventos "data" (esto pone el stream en modo fluido, lo cual es compatible con `pipe`). Paso 5: en "end", imprime el total de pedazos. Paso 6: conecta con `pipe` y arranca el servidor con su propio oyente de "error".

```

const http = require("http");
const fs = require("fs");
const url = require("url");

const servidor = http.createServer((req, res) => {
  const nombre = url.parse(req.url, true).query.nombre;

  if (!nombre) {
    res.writeHead(400);
    res.end("Falta el parámetro nombre");
    return;
  }

  const lector = fs.createReadStream(nombre);
  let pedazos = 0;

  lector.on("error", (err) => {
    console.log("Error: " + err.message);
    if (!res.headersSent) {
      res.writeHead(404, { "Content-Type": "text/plain" });
    }
    res.end("Archivo no encontrado");
  });

  lector.on("data", () => { pedazos++; });
  lector.on("end", () => {
    console.log("Descarga de " + nombre + " terminada en " + pedazos + "
pedazos.");
  });

  res.writeHead(200, { "Content-Type": "application/octet-stream" });
  lector.pipe(res);
});

servidor.on("error", (err) => console.log("Error del servidor: " + err.message));

servidor.listen(3000, () => console.log("Servidor en http://localhost:3000"));

```

Pruébalo con un archivo pequeño (verás 1 pedazo) y con uno grande (verás muchos). Este ejercicio integra las tres piezas de la unidad en un programa real: el servidor y el lector son emisores de eventos, los datos viajan como buffers por el stream, y el pipe los mueve con memoria constante. La guarda de headersSent evita el error clásico de intentar responder dos veces cuando el archivo no existe. Como reto extra, agrega la cabecera Content-Disposition con el nombre del archivo para que el navegador lo descargue en lugar de mostrarlo.

Resumen de la unidad

Llegamos al final de la unidad del corazón asíncrono. Esto es lo esencial que debes llevarte grabado.

El patrón emisor oyente es la base de todo Node.js: un objeto emite avisos con nombres (eventos) y las funciones suscritas reaccionan, sin que el emisor conozca a sus oyentes. Es como la radio: la estación transmite una vez y cada radio sintonizado escucha a su manera. Este patrón invierte la responsabilidad del polling ("preguntar a cada rato") al aviso ("suscríbete y te aviso"), y es lo que permite a Node atender miles de operaciones sin bloquearse.

EventEmitter, del módulo events, es la implementación del patrón: `new EventEmitter()` crea un emisor; `on` registra oyentes permanentes y `once` oyentes de una sola vez; `emit` dispara el evento y ejecuta los oyentes en orden, de inmediato y con los argumentos que le pases; `removeListener` u `off` quitan un oyente (guarda la referencia en variable) y `removeAllListeners` limpia por evento o por completo. Tus propias clases pueden heredar de EventEmitter para que tus objetos "hablen" con eventos. Y la regla de oro: el evento "error" siempre debe tener oyente, porque un "error" sin oyente mata el proceso. Los servidores, las peticiones, las respuestas, los sockets y los procesos hijos de Node ya son emisores; cuando usas sus eventos, estás hablando el idioma materno de la plataforma.

Los streams son tuberías para datos que llegan poco a poco: en lugar de cargar todo en memoria (la cubeta), los datos fluyen por pedazos (la tubería) y la memoria se mantiene constante sin importar el tamaño. Hay cuatro tipos: legibles (fuentes, como `createReadStream`), escribibles (destinos, como la respuesta `res`), dúplex (ambas direcciones, como un socket) y de transformación (dúplex que modifica los datos, como `gzip`). El método `pipe` enchufa un legible a un escribible y mueve los datos con control de flujo automático; los pipes se encadenan para armar líneas de procesamiento. Los eventos "data", "end" y "error" te permiten consumir un stream a mano cuando necesitas control fino. El ejemplo cumbre, `fs.createReadStream("grande.mp4").pipe(res)`, sirve video a muchos clientes simultáneos usando apenas kilobytes de memoria por conexión.

Los buffers son secuencias de bytes crudos: el idioma de las máquinas. JavaScript los necesita porque los strings son para texto y se corrompen con datos binarios como imágenes, video o paquetes de red. `Buffer.from` convierte strings o arreglos en bytes, `Buffer.alloc` crea espacio vacío y seguro, y `toString` con `utf8`, `hex` o `base64` interpreta esos bytes de distintas formas. `Buffer.concat` une pedazos y `subarray` los corta; `equals` los compara. Regla práctica: los binarios viajan como buffers de principio a fin y solo se convierten a texto en los extremos.

Las tres piezas forman un solo organismo: los streams mueven buffers y los eventos avisan de todo lo que pasa en el camino. Cada "data" que recibes es un evento entregándote un buffer que viajó por un stream. Y los tres hábitos que te salvan en producción: escucha siempre "error", no acumules oyentes sin removerlos, y no conviertas bytes a texto antes de tiempo.

En la siguiente unidad seguiremos construyendo sobre esta base: con el corazón asíncrono latiendo, estás listo para temas que lo usan intensivamente. Por ahora, práctica: crea emisores, encadena streams, toca buffers. La teoría de esta unidad se vuelve intuición solo cuando la ejecutas.

UNIDAD 7: Dominando la asincronía: callbacks, promesas y async/await

Hay un momento en el aprendizaje de Node.js que separa a quienes solo copian código de quienes realmente entienden la herramienta. Ese momento llega cuando tu programa necesita hacer algo lento: leer un archivo del disco, pedir datos a una API por internet, consultar una base de datos. De pronto tu código, que hasta ahora se ejecutaba línea por línea en orden perfecto, empieza a comportarse de forma extraña. Lees un archivo y justo en la línea siguiente intentas usar su contenido... y está vacío. Pides datos a un servidor y tu programa sigue corriendo como si nada antes de que la respuesta llegue. No es un error de sintaxis ni un bug de Node: es que acabas de chocar con la naturaleza asíncrona de JavaScript, y esta unidad existe para convertir ese choque en tu mayor fortaleza.

La asincronía no es un tema más del curso. Es EL tema de Node.js. Todo el ecosistema, desde el servidor web más simple hasta las aplicaciones más grandes del mundo, está construido sobre la idea de que las operaciones lentas no deben detener tu programa. Si vienes de lenguajes donde cada línea espera pacientemente a la anterior, Node te va a exigir un cambio de mentalidad. Pero es un cambio que vale oro, porque cuando lo dominas entiendes por qué Node puede atender miles de usuarios al mismo tiempo con recursos modestos, por qué las empresas lo eligen para servicios en tiempo real, y por qué tu código puede hacer mucho más con mucho menos.

Piensa en un restaurante. En una cocina tradicional, el cocinero pone un taco al fuego y se queda parado mirándolo hasta que se cocina, luego pone el siguiente y vuelve a esperar. Eso es el modelo síncrono: una cosa a la vez, esperando cada paso. Ahora piensa en un buen taquero: pone varios tacos en el comal, mientras se cocinan corta cebolla y cilantro, sirve una salsa, cobra a un cliente, voltea los tacos, prepara el siguiente pedido. No espera a que cada taco esté listo para seguir trabajando. El taquero es Node.js: un solo trabajador que hace muchas cosas a la vez porque nunca se queda bloqueado esperando. Esta unidad te enseña a pensar como ese taquero.

Aquí vas a aprender los tres estilos históricos para manejar la asincronía en Node, porque los tres siguen vivos en el mundo real: los callbacks, que fueron la forma original y todavía aparecen en muchísimo código y librerías; las promesas, que ordenaron el caos de los callbacks y se convirtieron en el estándar moderno; y `async/await`, la sintaxis que hace que el código asíncrono se lea de arriba a abajo como si fuera síncrono, sin perder ninguna de sus ventajas. Además vas a entender a fondo el event loop, el motor que hace posible toda esta magia: cómo Node, con un solo hilo, logra atender miles de operaciones simultáneas sin despeinarse.

No te voy a mentir: este tema confunde a todo el mundo al principio. Es normal que al leer los primeros ejemplos sientas que el orden de ejecución no tiene sentido. La buena noticia es que esa confusión tiene cura, y la cura es práctica con ejemplos reales. Al final de esta unidad vas a leer tres archivos encadenados de tres formas distintas, vas a manejar errores en cada estilo, vas a paralelizar operaciones con `Promise.all`, y vas a escribir código `async/await` que se lee como una receta de cocina paso a paso. Y lo más importante: vas a entender POR QUÉ funciona así, no solo CÓMO se escribe.

Un aviso importante antes de empezar. Esta unidad asume que ya dominas las unidades anteriores: módulos, el sistema de archivos (fs), y cómo crear scripts que corran con node. Si algo de eso te suena borroso, vale la pena repasar la Unidad 4 (módulos) y la Unidad 5 (sistema de archivos) antes de continuar, porque aquí vamos a usar fs constantemente como nuestro laboratorio de asincronía. El disco duro es el ejemplo perfecto de operación lenta: leer un archivo toma tiempo real, y es exactamente el tipo de espera que Node aprendió a manejar sin bloquearse.

Prepara tu editor, abre tu terminal, y olvídate por un momento de la idea de que el código siempre corre en orden. Bienvenido a la asincronía: el corazón de Node.js.

Explicación detallada

Qué es la asincronía y por qué Node la necesita

Empecemos con lo básico: en programación, una operación síncrona es aquella que bloquea. Cuando tu programa ejecuta una operación síncrona, se detiene por completo hasta que esa operación termina. Es como llamar por teléfono a alguien y quedarte en la línea, sin hacer nada más, hasta que te contesten. Una operación asíncrona, en cambio, es como mandar un mensaje de texto: lo envías y sigues con tu vida; cuando te responden, te avisan y atiendes la respuesta.

En la mayoría de los programas que has escrito hasta ahora, todo era síncrono. Declarabas una variable, hacías una operación, imprimías el resultado, todo en orden, todo inmediato. Eso funciona porque las operaciones en memoria son rapidísimas: sumar dos números, recorrer un arreglo de cien elementos, concatenar textos. La computadora hace eso en microsegundos y ni te enteras de la espera.

Pero hay operaciones que NO son rápidas. Leer un archivo del disco duro puede tomar milisegundos. Pedir datos a un servidor al otro lado del mundo puede tomar cientos de milisegundos o incluso segundos. Consultar una base de datos con millones de registros puede tomar un rato. En el mundo de la computación, un milisegundo es una eternidad: tu procesador puede ejecutar millones de instrucciones en ese tiempo. Si tu programa se queda bloqueado esperando cada operación lenta, está desperdiciando una cantidad obscena de capacidad.

Aquí es donde entra la filosofía de Node.js. Node nació con una apuesta radical: en lugar de crear un hilo nuevo por cada tarea (como hacen otros servidores, gastando memoria y recursos en cada conexión), usar UN SOLO HILO que nunca se bloquee. Cuando llega una operación lenta, Node la delega, sigue atendiendo otras cosas, y cuando la operación termina, Node retoma el resultado. Es el taquero: un solo trabajador, cero tiempo muerto.

Para que esto funcione, Node necesita que TÚ escribas tu código de forma asíncrona. Necesita que le digas: "oye, lee este archivo, y CUANDO TERMINES avísame con esta función". Esa función que le pasas para que te avise se llama callback, y es la primera pieza del rompecabezas que vamos a armar.

Mira este ejemplo para que sientas la diferencia en carne propia:

```
const fs = require("fs");

console.log("1. Antes de leer el archivo");

fs.readFile("nota.txt", "utf8", function(error, contenido) {
  if (error) {
    console.log("Hubo un error:", error.message);
    return;
  }
  console.log("3. Contenido del archivo:", contenido);
});

console.log("2. Después de pedir la lectura");
```

Si corres este código, la salida NO es 1, 3, 2. Es 1, 2, 3. ¿Por qué? Porque `fs.readFile` es asíncrono: Node le pide al sistema operativo que lea el archivo, registra tu función callback como "avísame cuando termines", y SIGUE ADELANTE con la línea siguiente sin esperar. Cuando el sistema operativo termina de leer el disco, Node ejecuta tu callback con el contenido. El "2. Después de pedir la lectura" se imprime antes que el contenido porque pedir la lectura no es lo mismo que terminarla.

Este es el momento "ajá" más importante de la unidad. Grábatelo: en Node, pedir una operación lenta NO te entrega el resultado; te entrega una promesa de avisarte cuando esté listo. Tu código sigue corriendo. El resultado llega después, a través del mecanismo que elijas: callback, promesa o `async/await`.

La versión síncrona del mismo código sería `fs.readFileSync`, que SÍ bloquea: tu programa se congela hasta que el archivo se lee por completo. Funciona, y para scripts pequeños está bien, pero en un servidor que atiende a cien usuarios a la vez, un `readFileSync` es un crimen: mientras un usuario espera su archivo, los otros 99 se quedan congelados mirando una pantalla en blanco. Por eso en Node la regla de oro es: nunca bloquee el hilo principal con operaciones lentas.

El event loop: el corazón que nunca duerme

Ahora la pregunta del millón: si Node tiene un solo hilo, ¿cómo demonios hace muchas cosas a la vez? La respuesta es el event loop, el ciclo de eventos, y entenderlo es entender a Node.js de verdad.

Imagina la fila de un banco con UN SOLO cajero, pero un cajero extremadamente eficiente y con un sistema inteligente. Llega un cliente y pide un trámite que tarda 20 minutos en el sistema central. Un cajero normal lo atendería y la fila se detendría 20 minutos. Pero este cajero hace algo distinto: registra el trámite, le dice al cliente "siéntese, le aviso cuando esté listo", y llama al siguiente de la fila. Atiende a otro, registra otro trámite lento, sigue con el siguiente. Cuando el sistema central termina uno de los trámites pendientes, suena una campanita, el cajero retoma ese caso un momento, le entrega el resultado al cliente, y sigue con la fila. Un solo cajero, decenas de clientes atendidos "al mismo tiempo", cero tiempo muerto.

Ese cajero es el event loop. El "sistema central" lento son las operaciones de entrada/salida: disco, red, base de datos. Y la "campanita" es el mecanismo por el cual Node se entera de que una operación terminó y ejecuta el callback correspondiente.

Técnicamente, el event loop es un ciclo infinito que corre en el hilo principal de Node y que se la pasa preguntando: "¿hay algo pendiente? ¿terminó alguna operación? ¿hay callbacks

esperando su turno?". Cuando una operación asíncrona termina (el sistema operativo leyó el archivo, llegó la respuesta de la red), su callback se pone en una cola, y el event loop lo ejecuta en cuanto el hilo principal está libre. Por eso el orden de tu código no es el orden de ejecución: el event loop decide cuándo corre cada callback según cuándo terminen las operaciones reales.

Vamos a desmenuzar las fases del event loop, sin clavarnos en detalles excesivos pero con la precisión suficiente para que entiendas qué pasa bajo el cofre:

1. Timers: aquí se ejecutan los callbacks de `setTimeout` y `setInterval` cuyo tiempo ya se cumplió. Si pusiste `setTimeout(fn, 1000)`, tu función espera en esta fase hasta que pase un segundo.
2. Callbacks pendientes de I/O: callbacks de operaciones de entrada/salida que se retrasaron, como algunos errores de red o de sistema.
3. Poll (espera y recolección): esta es la fase más importante. Aquí el event loop espera a que lleguen eventos nuevos (datos de un socket, un archivo que terminó de leerse) y ejecuta sus callbacks. Si no hay nada que hacer, puede quedarse aquí esperando, como el taquero atento a que un taco esté listo para voltear.
4. Check: aquí corren los callbacks de `setImmediate`, que son como decir "haz esto justo después de terminar la fase de poll".
5. Close: callbacks de eventos de cierre, como cuando se cierra una conexión.

Y entre cada fase hay colas de microtarearías (donde viven las promesas) que tienen prioridad: antes de pasar a la siguiente fase, Node vacía por completo la cola de microtarearías. Esto explica un detalle que confunde a muchos: los `.then` de las promesas SIEMPRE se ejecutan antes que los `setTimeout`, aunque el timeout sea de cero milisegundos. Las promesas son microtarearías con pase VIP; los timers esperan su fase.

Volvamos al taquero, porque la analogía aguanta más. El taquero (event loop) tiene un comal (el hilo principal) donde solo cabe una acción a la vez: voltear un taco, cortar cebolla, cobrar. Pero los tacos se cocinan SOLOS en el comal mientras él hace otras cosas (las operaciones de I/O corren en segundo plano, manejadas por el sistema operativo o por un grupo de hilos trabajadores que Node usa internamente para ciertas tareas). Cuando un taco está listo, el taquero lo nota y lo atiende. Nunca se queda parado mirando un taco crudo esperando a que se dore: eso sería bloquear el hilo, y un taquero bloqueado es un taquero que pierde clientes.

Hay un detalle técnico que vale la pena mencionar: aunque decimos "un solo hilo", Node por dentro sí usa un pequeño grupo de hilos trabajadores (el thread pool de libuv) para ciertas operaciones pesadas cómo leer archivos o comprimir datos. Pero TÚ, como programador de

JavaScript, solo ves un hilo: tu código JavaScript nunca corre en paralelo consigo mismo. Esto es una bendición disfrazada: no tienes que pelear con los problemas clásicos de la concurrencia (dos hilos modificando la misma variable al mismo tiempo), porque en tu código solo una cosa pasa a la vez. La "simultaneidad" de Node es una ilusión muy bien orquestada: muchas operaciones en vuelo, pero tu JavaScript siempre ejecuta una instrucción a la vez.

¿Qué pasa si bloqueas el event loop? Imagina que en tu servidor pones un ciclo for que cuenta hasta diez mil millones. Ese ciclo corre en el hilo principal, y mientras corre, el event loop no puede atender a nadie: ni timers, ni lecturas de archivo, ni peticiones de usuarios. Todo se congela. Por eso la regla de oro: el hilo principal es para coordinación rápida, nunca para trabajo pesado y nunca para esperas bloqueantes. Si tienes un cálculo muy pesado, hay herramientas para eso (worker threads), pero eso es tema de otra unidad.

Un experimento mental para cerrar esta sección: ¿en qué orden se imprime esto?

```
console.log("A");

setTimeout(function() {
  console.log("B");
}, 0);

Promise.resolve().then(function() {
  console.log("C");
});

console.log("D");
```

La respuesta es A, D, C, B. A y D son código síncrono: corren de inmediato, en orden. Luego el hilo principal se libera y el event loop atiende primero las microtarefas: la promesa (C) tiene pase VIP. Al final, en la fase de timers, corre el setTimeout (B), aunque su espera fuera de cero milisegundos. Si entendiste por qué el orden es A-D-C-B, ya entiendes el event loop mejor que la mayoría de los programadores que usan Node todos los días.

Callbacks: la forma original de decir "avísame cuando termines"

Un callback es simplemente una función que le pasas a otra función para que la ejecute después, cuando termine su trabajo. En Node, los callbacks son el mecanismo más antiguo para manejar la asincronía, y aunque hoy tenemos herramientas más modernas, los callbacks siguen apareciendo por todas partes: en el módulo fs clásico, en muchas librerías, en los manejadores de eventos. Entenderlos bien no es opcional, es requisito para leer código real.

La convención más importante de Node es el callback "error-first" (el error primero). Toda función asíncrona clásica de Node recibe un callback cuyo PRIMER parámetro es el error (o null si todo salió bien) y los siguientes parámetros son los resultados. Míralo en acción:

```
const fs = require("fs");

fs.readFile("nota.txt", "utf8", function(error, datos) {
  if (error) {
    console.error("Algo salió mal:", error.message);
    return; // importantísimo: salimos para no seguir con datos inválidos
  }
  console.log("El archivo dice:", datos);
});
```

Esta convención existe por una razón práctica: te obliga a pensar en el error ANTES que en el éxito. En el mundo real los archivos no existen, los permisos fallan, la red se cae. El error-first te pone el manejo de errores en la cara, en la primera línea del callback, donde no puedes ignorarlo... aunque muchos programadores novatos lo intentan, y luego se preguntan por qué su programa truena con un error críptico tres funciones más abajo.

Fíjate en el "return" después de manejar el error. Es un detalle pequeño pero crucial: si no sales de la función, el código sigue corriendo con "datos" valiendo undefined, y el error real se pierde entre errores falsos más adelante. Acostúmbrate a ese patrón: revisa el error, manéjalo, salte.

Los callbacks también aparecen en operaciones que inventas tú. Cualquier función puede recibir un callback y llamarlo cuando termine. Por ejemplo, una función que simula una tarea lenta:


```
function tareaLenta(nombre, duracion, callback) {
  console.log("Iniciando:", nombre);
  setTimeout(function() {
    console.log("Terminada:", nombre);
    callback(null, "Resultado de " + nombre);
  }, duracion);
}

tareaLenta("descargar", 2000, function(error, resultado) {
  if (error) {
    console.error("Falló:", error);
    return;
  }
  console.log("Recibí:", resultado);
});

console.log("El programa sigue sin esperar...");
```

La salida muestra "Iniciando: descargar", luego "El programa sigue sin esperar...", y dos segundos después "Terminada: descargar" y "Recibí: Resultado de descargar". El `setTimeout` simula la operación lenta: Node programa el aviso y sigue con su vida.

Hasta aquí todo bonito. El problema empieza cuando necesitas hacer varias operaciones ENCADENADAS, donde cada una depende del resultado de la anterior. Por ejemplo: lee un archivo de configuración, con su contenido arma la ruta de otro archivo, lee ese segundo archivo, con su contenido consulta algo más... Cada paso necesita un callback dentro del callback anterior. Y ahí nace el monstruo: la pirámide de la perdición, también conocida como callback hell (el infierno de los callbacks).

Mira este ejemplo horrible, deliberadamente feo, para que sientas el dolor:

```

const fs = require("fs");

fs.readFile("paso1.txt", "utf8", function(error1, datos1) {
  if (error1) {
    console.error("Falló el paso 1:", error1.message);
  } else {
    fs.readFile(datos1.trim(), "utf8", function(error2, datos2) {
      if (error2) {
        console.error("Falló el paso 2:", error2.message);
      } else {
        fs.readFile(datos2.trim(), "utf8", function(error3, datos3) {
          if (error3) {
            console.error("Falló el paso 3:", error3.message);
          } else {
            console.log("Resultado final:", datos3);
            fs.writeFile("resultado.txt", datos3, function(error4) {
              if (error4) {
                console.error("Falló al guardar:", error4.message);
              } else {
                console.log("Guardado con éxito");
              }
            });
          }
        });
      }
    });
  }
});

```

¿Ves la pirámide? Cada nivel se indenta más a la derecha, el código se va angostando como una pirámide invertida, y para cuando llegas al cuarto nivel ya no sabes en qué nivel de indentación estás ni qué variable corresponde a qué paso. Esto con cuatro pasos; imagina un flujo real con ocho o diez. El callback hell tiene tres pecados capitales:

Primero, es ilegible. El flujo real del programa (paso 1, luego paso 2, luego paso 3) queda sepultado bajo la estructura de indentación. Leerlo de arriba a abajo NO te cuenta la historia en orden.

Segundo, el manejo de errores es un desastre. Cada nivel necesita su propio if de error, y es facilísimo olvidar uno. Si olvidas manejar el error en el nivel 3, el programa sigue con datos corruptos y truena en un lugar que no tiene nada que ver con la causa real.

Tercero, es imposible de mantener. ¿Quieres agregar un paso entre el 2 y el 3? Tienes que reindentar todo lo que sigue. ¿Quieres reutilizar el paso 2 en otro lado? Está atrapado dentro del callback del paso 1, inseparable.

Ahora, antes de que pienses que los callbacks son inútiles, mira la versión LEGIBLE del mismo problema usando callbacks bien organizados. El truco es nombrar tus funciones en lugar de anidar funciones anónimas:

```

const fs = require("fs");

function leerPaso1(callback) {
  fs.readFile("paso1.txt", "utf8", callback);
}

function leerPaso2(nombreArchivo, callback) {
  fs.readFile(nombreArchivo.trim(), "utf8", callback);
}

function leerPaso3(nombreArchivo, callback) {
  fs.readFile(nombreArchivo.trim(), "utf8", callback);
}

function guardarResultado(datos, callback) {
  fs.writeFile("resultado.txt", datos, callback);
}

function manejarError(paso, error) {
  console.error("Falló el " + paso + ":", error.message);
  process.exit(1);
}

leerPaso1(function(error1, datos1) {
  if (error1) return manejarError("paso 1", error1);
  leerPaso2(datos1, function(error2, datos2) {
    if (error2) return manejarError("paso 2", error2);
    leerPaso3(datos2, function(error3, datos3) {
      if (error3) return manejarError("paso 3", error3);
      console.log("Resultado final:", datos3);
      guardarResultado(datos3, function(error4) {
        if (error4) return manejarError("guardado", error4);
        console.log("Guardado con éxito");
      });
    });
  });
});

```

Sigue habiendo anidación, no nos engañemos, pero ahora cada paso tiene nombre, el manejo de errores está centralizado en una función, y el flujo se lee mejor. Esta es la forma "profesional" de escribir callbacks cuando no tienes otra opción. Pero seamos honestos: sigue siendo incómodo. La comunidad de Node lo sufrió durante años, y de ese sufrimiento nacieron las promesas.

Antes de pasar a las promesas, dos patrones con callbacks que debes conocer porque los verás en código real. El primero es el callback que se ejecuta varias veces, como en los eventos:

```
const EventEmitter = require("events");
const emisor = new EventEmitter();

emisor.on("pedido", function(cliente, tacos) {
  console.log(cliente + " pidió " + tacos + " tacos");
});

emisor.emit("pedido", "María", 5);
emisor.emit("pedido", "Juan", 3);
```

Aquí el callback no es "avísame cuando termines" sino "avísame CADA VEZ que pase esto". Es el mismo mecanismo, distinto uso, y es la base de todo el sistema de eventos de Node.

El segundo patrón es el error clásico de novato: llamar al callback dos veces o no llamarlo nunca. Si tu función asíncrona a veces llama al callback y a veces no (porque olvidaste un caso), el programa se queda colgado en silencio, esperando un aviso que nunca llega. Y si lo llamas dos veces, el código que sigue se ejecuta dos veces con resultados impredecibles. Regla de hierro: todo camino de tu función asíncrona debe llamar al callback exactamente una vez, con error-first siempre.

Promesas: orden en el caos asíncrono

Las promesas nacieron para resolver el callback hell, y lo lograron tan bien que hoy son el estándar del JavaScript moderno. Una promesa es un objeto que representa un valor futuro: algo que todavía no tienes, pero que tendrás (o fallará) más adelante. Es como el ticket que te dan en la fila de las tortillas: todavía no tienes las tortillas, pero tienes la garantía de que cuando sea tu turno, te las dan o te explican por qué no hay.

Toda promesa vive en uno de tres estados, y entenderlos es fundamental:

- pending (pendiente): la operación todavía no termina. Es el estado inicial de toda promesa. El ticket recién impreso, todavía no sabes si habrá tortillas.
- fulfilled (cumplida): la operación terminó con éxito y la promesa tiene un valor. Te entregaron las tortillas calientitas.
- rejected (rechazada): la operación falló y la promesa tiene una razón del falló (un error). Se acabaron las tortillas y te dicen por qué.

Un detalle crucial: una promesa solo cambia de estado UNA VEZ. De pendiente pasa a cumplida o a rechazada, y ahí se queda para siempre. No hay vuelta atrás, no hay cambios de opinión. Esta garantía es lo que hace a las promesas confiables: cuando te dicen que algo se cumplió, se cumplió.

Vamos a crear una promesa desde cero para ver su anatomía:

```
function tareaLenta(nombre, duracion, debeFallar) {
  return new Promise(function(resolve, reject) {
    console.log("Iniciando:", nombre);
    setTimeout(function() {
      if (debeFallar) {
        reject(new Error("Falló " + nombre));
      } else {
        console.log("Terminada:", nombre);
        resolve("Resultado de " + nombre);
      }
    }, duracion);
  });
}
```

El constructor Promise recibe una función ejecutora con dos parámetros: resolve (para cumplir la promesa con un valor) y reject (para rechazarla con un error). Dentro pones tu operación asíncrona, y cuando termina llamas a resolve o reject según corresponda. Fíjate que la función DEVUELVE la promesa de inmediato, aunque la operación todavía no termina: quien la recibe obtiene el "ticket" y decide qué hacer cuando llegue el resultado.

¿Cómo consumes una promesa? Con los métodos .then, .catch y .finally:

```
tareaLenta("descargar", 2000, false)
  .then(function(resultado) {
    console.log("Éxito:", resultado);
  })
  .catch(function(error) {
    console.error("Error:", error.message);
  })
  .finally(function() {
    console.log("Esto corre siempre, haya éxito o error");
  });

console.log("El programa sigue sin esperar...");
```

.then se ejecuta cuando la promesa se cumple, recibiendo el valor. .catch se ejecuta cuando se rechaza, recibiendo el error. .finally se ejecuta en ambos casos, y sirve para limpieza: cerrar conexiones, apagar indicadores de carga, liberar recursos. Es el equivalente elegante del "pase lo que pase, haz esto al final".

La verdadera magia de las promesas aparece en el ENCADENAMIENTO. Cada .then devuelve una NUEVA promesa, lo que te permite encadenar pasos en secuencia sin anidar nada. Mira cómo el callback hell de la sección anterior se transforma:

```
const fs = require("fs").promises;

fs.readFile("paso1.txt", "utf8")
  .then(function(datos1) {
    return fs.readFile(datos1.trim(), "utf8");
  })
  .then(function(datos2) {
    return fs.readFile(datos2.trim(), "utf8");
  })
  .then(function(datos3) {
    console.log("Resultado final:", datos3);
    return fs.writeFile("resultado.txt", datos3);
  })
  .then(function() {
    console.log("Guardado con éxito");
  })
  .catch(function(error) {
    console.error("Falló algo en la cadena:", error.message);
  });
```

¿Ves la diferencia? El flujo se lee de arriba a abajo: paso 1, paso 2, paso 3, guardar. Cada .then devuelve la promesa del siguiente paso, y el siguiente .then recibe su resultado. Y lo mejor: UN SOLO .catch al final atrapa el error de CUALQUIER paso de la cadena. Si falla el paso 2, los .then siguientes se saltan automáticamente y el control cae directo al .catch. Esto es lo que los callbacks no podían hacer: manejo de errores centralizado en flujos secuenciales.

Hay dos reglas de oro en el encadenamiento que los novatos rompen constantemente. Primera: si dentro de un .then inicias otra operación asíncrona, DEBES devolver su promesa con return. Si no la devuelves, el siguiente .then se ejecuta de inmediato con undefined, sin esperar a que termine tu operación. Es el error más común con promesas y el más difícil de detectar, porque el programa "funciona" pero en el orden equivocado.

Segunda: no mezcles estilos sin necesidad. Si ya estás en el mundo de las promesas, no metas callbacks anidados dentro de un `.then`; convierte todo a promesas o usa `async/await`. Mezclar estilos es como hablar dos idiomas en la misma frase: se puede, pero nadie lo disfruta.

Nota algo importante en el ejemplo: usé `require("fs").promises`. El módulo `fs` de Node tiene una versión basada en promesas integrada, y es la que deberías usar en código nuevo. Cada función (`readFile`, `writeFile`, `readdir`, etc.) devuelve una promesa en lugar de pedir un callback. Es el mismo `fs` de siempre, con la interfaz moderna.

Ahora hablemos de paralelismo, porque aquí las promesas brillan de verdad. Con callbacks, lanzar varias operaciones en paralelo y esperar a que terminen TODAS era un dolor de cabeza manual (contadores, banderas, código frágil). Con promesas es una línea: `Promise.all`.

```
const fs = require("fs").promises;

Promise.all([
  fs.readFile("archivo1.txt", "utf8"),
  fs.readFile("archivo2.txt", "utf8"),
  fs.readFile("archivo3.txt", "utf8")
])
  .then(function(contenidos) {
    console.log("Los tres archivos están listos:");
    console.log("Archivo 1:", contenidos[0]);
    console.log("Archivo 2:", contenidos[1]);
    console.log("Archivo 3:", contenidos[2]);
  })
  .catch(function(error) {
    console.error("Al menos uno falló:", error.message);
  });
```

`Promise.all` recibe un arreglo de promesas, las lanza TODAS AL MISMO TIEMPO (en paralelo, como el taquero poniendo varios tacos al comal), y devuelve una promesa que se cumple cuando todas terminan, entregándote los resultados en el mismo orden del arreglo. Si ALGUNA falla, `Promise.all` se rechaza de inmediato con ese error, sin esperar a las demás. Es perfecto para cuando necesitas varios resultados independientes: leer varios archivos, pedir datos a varias APIs, consultar varias tablas.

Su primo es `Promise.race`: también recibe un arreglo de promesas, pero se cumple (o rechaza) con la PRIMERA que termine, ignorando a las demás. ¿Para qué sirve? El caso clásico es un `timeout`: compites tu operación real contra una promesa que se rechaza después de N segundos, y si la operación tarda demasiado, "gana" el `timeout`:


```
function conTimeout(promesa, milisegundos) {
  const limite = new Promise(function(resolve, reject) {
    setTimeout(function() {
      reject(new Error("Se acabó el tiempo"));
    }, milisegundos);
  });
  return Promise.race([promesa, limite]);
}

const fs = require("fs").promises;

conTimeout(fs.readFile("grande.txt", "utf8"), 3000)
  .then(function(contenido) {
    console.log("Leído a tiempo:", contenido.length, "caracteres");
  })
  .catch(function(error) {
    console.error(error.message);
  });
```

Si el archivo se lee en menos de 3 segundos, gana la lectura. Si tarda más, gana el timeout y obtienes un error controlado en lugar de una espera eterna. Este patrón es oro en aplicaciones reales: ninguna operación de red debería esperar para siempre.

Hay dos compañeros más que debes conocer. `Promise.allSettled` es como `Promise.all` pero NUNCA se rechaza: espera a que todas terminen y te entrega el resultado de cada una indicando si se cumplió o se rechazó. Útil cuando quieres intentar varias cosas y te importa saber el destino de cada una, no solo que "al menos una falló". Y `Promise.any` se cumple con la PRIMERA promesa EXITOSA, ignorando los rechazos a menos que todas fallen. Útil cuando tienes varias fuentes redundantes y te basta con que una responda.

Ahora, la pregunta práctica: ¿qué hago con todo el código viejo y las librerías que usan callbacks? Node te da un puente: `util.promisify`. Esta función toma una función basada en callbacks (con convención error-first) y te devuelve una versión que devuelve promesas:

```

const fs = require("fs");
const util = require("util");

const readFilePromesa = util.promisify(fs.readFile);
const writeFilePromesa = util.promisify(fs.writeFile);

readFilePromesa("nota.txt", "utf8")
  .then(function(contenido) {
    console.log("Contenido:", contenido);
    return writeFilePromesa("copia.txt", contenido.toUpperCase());
  })
  .then(function() {
    console.log("Copia en mayúsculas guardada");
  })
  .catch(function(error) {
    console.error("Error:", error.message);
  });

```

útil.promisify solo funciona con funciones que siguen la convención error-first (callback con el error primero). Si la función usa otro formato de callback, tendrás que envolverla manualmente en new Promise, que ya sabes hacer. Este puente es tu mejor amigo cuando trabajas con librerías antiguas: las modernizas sin reescribirlas.

Un último tema de esta sección, y es serio: las promesas rechazadas que nadie atiende. Si una promesa se rechaza y no tiene ningún .catch (ni un await con try/catch, que veremos en la siguiente sección), Node emite un evento unhandledRejection. En versiones modernas de Node, esto TERMINA tu programa por defecto. Y tiene sentido: una promesa rechazada sin manejar es como una excepción que nadie atrapó; ignorarla escondería bugs gravísimos. La lección: toda cadena de promesas debe terminar en un .catch, sin excepciones. Si ves el error "UnhandledPromiseRejection" en tu terminal, no es mala suerte: es Node diciéndote que olvidaste manejar un posible falló.

Puedes escuchar ese evento globalmente como red de seguridad, pero úsalo solo para registrar el error y cerrar con dignidad, no como sustituto del manejo real:

```

process.on("unhandledRejection", function(error) {
  console.error("Promesa rechazada sin manejar:", error.message);
  process.exit(1);
});

```

Esto no te salva del bug, solo te avisa con claridad antes de morir. El manejo real siempre va en el .catch de cada cadena.

Async/await: código asíncrono que se lee de arriba a abajo

Si las promesas ordenaron el caos de los callbacks, async/await lo hizo desaparecer por completo... al menos visualmente. async/await es azúcar sintáctica sobre las promesas: no es un mecanismo nuevo, es una forma de escribir código basado en promesas que SE LEE como si fuera síncrono. Y esa diferencia de legibilidad es tan grande que hoy es el estilo recomendado para casi todo el código nuevo en Node.

Mira la transformación completa. El mismo flujo de leer tres archivos encadenados, ahora con async/await:

```
const fs = require("fs").promises;

async function procesarArchivos() {
  try {
    const datos1 = await fs.readFile("paso1.txt", "utf8");
    const datos2 = await fs.readFile(datos1.trim(), "utf8");
    const datos3 = await fs.readFile(datos2.trim(), "utf8");
    console.log("Resultado final:", datos3);
    await fs.writeFile("resultado.txt", datos3);
    console.log("Guardado con éxito");
  } catch (error) {
    console.error("Falló algo:", error.message);
  }
}

procesarArchivos();
```

Léelo en voz alta: "lee el paso 1, con eso lee el paso 2, con eso lee el paso 3, imprímelo, guárdalo, avisa que quedó". Se lee EXACTAMENTE en el orden en que ocurre. No hay .then, no hay anidación, no hay pirámide. Y el manejo de errores es un try/catch normal, el mismo que usas en código síncrono desde que aprendiste JavaScript. Esta es la frase clave de la unidad: código asíncrono que se lee de arriba a abajo.

Vamos por partes. La palabra async va antes de function (también funciona con funciones flecha: const fn = async () => {...}) y hace dos cosas: primero, garantiza que la función SIEMPRE devuelve una promesa, aunque tú devuelvas un valor normal; segundo, te permite usar await dentro de ella. Si tu función async devuelve 42, quien la llame recibe una promesa que se cumple con 42. Si lanzas un error dentro, quien la llame recibe una promesa rechazada. Es un contrato automático.

La palabra `await` va antes de una promesa y hace algo que parece magia: PAUSA la ejecución de tu función hasta que la promesa se resuelve, y luego te entrega su valor directamente, sin `.then`. Si la promesa se rechaza, `await` LANZA el error como si fuera una excepción normal, y por eso lo atrapas con `try/catch`. Ojo con la palabra "pausa": solo pausa TU función, no todo el programa. Mientras tu función espera, el event loop sigue atendiendo a todo lo demás. Es el taquero otra vez: tú dejas tu taco en el comal y te pones a hacer otra cosa; cuando está listo, regresas justo donde te quedaste.

Regla inquebrantable: `await` SOLO funciona dentro de funciones `async`. Si intentas usar `await` en una función normal, obtienes un error de sintaxis. (Hay una excepción moderna: el "top-level `await`" en módulos ES, pero en este curso usamos CommonJS con `require`, así que la regla se mantiene: envuelve tu código en una función `async` y llámala.)

El ejemplo canónico que pide el curso, leer un archivo con `fs.promises`, queda así de limpio:

```
const fs = require("fs").promises;

async function leerNota() {
  try {
    const datos = await fs.readFile("nota.txt", "utf8");
    console.log("La nota dice:", datos);
  } catch (error) {
    console.error("No se pudo leer la nota:", error.message);
  }
}

leerNota();
```

Una línea hace lo que antes tomaba un callback completo con su `if` de error: `const datos = await fs.readFile("nota.txt", "utf8")`. El `await` espera a que el disco termine, el `try/catch` maneja el error si el archivo no existe. Simple, directo, legible.

Ahora, los errores comunes, porque aquí es donde la gente se tropieza de verdad. Error número uno: OLVIDAR el `await`. Mira:

```
async function ejemplo() {
  const datos = fs.readFile("nota.txt", "utf8"); // FALTA await
  console.log(datos); // imprime: Promise { <pending> }
  console.log(datos.length); // undefined, no el contenido
}
```

Sin await, fs.readFile devuelve la promesa sin resolver, no el contenido. Tu variable "datos" es un objeto Promise pendiente, no un texto. El programa no truena de inmediato (las promesas son objetos válidos), así que el bug se manifiesta más adelante de formas confusas. Si ves "Promise { <pending >}" en tu consola, ya sabes: te faltó un await. Acostúmbrate a revisar cada llamada asíncrona y preguntarte: "¿lleva await?".

Error número dos: usar await dentro de forEach esperando que espere. NO funciona:

```
async function procesar(lista) {
  lista.forEach(async function(item) {
    const datos = await fs.readFile(item, "utf8");
    console.log(datos);
  });
  console.log("¿Ya terminé?"); // se imprime ANTES que los archivos
}
```

¿Por qué? Porque forEach no sabe nada de promesas: llama a tu función para cada elemento y no espera a nadie. Cada iteración corre por su cuenta y el forEach termina de inmediato. El "¿Ya terminé?" se imprime antes de que se lea el primer archivo. Si necesitas procesar elementos EN SECUENCIA (uno tras otro), usa un ciclo for...of normal, que SÍ respeta el await:

```
async function procesarEnSecuencia(lista) {
  for (const item of lista) {
    const datos = await fs.readFile(item, "utf8");
    console.log("Leído:", item);
  }
  console.log("Ahora sí terminé, en orden");
}
```

El for...of pausa en cada await y continúa con el siguiente elemento solo cuando el anterior terminó. Secuencia garantizada. Y si quieres procesarlos EN PARALELO (todos al mismo tiempo), combina map con Promise.all:

```
async function procesarEnParalelo(lista) {
  const promesas = lista.map(function(item) {
    return fs.readFile(item, "utf8");
  });
  const contenidos = await Promise.all(promesas);
  console.log("Todos listos:", contenidos.length, "archivos");
}
```

Aquí map crea un arreglo de promesas (todas arrancan al mismo tiempo) y un solo await con Promise.all espera a que terminen todas. Esta combinación (map + Promise.all) es uno de los patrones más usados en Node profesional. Apréndetelo de memoria: for...of para secuencia, map + Promise.all para paralelo.

Error número tres: envolver todo en try/catch gigantes que ocultan dónde falló. Es mejor tener bloques try/catch enfocados, o al menos incluir contexto en el error:

```
async function cargarConfiguracion() {
  try {
    const datos = await fs.readFile("config.json", "utf8");
    return JSON.parse(datos);
  } catch (error) {
    throw new Error("No se pudo cargar config.json: " + error.message);
  }
}
```

Al relanzar el error con contexto ("No se pudo cargar config.json"), quien atrape el error más arriba sabrá QUÉ operación falló, no solo "file not found". En aplicaciones reales, los errores sin contexto son pesadillas de depuración a las 3 de la mañana.

Error número cuatro: creer que async/await hace tu código más rápido. No. async/await no acelera nada; solo lo hace más legible. Tres awaits en secuencia tardan lo mismo que tres .then encadenados: la suma de los tres tiempos. Si las operaciones son independientes, el código secuencial desperdicia tiempo:

```
// LENTO: secuencial innecesario (tarda la suma de los tres)
const a = await fs.readFile("a.txt", "utf8");
const b = await fs.readFile("b.txt", "utf8");
const c = await fs.readFile("c.txt", "utf8");

// RÁPIDO: paralelo (tarda lo que tarde el más lento)
const [x, y, z] = await Promise.all([
  fs.readFile("a.txt", "utf8"),
  fs.readFile("b.txt", "utf8"),
  fs.readFile("c.txt", "utf8")
]);
```

La desestructuración `const [x, y, z] = await Promise.all(...)` es el patrón elegante para "necesito varios resultados independientes al mismo tiempo". Úsalo cada vez que veas awaits secuenciales cuyas operaciones no dependen entre sí.

Hablemos ahora de `fs.promises` en la práctica, porque es tu herramienta diaria. El objeto `fs.promises` expone casi todas las funciones de `fs` en versión promesa: `readFile`, `writeFile`, `appendFile`, `readdir`, `mkdir`, `stat`, `unlink`, `rename`, `copyFile`. Un script típico de procesamiento de archivos queda así:

```
const fs = require("fs").promises;
const path = require("path");

async function organizarDirectorio(ruta) {
  try {
    const entradas = await fs.readdir(ruta, { withFileTypes: true });
    for (const entrada of entradas) {
      const rutaCompleta = path.join(ruta, entrada.name);
      if (entrada.isFile()) {
        const info = await fs.stat(rutaCompleta);
        console.log(entrada.name + ": " + info.size + " bytes");
      } else if (entrada.isDirectory()) {
        console.log(entrada.name + " / (carpeta)");
      }
    }
  } catch (error) {
    console.error("No se pudo leer el directorio:", error.message);
  }
}

organizarDirectorio("./documentos");
```

Fíjate en el `for...of` con `awaits` adentro: lee cada entrada en secuencia. Para un directorio con cientos de archivos, la versión paralela con `map` + `Promise.all` sería más rápida, pero la secuencial es más fácil de leer y suficiente para la mayoría de los scripts. La elegancia también es una forma de rendimiento: el código que entiendes es código que no rompes.

Cuándo usar cada estilo y buenas prácticas

Ya conoces los tres estilos. La pregunta ahora es: ¿cuándo uso cada uno? La respuesta corta: usa `async/await` por defecto, promesas cuando necesites composición avanzada, y `callbacks` solo cuando no tengas opción.

`async/await` es tu herramienta principal para código nuevo: flujos secuenciales, scripts, manejadores de peticiones en servidores, cualquier cosa que se lea mejor de arriba a abajo. Es lo que los equipos modernos esperan ver y lo que te pedirán en entrevistas.

Las promesas "puras" (con `.then`) siguen siendo útiles en casos específicos: cuando construyes utilidades que combinan promesas (como nuestra función `conTimeout` con `Promise.race`), cuando trabajas con `Promise.all/allSettled/any` en patrones de composición, o cuando escribes una función que devuelve una promesa sin necesidad de pausar nada en medio. También verás `.then` en código que encadena transformaciones rápidas donde `async/await` agregaría verbosidad innecesaria.

Los callbacks quedan para tres situaciones: código heredado que no puedes reescribir, librerías antiguas sin versión de promesas (ahí entra `util.promisify` como puente), y manejadores de eventos (`emisor.on`), donde el patrón "avísame cada vez" es natural y no hay promesa que lo reemplace bien.

Las buenas prácticas que separan al profesional del aficionado en código asíncrono:

Primera: maneja SIEMPRE los errores. Toda cadena de promesas termina en `.catch`, toda función `async` que llames debería tener su `try/catch` o un `.catch` al invocarla. Las promesas rechazadas sin manejar matan tu programa en Node moderno, y con razón.

Segunda: no mezcles estilos en el mismo flujo. Si empezaste con `async/await`, no metas un `.then` a la mitad ni un callback anidado. La consistencia es legibilidad, y la legibilidad es mantenibilidad.

Tercera: piensa en paralelo por defecto. Ante varios `awaits`, pregúntate: "¿estos dependen entre sí?". Si no, `Promise.all`. El código secuencial innecesario es el desperdicio más común en aplicaciones Node, y también el más fácil de corregir.

Cuarta: nunca bloquee el event loop. Nada de ciclos pesados ni operaciones síncronas lentas (como `readFileSync`) en código que atiende usuarios. El hilo principal es sagrado: coordinación rápida, nada más.

Quinta: dale contexto a tus errores. Un "file not found" no dice nada; un "No se pudo cargar la configuración de config.json: file not found" lo dice todo. Tu yo del futuro, depurando a medianoche, te lo agradecerá.

Sexta: cuidado con el `await` en ciclos cuando el orden no importa. Un `for...of` con `await` es secuencial por diseño; si procesas mil archivos así, tardarás mil veces lo de uno. Para volúmenes grandes sin dependencia de orden, procesa en lotes con `Promise.all` o usa un límite de concurrencia.

Con esto cierras la teoría. Tienes el mapa completo: el event loop como motor, los callbacks como origen, las promesas como orden, y async/await como la forma moderna de escribirlo todo. Ahora vamos a verlo en acción con ejemplos paso a paso que puedes correr en tu máquina.

Preguntas frecuentes de la asincronía

Antes de pasar a los ejemplos, respondamos las dudas que todo estudiante trae a esta unidad, porque si te las estás preguntando, vas por buen camino.

¿Si await pausa mi función, no estoy bloqueando el programa? No. Esta es la confusión número uno. await pausa SOLO tu función async, como poner tu taco en el comal y voltear a atender otra cosa. El event loop sigue corriendo, otros callbacks se ejecutan, otros usuarios son atendidos. Tu función "duerme" sin consumir el hilo: cuando la promesa se resuelve, el event loop la despierta justo donde se quedó. Bloquear sería un ciclo while infinito o un `readFileSync`: eso sí congela todo porque el hilo principal nunca se libera.

¿Entonces Node sí hace varias cosas "al mismo tiempo"? Sí y no. Las operaciones de entrada/salida (disco, red) sí ocurren en paralelo real, manejadas por el sistema operativo y el thread pool interno. Pero tu código JavaScript siempre ejecuta una instrucción a la vez, en un solo hilo. La magia es la orquestación: mientras el disco lee tu archivo, tu JavaScript ya está atendiendo otra petición. Es concurrencia (muchas tareas en progreso) sin paralelismo en tu código (una instrucción a la vez). Y eso es una ventaja: jamás pelearás con dos hilos modificando la misma variable al mismo tiempo.

¿Puedo mezclar callbacks, promesas y async/await en un mismo proyecto? Puedes, y en proyectos reales pasará, porque las librerías viejas usan callbacks. Pero no los mezcles en el MISMO flujo: si una función es async, que todo su interior sea await y promesas; si consumes una librería con callbacks, envuélvela una vez con `util.promisify` en la frontera y olvídate. La mezcla dentro de un mismo flujo es la receta para bugs de orden imposibles de depurar.

¿Qué pasa si olvido el try/catch en una función async? Si la promesa que devuelve tu función async se rechaza y nadie la atiende (ni try/catch interno, ni `.catch` al llamarla), Node emite `unhandledRejection` y en versiones modernas TERMINA el proceso. No es un warning decorativo: es Node diciéndote que un error se perdió en el limbo. Acostúmbrate a que cada llamada a una función async lleve su manejo de error, igual que cada cadena de promesas termina en `.catch`.

¿async/await reemplaza por completo a las promesas? No las reemplaza: las USA. Todo await trabaja sobre promesas, y async siempre devuelve promesas. Necesitas entender promesas para entender qué hace await por debajo, para usar `Promise.all`, para escribir timeouts con

Promise.race, y para leer el montón de código que usa .then directamente. Piensa en las promesas como el motor y en async/await como la carrocería bonita: no puedes manejar bien el carro sin saber que hay un motor.

Ejemplos paso a paso

La teoría ya la tienes; ahora vamos a ensuciarnos las manos. Todos los ejemplos de esta sección usan archivos de texto reales, así que antes de correrlos crea estos tres archivos en la misma carpeta de tus scripts:

archivo1.txt con el contenido: Hola desde el archivo uno archivo2.txt con el contenido: Saludos desde el archivo dos archivo3.txt con el contenido: Adiós desde el archivo tres

Puedes crearlos con tu editor o desde la terminal. Con eso listo, vamos ejemplo por ejemplo, del problema a la solución.

Ejemplo 1: leer 3 archivos con callbacks anidados (el problema)

Este es el estilo clásico de Node, el que encontrarás en tutoriales viejos y código heredado. Queremos leer los tres archivos en orden y mostrar sus contenidos:

```

const fs = require("fs");

console.log("Iniciando lectura con callbacks...");

fs.readFile("archivo1.txt", "utf8", function(error1, contenido1) {
  if (error1) {
    console.error("Error en archivo1:", error1.message);
    return;
  }
  console.log("Archivo 1:", contenido1);

  fs.readFile("archivo2.txt", "utf8", function(error2, contenido2) {
    if (error2) {
      console.error("Error en archivo2:", error2.message);
      return;
    }
    console.log("Archivo 2:", contenido2);

    fs.readFile("archivo3.txt", "utf8", function(error3, contenido3) {
      if (error3) {
        console.error("Error en archivo3:", error3.message);
        return;
      }
      console.log("Archivo 3:", contenido3);
      console.log("Lectura con callbacks terminada.");
    });
  });
});

console.log("El programa no se detuvo a esperar.");

```

Paso a paso: primero pedimos archivo1 con su callback error-first. Dentro de ese callback, si no hubo error, pedimos archivo2. Dentro del callback de archivo2, pedimos archivo3. Cada nivel maneja su propio error con el patrón `if (error) + return`. Al final, la línea "El programa no se detuvo a esperar" se imprime PRIMERO, antes que cualquier contenido, porque las lecturas son asíncronas.

Observa el problema con ojos críticos: el flujo real (leer 1, 2, 3) está enterrado en la indentación. El manejo de errores se repite tres veces con el mismo patrón. Si mañana necesitas leer un cuarto archivo, agregas otro nivel de indentación a todo lo que ya existe. Funciona, pero duele.

Ejemplo 2: lo mismo con promesas encadenadas

Ahora la misma tarea con `fs.promises` y encadenamiento `.then`. Nota cómo el código se "aplana":

```
const fs = require("fs").promises;

console.log("Iniciando lectura con promesas...");

fs.readFile("archivo1.txt", "utf8")
  .then(function(contenido1) {
    console.log("Archivo 1:", contenido1);
    return fs.readFile("archivo2.txt", "utf8");
  })
  .then(function(contenido2) {
    console.log("Archivo 2:", contenido2);
    return fs.readFile("archivo3.txt", "utf8");
  })
  .then(function(contenido3) {
    console.log("Archivo 3:", contenido3);
    console.log("Lectura con promesas terminada.");
  })
  .catch(function(error) {
    console.error("Algo falló en la cadena:", error.message);
  });

console.log("El programa no se detuvo a esperar.");
```

Paso a paso: la primera lectura devuelve una promesa. Su `.then` recibe el contenido, lo imprime y DEVUELVE la promesa de la segunda lectura (ese `return` es sagrado: sin él, la cadena se rompe). El siguiente `.then` recibe el contenido del archivo 2, lo imprime y devuelve la tercera lectura. El último `.then` imprime el archivo 3. Y un solo `.catch` al final atrapa el error de cualquiera de los tres pasos.

Compara con el ejemplo 1: el flujo se lee de arriba a abajo, el manejo de errores está en un solo lugar, y agregar un cuarto archivo es agregar un `.then` más, sin reindentar nada. Esta es la victoria de las promesas sobre los callbacks.

Ejemplo 3: lo mismo con `async/await` (la solución)

La versión moderna, la que deberías escribir en código nuevo:

```

const fs = require("fs").promises;

async function leerTresArchivos() {
  try {
    console.log("Iniciando lectura con async/await...");

    const contenido1 = await fs.readFile("archivo1.txt", "utf8");
    console.log("Archivo 1:", contenido1);

    const contenido2 = await fs.readFile("archivo2.txt", "utf8");
    console.log("Archivo 2:", contenido2);

    const contenido3 = await fs.readFile("archivo3.txt", "utf8");
    console.log("Archivo 3:", contenido3);

    console.log("Lectura con async/await terminada.");
  } catch (error) {
    console.error("Algo falló:", error.message);
  }
}

leerTresArchivos();
console.log("El programa no se detuvo a esperar.");

```

Paso a paso: la función `async` nos permite usar `await`. Cada `await` pausa la función hasta que el archivo se lee y nos entrega el contenido directamente en una constante, como si fuera código síncrono. El `try/catch` envuelve todo: cualquier falló en cualquiera de las tres lecturas cae al `catch`. No hay `.then`, no hay `callbacks`, no hay indentación piramidal.

Los tres ejemplos hacen exactamente lo mismo y tardan exactamente lo mismo (las lecturas son secuenciales en los tres casos). La diferencia es pura legibilidad y mantenibilidad. Cuando alguien te pregunte por qué `async/await` ganó, muéstrale estos tres ejemplos lado a lado: esa es toda la explicación.

Ejemplo 4: Promise.all para leer varios archivos en paralelo

Los ejemplos anteriores leen en secuencia porque cada paso dependía del anterior (en el ejemplo original de la pirámide). Pero nuestros tres archivos son independientes: no necesitamos esperar al archivo 1 para pedir el archivo 2. Lanzarlos en paralelo es más rápido:

```

const fs = require("fs").promises;

async function leerEnParalelo() {
  try {
    console.log("Lanzando las tres lecturas al mismo tiempo...");

    const inicio = Date.now();

    const contenidos = await Promise.all([
      fs.readFile("archivo1.txt", "utf8"),
      fs.readFile("archivo2.txt", "utf8"),
      fs.readFile("archivo3.txt", "utf8")
    ]);

    const fin = Date.now();
    console.log("Las tres lecturas tomaron " + (fin - inicio) + " ms");

    contenidos.forEach(function(contenido, indice) {
      console.log("Archivo " + (indice + 1) + ":", contenido);
    });
  } catch (error) {
    console.error("Al menos una lectura falló:", error.message);
  }
}

leerEnParalelo();

```

Paso a paso: creamos un arreglo con las tres promesas de lectura. Las tres ARRANCAN de inmediato, en paralelo, como tres tacos en el comal al mismo tiempo. Promise.all devuelve una promesa que se cumple cuando las tres terminan, y el await nos entrega un arreglo con los tres contenidos EN EL MISMO ORDEN del arreglo original (aunque el archivo 3 haya terminado antes que el 1, contenidos[2] siempre es el archivo 3). Medimos el tiempo para comprobar la ganancia: en paralelo tardas lo que tarde la lectura más lenta, no la suma de las tres.

Compara los tiempos con el ejemplo 3 si quieres sentir la diferencia: agrega la misma medición de Date.now() a la versión secuencial y verás que tarda aproximadamente la suma de los tres tiempos. En archivos pequeños la diferencia es de milisegundos, pero con operaciones de red que tardan segundos, el paralelo contra el secuencial es la diferencia entre una aplicación rápida y una insoportable.

Recuerda la regla: `Promise.all` se rechaza si ALGUNA promesa falla. Si necesitas que cada lectura se intente aunque otras fallen, usa `Promise.allSettled` y revisa el estado de cada resultado.

Ejemplo 5: script robusto con try/catch y errores con contexto

Cerremos con un script completo, del tipo que escribirías en un trabajo real: lee un archivo de configuración JSON, valida su contenido, y genera un reporte. Con manejo de errores en cada capa y mensajes que dicen exactamente qué falló:

```

const fs = require("fs").promises;
const path = require("path");

async function leerJsonSeguro(ruta) {
  try {
    const texto = await fs.readFile(ruta, "utf8");
    return JSON.parse(texto);
  } catch (error) {
    throw new Error("No se pudo cargar " + ruta + ": " + error.message);
  }
}

async function generarReporte() {
  try {
    const config = await leerJsonSeguro("config.json");

    if (!config.titulo || !config.archivos) {
      throw new Error("config.json no tiene 'titulo' ni 'archivos'");
    }

    console.log("Generando reporte: " + config.titulo);

    const lecturas = config.archivos.map(function(nombre) {
      return fs.readFile(nombre, "utf8").catch(function(error) {
        return "[ERROR al leer " + nombre + ": " + error.message + "]";
      });
    });

    const contenidos = await Promise.all(lecturas);

    let reporte = "# " + config.titulo + "\n\n";
    config.archivos.forEach(function(nombre, indice) {
      reporte += "## " + nombre + "\n" + contenidos[indice] + "\n\n";
    });

    await fs.writeFile("reporte.md", reporte);
    console.log("Reporte guardado en reporte.md");
  } catch (error) {
    console.error("No se pudo generar el reporte:", error.message);
    process.exit(1);
  }
}

generarReporte();

```


Paso a paso, porque este ejemplo junta todo lo aprendido: `leerJsonSeguro` es una función auxiliar que lee y parsea un JSON, y si algo falla relanza el error CON CONTEXTO (dice qué archivo falló y por qué). `generarReporte` la usa para cargar `config.json`, valida que tenga los campos necesarios (si no, lanza un error descriptivo), y luego lee todos los archivos listados en `config.archivos` EN PARALELO con `map + Promise.all`.

Fíjate en el detalle fino: cada lectura individual lleva su propio `.catch` que devuelve un texto de error en lugar de rechazar. Así, si un archivo de la lista no existe, `Promise.all` NO se rechaza: ese archivo simplemente aporta su mensaje de error al reporte y los demás se procesan normalmente. Es el patrón "que fallen los que fallen, yo sigo con los que sí pude": ideal para reportes, migraciones y procesamientos por lotes donde un elemento roto no debe tumbar todo el trabajo.

Al final, el reporte se arma concatenando textos y se guarda con `writeFile`. Si CUALQUIER cosa estructural falla (no hay `config.json`, el JSON está mal formado, falta el campo `título`), el `catch` principal lo atrapa, imprime un mensaje claro y termina con `process.exit(1)` para señalar el fracaso al sistema operativo (los scripts bien hechos devuelven código distinto de cero cuando fallan; así otros programas y scripts pueden detectar el error).

Para probarlo, crea un `config.json` así:

```
{
  "titulo": "Mi primer reporte",
  "archivos": ["archivo1.txt", "archivo2.txt", "archivo3.txt"]
}
```

Y corre el script. Luego prueba romper cosas a propósito: borra `config.json`, ponle JSON inválido, quita el campo `"archivos"`, agrega a la lista un archivo que no existe. Cada escenario produce un mensaje de error distinto y claro. Esa es la marca de un script robusto: no evita que las cosas fallen (eso es imposible), sino que cuando fallan, te dice exactamente qué pasó y dónde.

Práctica

Llegó la hora de sudar. Estos ejercicios están ordenados de menor a mayor dificultad y cada uno incluye su solución explicada paso a paso. Intenta resolverlos ANTES de leer la solución: la asincronía solo se aprende escribiendo código que falla y entendiendo por qué falló. Si un ejercicio te toma más de 20 minutos atorado, lee la explicación del primer paso, inténtalo de nuevo, y repite.

Ejercicio 1: predice el orden de ejecución

Sin correr el código, escribe en qué orden se imprimirán las letras. Luego córrelo y compara.

```
console.log("1");

fs.readFile("archivo1.txt", "utf8", function() {
  console.log("2");
});

Promise.resolve().then(function() {
  console.log("3");
});

setTimeout(function() {
  console.log("4");
}, 0);

console.log("5");
```

(Asume que ya hiciste `require("fs")` al inicio.)

Solución paso a paso:

Paso 1: identifica el código síncrono. `console.log("1")` corre de inmediato. Luego viene `fs.readFile`, que es asíncrono: Node pide la lectura y sigue. `Promise.resolve().then(...)` programa una microtarea. `setTimeout(..., 0)` programa un timer. `console.log("5")` corre de inmediato. Hasta aquí, lo síncrono imprime 1 y 5, en ese orden.

Paso 2: el hilo principal se libera y el event loop atiende las microtareas primero. La promesa ya resuelta tiene pase VIP: se imprime 3.

Paso 3: en la fase de timers corre el `setTimeout` de 0 ms: se imprime 4.

Paso 4: la lectura del archivo depende del sistema operativo y del thread pool; en la práctica termina después de los timers en este ejemplo simple: se imprime 2.

Orden final: 1, 5, 3, 4, 2. Si lo predijiste bien, ya piensas como el event loop. Si fallaste en el 2 al final, recuerda: las operaciones de I/O reales (disco, red) no tienen tiempo garantizado; lo único seguro es que el código síncrono va primero, luego las microtareas, luego los timers, y el I/O cuando el sistema lo complete.

Ejercicio 2: convierte callbacks a promesas con `util.promisify`

Tienes esta función vieja basada en callbacks que simula consultar el clima de una ciudad (tarda 1 segundo y a veces falla):

```
function consultarClima(ciudad, callback) {
  setTimeout(function() {
    if (ciudad === "") {
      callback(new Error("Ciudad vacía"), null);
    } else {
      callback(null, "Soleado en " + ciudad + ", 28 grados");
    }
  }, 1000);
}
```

Tu tarea: sin modificar `consultarClima`, crea una versión basada en promesas usando `util.promisify`, consúmela con `async/await`, y maneja el error cuando la ciudad sea "".

Solución paso a paso:

Paso 1: verifica que la función sigue la convención error-first. Sí: el callback recibe (error, resultado). Entonces `util.promisify` puede envolverla.

Paso 2: crea la versión promisifyada y úsala dentro de una función `async` con `try/catch`:

```

const util = require("util");

function consultarClima(ciudad, callback) {
  setTimeout(function() {
    if (ciudad === "") {
      callback(new Error("Ciudad vacía"), null);
    } else {
      callback(null, "Soleado en " + ciudad + ", 28 grados");
    }
  }, 1000);
}

const consultarClimaAsync = util.promisify(consultarClima);

async function mostrarClima(ciudad) {
  try {
    const clima = await consultarClimaAsync(ciudad);
    console.log(clima);
  } catch (error) {
    console.error("No se pudo consultar:", error.message);
  }
}

mostrarClima("Guadalajara");
mostrarClima("");

```

Paso 3: entiende qué pasó. `util.promisify` devolvió una función que ya no pide callback: devuelve una promesa que se cumple con "Soleado en..." o se rechaza con el Error. El `await` la pausa un segundo y el `try/catch` maneja la ciudad vacía. La salida muestra primero "Soleado en Guadalajara, 28 grados" y luego "No se pudo consultar: Ciudad vacía" (ambas consultas corren en paralelo porque llamamos a `mostrarClima` dos veces sin `await` entre ellas; el orden de llegada puede variar, pero cada una maneja su propio resultado).

Moraleja: no necesitas reescribir librerías viejas para usarlas con `async/await`; el puente `util.promisify` las moderniza en una línea.

Ejercicio 3: encadena tres operaciones donde cada una depende de la anterior

Crea tres archivos: `semilla.txt` contiene el texto `"archivo2.txt"`, `archivo2.txt` contiene el texto `"archivo3.txt"`, y `archivo3.txt` contiene el mensaje final `"Llegaste al final de la cadena"`. Escribe un script con `async/await` que lea `semilla.txt`, use su contenido como nombre del siguiente archivo, lea ese, use su contenido como nombre del último, lo lea e imprima el mensaje final. Maneja errores con `try/catch`.

Solución paso a paso:

Paso 1: crea los archivos. El truco del ejercicio es que el CONTENIDO de cada archivo es el NOMBRE del siguiente. Es una cadena donde cada eslabón te dice dónde está el siguiente.

Paso 2: escribe el script. Cada lectura depende del resultado anterior, así que van en secuencia con `await`:

```
const fs = require("fs").promises;

async function seguirCadena() {
  try {
    const nombre2 = await fs.readFile("semilla.txt", "utf8");
    console.log("La semilla apunta a:", nombre2.trim());

    const nombre3 = await fs.readFile(nombre2.trim(), "utf8");
    console.log("El segundo apunta a:", nombre3.trim());

    const mensaje = await fs.readFile(nombre3.trim(), "utf8");
    console.log("Mensaje final:", mensaje.trim());
  } catch (error) {
    console.error("La cadena se rompió:", error.message);
  }
}

seguirCadena();
```

Paso 3: nota los `.trim()`. Los archivos de texto suelen terminar con un salto de línea invisible, y `"archivo2.txt\n"` no es un nombre de archivo válido. El `trim()` quita esos espacios y saltos de línea. Olvidar el `trim` es un bug clásico en este tipo de ejercicios: el error diría algo como `"ENOENT: no such file or directory"` con un nombre que SE VE bien pero trae un salto de línea escondido.

Paso 4: prueba romper la cadena a propósito (borra `archivo2.txt`) y observa cómo el `catch` atrapa el error con un mensaje claro. Este ejercicio es la versión `async/await` del `callback hell` original: tres pasos encadenados que se leen como una lista de instrucciones.

Ejercicio 4: lee 5 archivos en paralelo y mide la ganancia

Crea 5 archivos (dato1.txt a dato5.txt) con cualquier contenido. Escribe dos versiones: una que los lea en secuencia con `for...of` y `await`, y otra que los lea en paralelo con `Promise.all`. Mide el tiempo de cada una con `Date.now()` e imprime la diferencia.

Solución paso a paso:

Paso 1: crea los archivos. Puedes hacerlo rápido desde la terminal o con un script pequeño.

Paso 2: la versión secuencial:

```
const fs = require("fs").promises;

async function leerSecuencial(archivos) {
  const inicio = Date.now();
  const resultados = [];
  for (const nombre of archivos) {
    const contenido = await fs.readFile(nombre, "utf8");
    resultados.push(contenido);
  }
  console.log("Secuencial: " + (Date.now() - inicio) + " ms");
  return resultados;
}
```

El `for...of` con `await` procesa un archivo a la vez: pide el primero, espera, pide el segundo, espera... El tiempo total es la suma de las cinco lecturas.

Paso 3: la versión en paralelo:

```

async function leerParalelo(archivos) {
  const inicio = Date.now();
  const promesas = archivos.map(function(nombre) {
    return fs.readFile(nombre, "utf8");
  });
  const resultados = await Promise.all(promesas);
  console.log("Paralelo: " + (Date.now() - inicio) + " ms");
  return resultados;
}

const archivos = ["dato1.txt", "dato2.txt", "dato3.txt", "dato4.txt", "dato5.txt"];

async function comparar() {
  await leerSecuencial(archivos);
  await leerParalelo(archivos);
}

comparar();

```

Paso 4: interpreta los resultados. Con archivos pequeños en disco local, ambas versiones tardarán pocos milisegundos y la diferencia será mínima (el disco local es rápido y el sistema operativo cachea). Para sentir la diferencia real, imagina que cada "lectura" tarda 1 segundo (como una petición de red): la secuencial tardaría 5 segundos y la paralela 1 segundo. Puedes simularlo agregando una pequeña espera artificial, o simplemente confía en la matemática: secuencial suma tiempos, paralelo toma el máximo.

Paso 5: la lección del ejercicio. Ante operaciones independientes, el `for...of` con `await` es un desperdicio silencioso: funciona, se ve limpio, pero es lento. El patrón `map` + `Promise.all` debería ser tu reflejo automático cuando el orden entre operaciones no importa.

Ejercicio 5: el reto final, mini-procesador de pedidos con timeout

Escribe un script que simule procesar pedidos de tacos (en honor al taquero). Cada pedido tarda un tiempo aleatorio entre 1 y 5 segundos en "prepararse". Si un pedido tarda más de 3 segundos, se cancela con `timeout`. Procesa 4 pedidos en paralelo, y al final imprime un resumen: cuántos se completaron y cuántos se cancelaron. Usa `Promise.allSettled` para que un pedido cancelado no tumbe a los demás.

Solución paso a paso:

Paso 1: simula la preparación de un pedido como una promesa que se resuelve después de un tiempo aleatorio:

```
function prepararPedido(numero) {
  const segundos = 1 + Math.floor(Math.random() * 5); // entre 1 y 5
  return new Promise(function(resolve) {
    setTimeout(function() {
      resolve("Pedido " + numero + " listo en " + segundos + "s");
    }, segundos * 1000);
  });
}
```

Paso 2: crea la función de timeout con Promise.race, como vimos en la teoría:

```
function conLimite(promesa, segundos, numero) {
  const limite = new Promise(function(resolve, reject) {
    setTimeout(function() {
      reject(new Error("Pedido " + numero + " cancelado: tardó más de " + segundos
+ "s"));
    }, segundos * 1000);
  });
  return Promise.race([promesa, limite]);
}
```

Paso 3: procesa los 4 pedidos en paralelo con allSettled para tolerar las cancelaciones:


```

async function atenderTaqueria() {
  const pedidos = [1, 2, 3, 4].map(function(numero) {
    return conLimite(prepararPedido(numero), 3, numero);
  });

  const resultados = await Promise.allSettled(pedidos);

  let completados = 0;
  let cancelados = 0;

  resultados.forEach(function(resultado) {
    if (resultado.status === "fulfilled") {
      completados++;
      console.log("OK:", resultado.value);
    } else {
      cancelados++;
      console.log("CANCELADO:", resultado.reason.message);
    }
  });

  console.log("Resumen: " + completados + " completados, " + cancelados + " cancelados");
}

atenderTaqueria();

```

Paso 4: entiende por qué `allSettled` y no `all`. Con `Promise.all`, el primer pedido cancelado rechazaría TODO el lote y perderías los resultados de los pedidos que sí se completaron. Con `allSettled`, cada pedido aporta su veredicto: `fulfilled` con su `value` o `rejected` con su `reason`. El resumen final cuenta cada caso. Es exactamente el patrón que usarías en producción cuando procesas lotes donde algunos elementos pueden fallar: pagos, envíos de correos, sincronizaciones.

Paso 5: corre el script varias veces y observa cómo el resumen cambia (los tiempos son aleatorios). A veces se completan los 4, a veces se cancelan 2 o 3. La aleatoriedad es la gracia: estás viendo al event loop atender cuatro "tacos" en paralelo, cancelando los que tardan demasiado, sin bloquearse jamás. Eso, en miniatura, es Node.js en producción.

Resumen de la unidad

Llegaste al final de la unidad más importante del curso. Vamos a amarrar todo lo aprendido en una red que puedas repasar cuando lo necesites.

La asincronía es la respuesta de Node a las operaciones lentas. En lugar de bloquearse esperando al disco o a la red, Node delega la espera, sigue trabajando, y retoma el resultado cuando está listo. Esa es toda la filosofía en una frase, y todo lo demás son mecanismos para vivirla en código.

El event loop es el motor que lo hace posible: un ciclo infinito en un solo hilo que atiende callbacks conforme las operaciones terminan. Sus fases (timers, I/O pendiente, poll, check, close) y la cola prioritaria de microtarefas explican el orden real de ejecución de tu programa, que no siempre coincide con el orden en que escribiste el código. Las analogías del taquero y del cajero eficiente del banco no son adornos: son modelos mentales que te van a servir cada vez que depures un problema de orden o de rendimiento. Y la regla de oro que nunca debes romper: no bloques el hilo principal.

Los callbacks fueron la primera forma de manejar la asincronía: le pasas una función a la operación para que te avise cuando termine, siguiendo la convención error-first (el error siempre primero, o null si todo salió bien). Su pecado original es la pirámide de la perdición: callbacks anidados que vuelven el código ilegible y el manejo de errores frágil. Cuando no tengas opción, organiza tus callbacks con funciones nombradas y manejo centralizado de errores, y recuerda la regla de hierro: llama al callback exactamente una vez por cada camino de ejecución.

Las promesas trajeron orden: un objeto con tres estados (pending, fulfilled, rejected) que representa un valor futuro y que solo cambia de estado una vez. Con `.then` encadenas pasos que se leen de arriba a abajo, con un solo `.catch` al final para todos los errores, y `.finally` para la limpieza garantizada. `Promise.all` paraleliza operaciones independientes, `Promise.race` implementa timeouts compitiendo contra el reloj, `allSettled` tolera fallos parciales y `any` se queda con el primer éxito. Y con `util.promisify` modernizas cualquier función vieja basada en callbacks sin reescribirla.

`async/await` es la sintaxis moderna sobre las promesas: código asíncrono que se lee de arriba a abajo. `async` marca la función y la obliga a devolver una promesa; `await` pausa tu función (no el programa) hasta que la promesa se resuelve y te entrega el valor, o lanza el error para que lo atrape tu `try/catch`. Con `fs.promises`, leer un archivo es una línea: `const datos = await fs.readFile("nota.txt", "utf8")`. Los errores clásicos que debes evitar: olvidar el `await` (y recibir una promesa pendiente en lugar del valor), usar `await` en `forEach` esperando que espere (usa `for...of` para secuencia o `map` + `Promise.all` para paralelo), y encadenar `awaits` secuenciales cuando las operaciones son independientes.

Las buenas prácticas que te llevas: maneja siempre los errores (toda cadena termina en `.catch`, toda función `async` se invoca con su `try/catch` o `.catch`), no mezcles estilos en el mismo

flujo, piensa en paralelo por defecto ante awaits independientes, nunca bloquee el event loop, y dale contexto a tus errores para que depurar no sea una pesadilla. Y cuando uses cada estilo: async/await por defecto, promesas puras para composición avanzada, callbacks solo donde no haya alternativa.

Lo que sigue en el curso construye directamente sobre esta unidad: los servidores web, las APIs y las bases de datos son asincronía pura de principio a fin. Cada endpoint que escribas será una función async, cada consulta una promesa, cada error un catch bien puesto. Si esta unidad te costó trabajo, no te preocupes: a todos les cuesta al principio, y la recompensa es que ahora entiendes el corazón de Node.js mejor que la mayoría. Práctica los ejercicios, rompe cosas a propósito, lee los mensajes de error con calma. El taquero no aprendió a atender diez pedidos a la vez en un día; tú tampoco necesitas dominar la asincronía en una sentada. Pero ya tienes el comal encendido y los primeros tacos en el fuego. Nos vemos en la Unidad 8, donde pondremos toda esta asincronía a trabajar en un servidor web de verdad.

UNIDAD 8: Express desde cero: tu primer framework

Bienvenido a la unidad 8. Hasta aquí has recorrido un camino que ya te pone por encima de la mayoría de la gente que dice "quiero aprender Node.js": conoces el entorno de ejecución, sabes crear módulos, entiendes el ciclo de eventos, dominas la diferencia entre código síncrono y asíncrono, leíste y escribiste archivos con el módulo `fs`, levantaste tu primer servidor con el módulo `http` nativo y aprendiste a manejar rutas, query strings y cuerpos de peticiones a pura mano, parseando texto crudo y contestando con `res.writeHead` y `res.end`. Eso no es poca cosa. De hecho, todo lo que hiciste con `http` nativo es exactamente lo que ocurre debajo del capó de cada framework web que existe en Node.js. Pero también es cierto que hacer todo a mano, aunque es un excelente ejercicio de aprendizaje, se vuelve tedioso cuando quieres construir aplicaciones de verdad: un API REST, el backend de una tienda en línea, el servidor de tu próximo proyecto de portafolio o el microservicio que necesita tu empresa. Ahí es donde entra Express, y de eso se trata esta unidad completa.

Express es el framework web más usado de Node.js. Esa frase no es un adorno de marketing: es una realidad medible. Si entras a npm, el registro de paquetes más grande del mundo, Express lleva años entre los paquetes más descargados de todo el ecosistema, con decenas de millones de descargas cada semana. Miles de empresas y startups lo usan en producción para lanzar sus productos rápido, y prácticamente cualquier tutorial, curso, oferta de trabajo o proyecto open source que hable de backend con JavaScript lo menciona o lo exige. Cuando una vacante dice "se requiere Node.js", en la gran mayoría de los casos lo que en realidad quiere decir es "se requiere Express". Así que esta unidad no es un tema más del curso: es la puerta de entrada al Node.js profesional, al que te da trabajo.

Pero antes de instalar nada, detengámonos un momento en la palabra clave: framework. ¿Qué es un framework y por qué deberías usar uno si ya sabes hacer servidores con `http` nativo? La analogía más clara es la de construir una casa. Imagina que te piden construir una casa y te dan dos opciones. Opción uno: te dan un terreno baldío, un camión de ladrillos, cemento, arena y una pala. Todo lo puedes hacer, por supuesto: cavas los cimientos a mano, levantas pared por pared, mides cada nivel con un cordel. Opción dos: te dan el mismo terreno pero además andamios ya armados, una revolvedora de cemento, un plano aprobado, escaleras de obra y una grúa para subir los materiales. La casa final puede quedar igual de sólida con cualquiera de las dos opciones, pero con la segunda terminas en semanas en lugar de meses, cometes menos errores y puedes concentrar tu energía en lo importante: el diseño, los acabados, la funcionalidad de cada cuarto. El framework es la opción dos. No construye la

casa por ti, pero te entrega los andamios, las herramientas y la estructura para que tu trabajo sea más rápido, más ordenado y menos propenso a errores tontos.

En programación, un framework web es una caja de herramientas especializada: ya trae resuelto todo lo repetitivo y aburrido de un servidor web (enrutar peticiones, leer cuerpos, servir respuestas en JSON, manejar errores, organizar el código) y te deja concentrarte en la lógica de tu aplicación: qué datos guardas, qué reglas de negocio aplicas, qué respuestas le das a cada cliente. Y aquí viene lo interesante: hay frameworks enormes y complejos que intentan hacerlo todo por ti, con convenciones rígidas, generadores de código y una estructura de proyecto que parece un laberinto. Express tomó el camino opuesto: ser minimalista. Su filosofía se resume en una frase que vas a leer en esta unidad varias veces porque es el lema del framework: "menos código, más servidor". Express no te impone una arquitectura, no te obliga a usar una base de datos específica, no te dice cómo nombrar tus archivos. Te da un núcleo pequeño y flexible (rutas, middleware, objetos de petición y respuesta mejorados) y a partir de ahí tú construyes lo que necesites. Esa es precisamente la razón por la que es el más usado: es lo suficientemente simple para aprenderlo en una tarde y lo suficientemente potente para sostener aplicaciones en producción.

Un poco de historia, porque todo buen programador debería conocer sus herramientas. Express fue creado en 2010 por TJ Holowaychuk, un programador que en ese entonces ya era una figura muy activa en la comunidad de Node.js. La inspiración vino de Sinatra, un framework minimalista del mundo de Ruby que demostró que un framework web podía caber en unas cuantas líneas y seguir siendo útil. La idea era la misma para Node: un framework delgado, sin opiniones fuertes, que hiciera elegante lo que con http nativo era verboso. El proyecto creció rapidísimo, la comunidad aportó cientos de piezas de funcionalidad reutilizables llamadas middlewares (que verás a fondo en la unidad 9) y con el tiempo el proyecto quedó bajo el cuidado de una fundación que lo mantiene hasta hoy. Más de una década después, Express sigue siendo la versión estable más usada en el mundo del backend JavaScript. Esa longevidad en un ecosistema donde los frameworks nacen y mueren cada dos años dice muchísimo de la calidad del diseño original.

Ahora, una pregunta honesta que quizá ya te estás haciendo: si ya sé hacer un servidor con http nativo, ¿de verdad necesito Express? La respuesta honesta es: para aprender, no; para trabajar y construir cosas reales, sí. Piensa en lo que hiciste en las unidades anteriores: para leer el cuerpo de una petición POST tuviste que concatenar pedazos de datos manualmente, para saber qué ruta pidió el cliente tuviste que analizar la URL a mano, para responder un JSON tuviste que poner los encabezados correctos tú mismo. Cada uno de esos pasos es una oportunidad de cometer un error, y cuando tu aplicación crece a veinte, cincuenta o cien rutas, el código se convierte en una maraña de condicionales anidados imposible de mantener.

Express elimina todo ese ruido. Con Express, una ruta GET se escribe en una línea, leer el cuerpo de una petición es automático una vez que activas la opción correspondiente, y responder JSON es tan simple como `res.json(datos)`. El resultado es código que se lee casi como español: "cuando llegue un GET a `/tareas`, responde con la lista de tareas". Esa legibilidad no es un lujo, es una necesidad cuando trabajas en equipo o cuando regresas a tu propio código tres meses después.

En esta unidad vas a dominar las clases 21, 22 y 23 del curso. La clase 21 te presenta a Express: qué es, cómo se instala con `npm install express` y por qué es el framework web más usado de Node. La clase 22 es tu primera app Express de verdad: unas pocas líneas de código y tu servidor ya está respondiendo peticiones, algo que con http nativo te tomaba veinte o treinta líneas. La clase 23 entra al corazón de cualquier aplicación web: las rutas GET y POST, el diálogo básico entre cliente y servidor (GET trae datos, POST los envía) y cómo Express hace que ese diálogo sea trivial de programar. Pero esta unidad va mucho más allá de esas tres clases: aquí vas a entender de verdad qué es un framework, vas a desmenuzar tu primera aplicación Express línea por línea, vas a conocer a fondo los objetos `req` y `res` de Express (qué son versiones mejoradas y mucho más cómodas que las del http nativo), vas a aprender por qué `req.body` es `undefined` si olvidas activar `express.json()` (el error más común de todos los principiantes, y el que más preguntas genera en foros), vas a probar tus rutas con el navegador y con curl como un profesional, vas a organizar tus rutas en archivos separados con `express.Router` cuando tu proyecto crezca, y vas a conocer los errores típicos y sus soluciones para que nunca te quedes atorado más de cinco minutos.

Al terminar esta unidad serás capaz de hacer algo que suena simple pero que es la base de todo el desarrollo backend moderno: levantar un servidor Express, definir rutas GET que entregan datos y rutas POST que reciben datos, responder en JSON con los códigos de estado correctos y estructurar tu código para que no se convierta en un desastre cuando crezca. En la unidad 9 daremos el siguiente paso con los middlewares, que son la característica más poderosa de Express, y más adelante construiremos APIs REST completas con todo lo aprendido. Pero todo empieza aquí, con estas tres clases y esta unidad. Así que abre tu terminal, asegúrate de tener Node.js instalado, y prepárate: en menos de una hora vas a tener tu primer servidor Express respondiendo, y vas a entender cada línea de lo que escribiste. Vamos con todo.

Explicación detallada

Qué es un framework y por qué Express

Empecemos por la base, sin prisas, porque este concepto te va a acompañar toda tu carrera. Un framework es un conjunto de herramientas, funciones y convenciones que te da una estructura para construir aplicaciones. La diferencia clave entre un framework y una librería es quién manda: con una librería, tú llamas a sus funciones cuando las necesitas y tú controlas el flujo del programa; con un framework, es el framework el que controla el flujo general y tú le entregas pedazos de código que él ejecuta cuando corresponde. En Express, tú no escribes el código que escucha el puerto ni el que decide qué hacer con cada petición: Express hace eso, y tú solo le dices "cuando llegue esta ruta con este método, ejecuta esta función". Esa inversión de control es lo que hace que tu código sea más corto y más ordenado.

Volvamos a la analogía de la casa, porque vale la pena exprimirla. Construir un servidor web con http nativo es como construir a mano: cada ladrillo lo pones tú. Tienes control total, y para aprender es perfecto, porque entiendes cada detalle. Pero cuando el proyecto crece, el control total se convierte en una carga: tienes que acordarte de todo, no se te puede olvidar nada, y cualquier descuido (un encabezado mal puesto, un cuerpo mal parseado) se convierte en un bug difícil de rastrear. Express es el equipo de andamios y la revolvedora: sigue siendo tu casa, tu diseño y tus decisiones, pero el trabajo pesado y repetitivo ya está resuelto. Y ojo, esto no significa que Express sea "para flojos" o que esconder la complejidad sea malo: significa que la complejidad repetitiva se resolvió una vez, bien, por gente experta, y tú la reutilizas en lugar de reinventarla con errores nuevos cada vez.

¿Por qué Express y no otro framework? Node.js tiene varias opciones: Fastify, que presume de ser más rápido; NestJS, que es más estructurado y usa TypeScript; Koa, creado por el mismo equipo de Express como su "sucesor espiritual"; Hapi, AdonisJS, y más. Cada uno tiene sus méritos, pero Express gana por tres razones que juntas son imbatibles. Primera: es el más usado, y eso significa más tutoriales, más respuestas en foros, más middlewares listos para usar y más ofertas de trabajo que lo piden. Cuando te atores (y te vas a atorar, todos nos atoramos), la probabilidad de que alguien ya haya tenido tu mismo problema y lo haya documentado es máxima con Express. Segunda: es minimalista. No te obliga a aprender una arquitectura completa antes de escribir tu primera línea útil. En esta misma unidad vas a tener un servidor funcionando con un puñado de líneas, sin generadores de proyecto ni configuraciones crípticas. Tercera: es estable. Su API central no ha cambiado de forma radical en años, así que lo que aprendas aquí te va a servir durante mucho tiempo. No es el framework más moderno ni el más rápido en benchmarks, pero es el caballo de batalla del backend JavaScript, y dominarlo es la mejor inversión que puedes hacer ahora.

La historia breve ya la conoces de la introducción: TJ Holowaychuk lo creó en 2010 inspirado en Sinatra de Ruby, con la idea de un framework delgado y sin opiniones. Lo que vale la pena

agregar es el contexto: en 2010 Node.js apenas tenía un año de vida. Era una tecnología prometedora pero cruda, y la gente que quería construir aplicaciones web con ella tenía que pelear con http nativo para todo. Express llegó y demostró que se podía tener elegancia sin perder potencia, y su sistema de middlewares (funciones que se ejecutan en cadena para cada petición) se convirtió en el patrón que después copiarían casi todos los frameworks de Node. Entender esa historia te ayuda a entender el diseño: Express es minimalista a propósito, no por falta de funciones, sino por filosofía. Cada cosa que necesites la agregas tú con middlewares, y el núcleo se mantiene pequeño y rápido.

Instalación: `npm install express`

Vamos a lo práctico. Para usar Express necesitas un proyecto de Node.js con npm inicializado. Si vienes siguiendo el curso desde la unidad 1, esto ya lo dominas, pero repasemos el flujo completo como debe hacerse en un proyecto real.

Primero, crea una carpeta para tu proyecto y entra en ella. Luego inicializa npm, que te va a hacer unas preguntas (nombre, versión, descripción) que puedes aceptar con Enter para usar los valores por defecto:

```
mkdir mi-primer-app  
cd mi-primer-app  
npm init -y
```

El flag `-y` acepta todo por defecto y te crea el archivo `package.json` de inmediato. Ese archivo es el acta de nacimiento de tu proyecto: ahí quedan registradas las dependencias que instales.

Ahora sí, la instalación de Express:

```
npm install express
```

Eso es todo. npm descarga Express y sus dependencias internas, las guarda en la carpeta `node_modules` y anota "express" con su versión en tu `package.json`. A partir de este momento, en cualquier archivo JavaScript de tu proyecto puedes escribir `require("express")` y Node va a encontrar el paquete. Un detalle importante para tu vida profesional: Express se instala como dependencia de producción (así queda registrado por defecto), porque tu aplicación lo necesita para correr, no solo para desarrollarla. Más adelante verás herramientas que sí van como dependencias de desarrollo, como nodemon, pero Express siempre va en dependencias normales.

Verifica que todo quedó bien abriendo tu package.json: deberías ver una sección "dependencies" con algo como "express": "^4.18.2" (la versión exacta puede variar según cuándo leas esto). Si lo ves, felicidades: ya tienes el framework web más usado de Node.js instalado en tu máquina. El siguiente paso es usarlo.

Tu primera app Express, línea por línea

Este es el momento que estabas esperando. Crea un archivo llamado app.js en tu proyecto y escribe lo siguiente:

```
const express = require("express");
const app = express();

app.get("/", (req, res) => {
  res.send("Hola mundo desde Express");
});

app.listen(3000, () => {
  console.log("Servidor corriendo en http://localhost:3000");
});
```

Guárdalo, corre node app.js en tu terminal, abre tu navegador en <http://localhost:3000> y vas a ver el texto "Hola mundo desde Express". Tres ideas fundamentales y tu servidor ya responde. Con http nativo, lo equivalente te tomaba unas veinte líneas entre crear el servidor, parsear la URL, comparar rutas y escribir los encabezados. Esa diferencia, multiplicada por cada ruta de una aplicación real, es exactamente el "menos código, más servidor" del que habla la filosofía de Express.

Ahora desmenucemos cada línea, porque aquí no se vale copiar y pegar sin entender.

Línea 1: `const express = require("express");` Aquí importas el paquete Express que instalaste con npm. Lo que te devuelve `require("express")` es una función: la función creadora de aplicaciones. Por convención se guarda en una constante llamada `express`, en minúsculas. Es importante que entiendas que en este punto todavía no tienes un servidor ni una aplicación: tienes la "fábrica" que sabe crear aplicaciones. Es como tener los planos y las herramientas, pero aún no has construido nada.

Línea 2: `const app = express();` Aquí llamas a esa función y te devuelve un objeto: tu aplicación. Por convención se llama `app`. Este objeto `app` es el corazón de todo lo que vas a hacer con Express en esta unidad y en las siguientes: sobre él defines rutas, configuras opciones, activas middlewares y al final lo pones a escuchar peticiones. Piensa en `app` como

el edificio de tu restaurante: todavía está vacío, pero ya existe el local donde vas a montar las mesas (rutas), la cocina (lógica) y la puerta de entrada (el puerto).

Líneas 4 a 6: `app.get("/", (req, res) => { res.send("Hola mundo desde Express"); });` Esta es tu primera ruta. Vamos por partes. `app.get` significa "cuando llegue una petición con el método GET". El primer argumento, `"/"`, es la ruta: la raíz del servidor, es decir, lo que el navegador pide cuando entras a `http://localhost:3000` sin nada más. El segundo argumento es una función, el manejador de la ruta (en inglés le dicen handler o route handler): Express la va a ejecutar automáticamente cada vez que alguien pida GET `/`. Esa función recibe dos parámetros por convención llamados `req` y `res`: `req` es la petición (request) con toda la información de lo que pidió el cliente, y `res` es la respuesta (response) con la que le contestas. Dentro del manejador, `res.send("Hola mundo desde Express")` envía ese texto como respuesta al cliente. Express se encarga de poner los encabezados correctos (como `Content-Type` y `Content-Length`) por ti. Cuando terminas de leer esta unidad vas a conocer a fondo todo lo que `req` y `res` pueden hacer; por ahora quédate con la idea: `req` es lo que llega, `res` es lo que respondes.

Líneas 8 a 10: `app.listen(3000, () => { console.log("Servidor corriendo en http://localhost:3000"); });` Esta línea prende el servidor. `app.listen` le dice a tu aplicación: "empieza a escuchar peticiones en el puerto 3000". El segundo argumento es una función que se ejecuta cuando el servidor ya quedó prendido y escuchando; por eso ahí ponemos el `console.log` que te confirma que todo salió bien. Si corres el archivo y ves ese mensaje en la terminal, tu servidor está vivo. El puerto 3000 es una convención muy usada en desarrollo para Node.js, pero podría ser cualquier número libre: 4000, 5000, 8080. Lo importante es que el puerto que pongas en `listen` sea el mismo que abras en el navegador.

Fíjate en algo hermoso de este código: no hay ni un solo `if`, no hay parseo manual de URLs, no hay manejo manual de encabezados. Express se encarga de todo lo mecánico y tu código solo expresa la intención: "en la ruta raíz, responde este texto; escucha en el puerto 3000". Esa es la magia de un buen framework: tu código se vuelve una descripción de lo que quieres, no una lista de instrucciones mecánicas para lograrlo.

El objeto app: tus herramientas principales

El objeto `app` que creas con `express()` tiene varios métodos que vas a usar todos los días. Ya viste dos (`get` y `listen`); conozcamos la familia completa de los esenciales.

`app.get(ruta, manejador)` define una ruta que responde a peticiones GET. GET es el método que usa el navegador cuando escribes una URL en la barra de direcciones o haces clic en un enlace: sirve para pedir datos, para leer, para mostrar. En la clase 23 lo veremos a fondo, pero adelanto la regla de oro: GET es para traer información del servidor al cliente.

`app.post(ruta, manejador)` define una ruta que responde a peticiones POST. POST es el método que se usa para enviar datos al servidor: un formulario de registro, un nuevo comentario, una tarea nueva en tu lista. La regla de oro complementaria: POST es para llevar información del cliente al servidor, normalmente para crear algo nuevo.

`app.put(ruta, manejador)` y `app.delete(ruta, manejador)` completan el cuarteto básico. PUT se usa para actualizar un recurso existente (editar una tarea), DELETE para eliminarlo (borrar una tarea). Estos cuatro métodos (GET, POST, PUT, DELETE) son la base de las APIs REST, que es el estilo de arquitectura más común para backends modernos. En esta unidad nos concentramos en GET y POST, que son las clases 22 y 23 del curso, pero quiero que desde ya conozcas los cuatro nombres porque los verás en cada proyecto profesional.

`app.listen(puerto, callback)` prende el servidor en el puerto indicado. Puedes guardar lo que devuelve en una variable si necesitas apagar el servidor programáticamente más adelante: `const servidor = app.listen(3000);` y luego `servidor.close()`. En desarrollo casi nunca lo necesitas porque apagas con Ctrl+C en la terminal, pero en pruebas automatizadas es muy útil.

`app.set(nombre, valor)` guarda configuraciones de tu aplicación. El uso más común que verás en tutoriales es `app.set("port", 3000)` para guardar el puerto, o configuraciones del motor de plantillas cuando aprendas a generar HTML dinámico. También existe `app.get(nombre)` con un solo argumento, que en lugar de definir una ruta lee una configuración guardada con set. Ojo con esta doble personalidad de `app.get`: con dos argumentos define una ruta GET; con un solo argumento lee una configuración. Al principio confunde, pero con la práctica se vuelve natural: si el primer argumento es una ruta que empieza con "/", es una ruta; si es un nombre de configuración, es una lectura.

Además, `app.use(manejador)` registra middlewares, funciones que se ejecutan para todas las peticiones o para un grupo de rutas antes de llegar al manejador final. Los middlewares son tan importantes que tienen su propia unidad (la 9), así que aquí solo los menciono para que el mapa quede completo: `express.json()`, que vas a usar muchísimo en esta unidad para leer cuerpos JSON, técnicamente se activa con `app.use(express.json())`. Ya lo verás en acción muy pronto.

Los objetos req y res de Express: tu mesa de trabajo

Cada vez que un cliente hace una petición a tu servidor, Express crea dos objetos y se los entrega a tu manejador de ruta: req (la petición) y res (la respuesta). Ya los conoces del http nativo, pero los de Express son versiones mejoradas: tienen métodos y propiedades extra que te ahorran muchísimo trabajo. Vamos a conocerlos cómo se debe, porque el noventa por ciento de tu código Express va a ser manipular estos dos objetos.

Empecemos con req, que representa todo lo que el cliente te mandó. Sus propiedades más útiles son:

req.body contiene el cuerpo de la petición, es decir, los datos que el cliente envió en un POST o PUT. Aquí viene el detalle crucial que causa más dolores de cabeza a los principiantes: por defecto, req.body es undefined. Express no lee el cuerpo automáticamente; tienes que activar el middleware `express.json()` con `app.use(express.json())` para que Express lea los cuerpos en formato JSON y los convierta en un objeto JavaScript utilizable. En la siguiente subsección lo explicamos a fondo, pero grábate la regla desde ya: si req.body te da undefined, en el 99 por ciento de los casos es porque olvidaste poner `app.use(express.json())`.

req.params contiene los parámetros de ruta: partes variables de la URL que definiste con dos puntos. Por ejemplo, si defines `app.get("/tareas/:id", ...)`, y el cliente pide `/tareas/42`, entonces req.params.id vale "42". Esto sirve para pedir un recurso específico: la tarea 42, el usuario 7, el producto 15. Es como el número de mesa en un restaurante: la ruta te dice la zona (las tareas) y el parámetro te dice exactamente cuál (la 42).

req.query contiene los query strings: los pares clave=valor que van después del signo de interrogación en la URL. Si el cliente pide `/buscar?q=express&limite=10`, entonces req.query.q vale "express" y req.query.limite vale "10". Los query strings sirven para filtros, búsquedas, paginación y opciones: son la forma de decirle al servidor "tráeme esto, pero con estas condiciones". A diferencia de req.params, que identifican un recurso específico, req.query modifica cómo quieres ver una lista de recursos.

req.method te dice el método HTTP de la petición ("GET", "POST", etc.), req.path te da la ruta pedida, y req.headers contiene los encabezados que mandó el cliente. No los vas a usar todos los días, pero es bueno saber que existen.

Ahora res, que es tu herramienta para contestar. Sus métodos más importantes:

res.send(datos) envía una respuesta y termina la petición. Es el método más versátil: si le pasas un string, lo envía como texto (o HTML si parece HTML); si le pasas un objeto o un arreglo, automáticamente lo convierte a JSON. Es el "comodín" que sirve para casi todo en las primeras etapas.

`res.json(datos)` envía una respuesta específicamente en formato JSON. Convierte el objeto o arreglo que le pases a JSON y pone el encabezado `Content-Type: application/json` automáticamente. Para APIs, `res.json` es el estándar: deja claro tu intención de responder datos estructurados.

`res.status(código)` establece el código de estado HTTP de la respuesta. Casi siempre se encadena antes de `send` o `json`: `res.status(404).json({ error: "No encontrado" })`. Los códigos de estado son el lenguaje con el que tu servidor le dice al cliente qué pasó con su petición, y los veremos en detalle en su propia subsección.

`res.redirect(ruta)` redirige al cliente a otra ruta: le responde con un código 302 y la nueva ubicación, y el navegador va automáticamente hacia allá. Sirve, por ejemplo, después de procesar un formulario: en lugar de mostrar una página de "gracias" en la ruta del formulario, rediriges a una página de confirmación.

`res.sendFile(rutaArchivo)` envía un archivo completo como respuesta, útil para servir páginas HTML estáticas o descargas. Lo verás en acción en los ejemplos paso a paso cuando sirvamos un formulario HTML.

Una regla de oro con `res`: cada petición debe recibir exactamente una respuesta. Si tu manejador termina sin llamar a ningún método de `res`, el cliente se queda esperando hasta que la conexión muera por `timeout`. Y si llamas dos veces (por ejemplo, un `res.send` y luego otro `res.json` por un error de lógica en tus condicionales), Express te va a lanzar un error de "headers already sent" (encabezados ya enviados). La práctica profesional es usar `return` al responder dentro de condicionales: `return res.status(404).json({ error: "No existe" });` para garantizar que la función se detiene ahí.

res.send vs res.json: ¿cuál usar?

Esta es una duda clásica y la respuesta es más simple de lo que parece. `res.send` es flexible: acepta strings, objetos, arreglos y buffers, y decide el formato por ti. `res.json` es específico: siempre responde JSON, siempre pone el encabezado correcto. En la práctica: si estás construyendo un API que va a consumir una app móvil, un frontend en React o cualquier otro cliente programático, usa `res.json` para que tu intención quede clara y el formato sea siempre consistente. Si estás respondiendo texto simple o HTML rápido en ejemplos y prototipos, `res.send` es perfecto. Muchos programadores usan `res.json` para todo lo que sea datos y `res.send` para mensajes de texto simples, y es una convención sana que te recomiendo adoptar desde ahora. Lo que nunca debes hacer es armar el JSON a mano con strings concatenados: para eso existe `res.json`, que además maneja correctamente caracteres especiales y acentos.

Cómo Express lee el cuerpo: `express.json()` y `express.urlencoded()`

Llegamos al tema que más preguntas genera en foros y más frustración causa a principiantes, así que vamos a explicarlo sin prisa y con todo detalle. Cuando un cliente hace una petición POST con datos (por ejemplo, un formulario o un JSON desde una app), esos datos viajan en el cuerpo de la petición. Con http nativo aprendiste a leerlos a mano: escuchando eventos "data", concatenando pedazos y luego parseando el resultado. Express puede hacer todo eso por ti, pero por diseño minimalista no lo hace automáticamente: tú decides qué tipos de cuerpo quieres aceptar, activando los middlewares correspondientes.

El más importante es `express.json()`. Cuando escribes `app.use(express.json())` en tu aplicación, le estás diciendo a Express: "para cada petición que llegue, si trae un cuerpo en formato JSON (encabezado Content-Type: application/json), léelo completo, conviértelo en objeto JavaScript y ponlo en `req.body`". A partir de esa línea, en tus rutas POST puedes hacer `const { nombre, precio } = req.body;` directamente, sin concatenar nada a mano. Es una de esas líneas mágicas que escribes una vez al inicio de tu app y te olvidas para siempre.

El segundo es `express.urlencoded({ extended: true })`. Este sirve para leer los cuerpos que mandan los formularios HTML tradicionales. Cuando un formulario HTML se envía con `method="post"`, el navegador no manda JSON: manda los datos en un formato llamado `urlencoded` (algo como `nombre=Juan&edad=30`). Si tu ruta POST va a recibir formularios HTML directos (sin JavaScript de por medio), necesitas `app.use(express.urlencoded({ extended: true }))` para que Express lea ese formato y también lo ponga en `req.body`. La opción `extended: true` permite objetos anidados en los datos; en la práctica casi todos los tutoriales la ponen en `true` y tú también deberías.

¿Por qué Express no activa esto por defecto? Por filosofía minimalista y por seguridad: leer cuerpos consume memoria y procesamiento, y un servidor que no necesita leer cuerpos (por ejemplo, uno que solo sirve rutas GET) no debería pagar ese costo. Además, obligarte a declararlo hace explícito qué formatos acepta tu aplicación. Es una decisión de diseño consciente, no un olvido.

El error clásico, repito porque vale oro: defines tu ruta POST, mandas datos con JSON, haces `console.log(req.body)` y ves `undefined`. No es que el cliente no haya mandado los datos, no es que la ruta esté mal: es que olvidaste `app.use(express.json())` antes de tus rutas. El orden importa: los middlewares se ejecutan en el orden en que los registras, así que las líneas `app.use(express.json())` y `app.use(express.urlencoded({ extended: true }))` deben ir antes de tus definiciones de rutas, justo después de crear `app`. Cuando algo falle con `req.body`, tu checklist de diagnóstico es: uno, ¿puse `app.use(express.json())`?; dos, ¿lo puse antes de las rutas?; tres, ¿el cliente está mandando el encabezado `Content-Type: application/json`? Con esas tres preguntas resuelves el 99 por ciento de los casos.

Rutas GET y POST: el diálogo cliente-servidor

Ahora sí, el corazón de la clase 23. Toda la web funciona como un diálogo entre dos partes: el cliente (navegador, app móvil, otro servidor) que pide, y el servidor (tu app Express) que responde. Los métodos HTTP son los "verbos" de ese diálogo, y GET y POST son los dos verbos fundamentales. La analogía más clara es la de un restaurante: GET es cuando le preguntas al mesero "¿qué hay en el menú?" o "¿me traes la carta de postres?": estás pidiendo información, no estás cambiando nada del restaurante. POST es cuando le dices "quiero ordenar dos tacos al pastor": estás enviando información que provoca un cambio (la cocina empieza a trabajar, tu cuenta aumenta). Esa es la diferencia esencial: GET trae datos, POST los envía, y POST normalmente crea o modifica algo en el servidor.

Vamos a verlo con código real. Imagina una pequeña aplicación de tareas guardadas en memoria (un arreglo, sin base de datos todavía):

```

const express = require("express");
const app = express();
app.use(express.json());

let tareas = [
  { id: 1, texto: "Aprender Express", hecha: false },
  { id: 2, texto: "Hacer mi primera app", hecha: false }
];

app.get("/tareas", (req, res) => {
  res.json(tareas);
});

app.post("/tareas", (req, res) => {
  const { texto } = req.body;
  const nuevaTarea = { id: tareas.length + 1, texto, hecha: false };
  tareas.push(nuevaTarea);
  res.status(201).json(nuevaTarea);
});

app.listen(3000, () => {
  console.log("Servidor corriendo en http://localhost:3000");
});

```

Analicemos el diálogo. La ruta GET /tareas no recibe datos del cliente: simplemente toma el arreglo tareas y lo devuelve con res.json. Es el mesero trayendo el menú: "aquí está todo lo que hay". La ruta POST /tareas sí recibe datos: extrae texto de req.body (que existe gracias a express.json()), crea un objeto de tarea nueva con un id, lo guarda en el arreglo con push y responde con la tarea creada. Es el cliente ordenando y el mesero confirmando: "anotado, tu orden es esta".

Fíjate en un detalle importante: ambas rutas usan la misma ruta "/tareas", y eso está perfectamente bien. Express distingue las rutas por la combinación de método más ruta: GET /tareas y POST /tareas son dos rutas distintas que conviven sin problema. Esto no es casualidad: es el diseño REST, donde la ruta nombra el recurso (las tareas) y el método dice qué quieres hacer con él (verlas o crear una). Cuando en unidades futuras construyas APIs REST completas, este patrón de "misma ruta, distinto método" va a estar en todas partes.

Otro detalle: en el POST usamos `res.status(201).json(nuevaTarea)`. El 201 significa "Created" (creado): es el código de estado correcto cuando una petición POST crea un recurso nuevo. Responder la tarea creada (con su id asignado) es una buena práctica porque el cliente normalmente necesita saber el id para futuras operaciones. Son pequeños detalles que separan un backend aficionado de uno profesional.

Códigos de estado: el lenguaje de las respuestas

Los códigos de estado HTTP son números de tres dígitos con los que tu servidor le comunica al cliente el resultado de su petición. No son decoración: los clientes programáticos (apps móviles, frontends, otros servidores) toman decisiones basadas en estos códigos. Los esenciales que debes dominar desde ya:

200 OK: todo salió bien. Es el código por defecto de `res.send` y `res.json` cuando no especificas otro. Úsalo para GETs exitosos y operaciones que salieron bien sin crear nada nuevo.

201 Created: la petición creó un recurso nuevo. Úsalo en tus POST cuando guardes algo: `res.status(201).json(nuevoRecurso)`. Le dice al cliente "no solo salió bien, sino que nació algo nuevo".

400 Bad Request: la petición venía mal formada o le faltan datos. Úsalo cuando validas la entrada: si en el POST de tareas no mandaron el texto, responde `res.status(400).json({ error: "El campo texto es obligatorio" })`. Es tu forma de decir "oye, me mandaste la orden incompleta".

404 Not Found: el recurso pedido no existe. Úsalo cuando buscan una tarea con un id que no está: `res.status(404).json({ error: "Tarea no encontrada" })`. También es lo que Express responde automáticamente cuando ninguna ruta coincide con la petición.

500 Internal Server Error: algo falló en tu servidor de forma inesperada. Normalmente no lo pones tú a mano para cada caso, sino que lo usas en manejadores de errores generales. Si tu código lanza una excepción no controlada, Express responde 500 por defecto.

La regla práctica: 2xx significa éxito (200 bien, 201 creado), 4xx significa que el cliente hizo algo mal (400 datos mal, 404 no existe) y 5xx significa que el servidor falló. Responder siempre con 200 aunque haya un error es un mal hábito muy común en principiantes: técnicamente "funciona", pero rompe el contrato con los clientes y hace imposible manejar errores correctamente del otro lado. Acostúmbrate desde ahora a elegir el código correcto; tu yo del futuro (y tus compañeros de equipo) te lo van a agradecer.

Probando tus rutas: navegador y curl

Escribir rutas sin probarlas es como cocinar sin probar la comida: una mala idea. Tienes dos herramientas básicas, y un profesional usa ambas según el caso.

El navegador sirve para probar rutas GET: escribes `http://localhost:3000/tareas` en la barra de direcciones y ves la respuesta JSON directamente. Es rápido y visual. Pero el navegador solo sabe hacer GET (y formularios POST, que veremos en los ejemplos). Para probar un POST con JSON necesitas otra herramienta.

Ahí entra curl, un programa de línea de comandos que viene instalado en la mayoría de los sistemas y que te permite hacer peticiones HTTP con control total. Para probar tu POST de tareas, abrirías otra terminal (dejando el servidor corriendo en la primera) y escribirías:

```
curl -X POST http://localhost:3000/tareas -H "Content-Type: application/json" -d '{"texto": "Probar con curl"}'
```

Vamos por partes: `-X POST` dice que el método es POST; `-H "Content-Type: application/json"` pone el encabezado que le dice a Express "esto es JSON" (recuerda que `express.json()` solo procesa cuerpos con ese encabezado); `-d` manda los datos del cuerpo. Si todo sale bien, verás en la terminal la tarea creada en JSON con su id. Y si luego haces `curl http://localhost:3000/tareas` (GET simple), verás tu nueva tarea en la lista. Acabas de hacer el ciclo completo: crear con POST, leer con GET, como lo haría cualquier cliente real.

Un truco profesional: agrega `console.log(req.body)` temporalmente en tu ruta POST mientras desarrollas, para ver exactamente qué está llegando. Y si usas Windows y curl te da problemas con las comillas, la alternativa es probar con la extensión de tu editor o con herramientas gráficas, pero curl es el estándar que todo backend developer debe conocer.

Organizando rutas: `express.Router`

Cuando tu aplicación tiene cinco rutas, tenerlas todas en `app.js` está bien. Cuando tiene cincuenta, ese archivo se convierte en un monstruo ilegible. La solución de Express se llama Router: un mini-enrutador que te permite definir rutas en archivos separados y luego conectarlos a tu aplicación principal. Piensa en tu app como un centro comercial: `app.js` es el edificio, y cada Router es una tienda con su propia organización interna.

Así se ve en la práctica. Creas un archivo `rutas/tareas.js`:

```

const express = require("express");
const router = express.Router();

let tareas = [];

router.get("/", (req, res) => {
  res.json(tareas);
});

router.post("/", (req, res) => {
  const { texto } = req.body;
  const nueva = { id: tareas.length + 1, texto, hecha: false };
  tareas.push(nueva);
  res.status(201).json(nueva);
});

module.exports = router;

```

Y en tu app.js principal lo conectas:

```

const express = require("express");
const app = express();
app.use(express.json());

const rutasTareas = require("./rutas/tareas");
app.use("/tareas", rutasTareas);

app.listen(3000, () => {
  console.log("Servidor corriendo en http://localhost:3000");
});

```

Fíjate en la elegancia: dentro del router, las rutas se definen como "/" porque el prefijo "/tareas" ya lo pone `app.use("/tareas", rutasTareas)` al montarlo. El resultado final es idéntico (GET /tareas y POST /tareas funcionan igual), pero el código quedó organizado por recurso, cada archivo con su responsabilidad. Cuando agregues usuarios, productos o pedidos, cada uno tendrá su propio archivo de rutas. Esta es la estructura que verás en prácticamente todos los proyectos Express profesionales, así que acostúmbrate a ella desde ahora: un archivo de rutas por recurso, montados en `app.js`.

Errores comunes y sus soluciones

Cierro la explicación con los tropiezos clásicos, porque conocerlos de antemano te ahorra horas de frustración. Todos los programadores pasamos por cada uno de estos; la diferencia entre un principiante y un profesional no es no cometerlos, sino diagnosticarlos en minutos.

Error 1: `req.body` es undefined. Causa: olvidaste `app.use(express.json())`. Solución: agrégalo justo después de crear app y antes de las rutas. Diagnóstico rápido: si el `console.log` del cuerpo muestra undefined pero curl sí está mandando datos, es este error el 99 por ciento de las veces.

Error 2: puerto ocupado (EADDRINUSE). Causa: ya hay otro servidor corriendo en el mismo puerto, típicamente porque olvidaste apagar una ejecución anterior de tu app o porque otro programa usa ese puerto. El mensaje en la terminal dice algo como "Error: listen EADDRINUSE: address already in use :::3000". Solución: apaga el otro proceso (Ctrl+C en su terminal) o cambia temporalmente de puerto. En desarrollo, este error casi siempre significa "ya tienes tu propio servidor corriendo en otra terminal".

Error 3: la ruta no coincide y Express responde 404 con un HTML de "Cannot GET /ruta". Causa: escribiste mal la ruta en el navegador o en el código (mayúsculas, slashes de más o de menos), o definiste la ruta con un método distinto al que estás probando. Solución: revisa letra por letra que la ruta del cliente coincida con la del servidor, incluyendo el método HTTP. Express distingue entre /tarejas y /Tarejas, y entre GET y POST sobre la misma ruta.

Error 4: "Cannot read properties of undefined" al hacer `req.body.texto`. Causa: es el error 1 disfrazado; intentas leer una propiedad de algo que es undefined. Solución: la misma, activa `express.json()`. Con el tiempo, este mensaje de error te va a hacer sonreír porque lo diagnosticarás en segundos.

Error 5: olvidar responder en algún camino del código. Causa: tienes un if que responde pero el else no, o un return temprano que salta la respuesta. El síntoma es que el cliente (navegador o curl) se queda esperando eternamente. Solución: revisa que todos los caminos posibles de tu manejador terminen en algún método de res. El hábito del `return res.json(...)` ayuda muchísimo.

Error 6: mandar el Content-Type equivocado desde el cliente. Causa: haces POST con curl o con código pero sin el encabezado `Content-Type: application/json`; Express recibe el cuerpo pero `express.json()` lo ignora porque no lo reconoce como JSON, y `req.body` queda como un objeto vacío o undefined. Solución: siempre incluye el encabezado correcto cuando mandes JSON.

Con estos seis errores dominados, ya tienes el kit de supervivencia para no atorarte más de unos minutos. En la siguiente sección los vas a ver en acción con ejemplos completos que puedes correr en tu máquina.

Ejemplos paso a paso

La teoría sin práctica se olvida en una semana. En esta sección vas a construir cuatro ejemplos completos, de menor a mayor complejidad, que puedes copiar, correr y modificar en tu máquina. Te recomiendo que no solo los leas: escríbelos tú mismo, ejecútalos, rómpelos a propósito y arrégloslos. Cada ejemplo incluye el código completo, la explicación de lo que hace y cómo probarlo.

Ejemplo 1: Hola mundo con Express

El clásico de todos los clásicos, pero esta vez entendiendo cada línea (ya lo desmenuzamos en la explicación, así que aquí va directo y con un par de mejoras).

Crea una carpeta proyecto1, entra en ella, corre `npm init -y` y `npm install express`. Luego crea `app.js`:

```
const express = require("express");
const app = express();
const PUERTO = 3000;

app.get("/", (req, res) => {
  res.send("Hola mundo desde Express");
});

app.get("/saludo", (req, res) => {
  res.send("Hola, bienvenido a mi primer servidor Express");
});

app.listen(PUERTO, () => {
  console.log("Servidor corriendo en http://localhost:" + PUERTO);
});
```

Corre con `node app.js`. Abre `http://localhost:3000` y verás el primer mensaje; abre `http://localhost:3000/saludo` y verás el segundo. Prueba también `http://localhost:3000/ruta-que-no-existe` y observa el 404 automático de Express con el mensaje "Cannot GET /ruta-que-no-existe".

¿Qué aprendiste aquí? Que definir rutas es declarar pares de método más ruta con su manejador, que puedes tener tantas rutas GET como quieras, y que Express maneja gratis el caso de rutas inexistentes. Nota el detalle de guardar el puerto en la constante PUERTO: es una buena práctica desde el día uno, porque cuando quieras cambiar de puerto lo haces en un solo lugar. En proyectos profesionales verás `const PUERTO = process.env.PORT || 3000;`, que usa el puerto que indique el entorno (útil al publicar tu app en un servidor) o 3000 por defecto. No necesitas entender `process.env` a fondo todavía; solo quédate con el patrón.

Ejercicio mental antes de seguir: ¿qué pasaría si agregas una tercera ruta `app.get("/", ...)` duplicada? Express usa la primera que coincida y nunca llega a la segunda. El orden de definición sí importa cuando hay coincidencias: la primera que empató, gana.

Ejemplo 2: Mini blog con GET (datos en memoria)

Ahora algo más parecido a una aplicación real: un mini blog donde los posts viven en un arreglo en memoria y se sirven con rutas GET. Vas a practicar `res.json`, `req.params` y `req.query` en un solo ejemplo.

En una carpeta `proyecto2` con Express instalado, crea `app.js`:

```

const express = require("express");
const app = express();

const posts = [
  { id: 1, titulo: "Mi primer post", contenido: "Hola, este es mi primer post con Express.", autor: "Lucas" },
  { id: 2, titulo: "Rutas GET", contenido: "GET sirve para pedir datos al servidor.", autor: "Lucas" },
  { id: 3, titulo: "JSON por todos lados", contenido: "res.json convierte objetos en respuestas JSON.", autor: "Ana" }
];

app.get("/posts", (req, res) => {
  const { autor } = req.query;
  if (autor) {
    const filtrados = posts.filter((p) => p.autor === autor);
    return res.json(filtrados);
  }
  res.json(posts);
});

app.get("/posts/:id", (req, res) => {
  const id = Number(req.params.id);
  const post = posts.find((p) => p.id === id);
  if (!post) {
    return res.status(404).json({ error: "Post no encontrado" });
  }
  res.json(post);
});

app.listen(3000, () => {
  console.log("Blog corriendo en http://localhost:3000");
});

```

Prueba estas URLs en tu navegador y observa cada resultado:

<http://localhost:3000/posts> te devuelve los tres posts en JSON. Es la lista completa, el "menú" del blog.

<http://localhost:3000/posts?autor=Ana> te devuelve solo el post de Ana. Aquí `req.query.autor` vale "Ana" y el filtro se activa. Prueba con `?autor=Lucas` y con `?autor=AlguienQueNoExiste` (debe devolver un arreglo vacío, lo cual es correcto: la petición estuvo bien formada, simplemente no hay coincidencias).

`http://localhost:3000/posts/2` te devuelve solo el post con id 2, gracias a `req.params.id`. Prueba con `/posts/99` y verás el 404 con el mensaje de error en JSON.

Hay varias lecciones profesionales escondidas en este ejemplo. Primera: la ruta `/posts/:id` debe definirse con cuidado respecto a `/posts`; Express las distingue sin problema porque una tiene el parámetro y la otra no, pero si tuvieras `/posts/nuevo` como ruta fija y `/posts/:id` como variable, el orden importaría (la fija primero). Segunda: `Number(req.params.id)` convierte el parámetro (que siempre llega como string) a número para compararlo con los ids numéricos; olvidar esa conversión es un bug clásico porque `"2" !== 2` en comparación estricta y el find nunca encontraría nada. Tercera: el `return` antes de `res.status(404).json` garantiza que la función se detiene y no intenta responder dos veces. Cuarta: devolver un arreglo vacío cuando un filtro no coincide es correcto y distinto de un 404; el 404 se reserva para cuando piden un recurso específico que no existe.

Ejemplo 3: Formulario HTML que hace POST y se guarda

Hasta ahora el cliente ha sido el navegador pidiendo datos o curl mandando JSON. En este ejemplo el cliente es un formulario HTML real, como los que llenas todos los días en la web, y el servidor guarda lo que recibe. Vas a usar `express.urlencoded()`, `res.sendFile` y `res.redirect`, y verás el ciclo completo de una interacción web clásica.

Crea la carpeta `proyecto3` con Express instalado. Dentro, crea una subcarpeta `public` y en ella un archivo `formulario.html`:

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  <title>Registro de tareas</title>
</head>
<body>
  <h1>Agrega una tarea</h1>
  <form action="/tareas" method="post">
    <label>Texto de la tarea:
      <input type="text" name="texto" required>
    </label>
    <button type="submit">Guardar</button>
  </form>
  <p><a href="/tareas">Ver todas las tareas</a></p>
</body>
</html>
```


Fíjate en los atributos del form: `action="/tareas"` dice a qué ruta se envían los datos, y `method="post"` dice con qué método HTTP. El input tiene `name="texto"`: ese nombre es la clave con la que el dato viajará al servidor.

Ahora el servidor, `app.js` en la raíz de `proyecto3`:

```
const express = require("express");
const path = require("path");
const app = express();

app.use(express.urlencoded({ extended: true }));

let tareas = [];
let siguienteId = 1;

app.get("/", (req, res) => {
  res.sendFile(path.join(__dirname, "public", "formulario.html"));
});

app.get("/tareas", (req, res) => {
  res.json(tareas);
});

app.post("/tareas", (req, res) => {
  const { texto } = req.body;
  if (!texto || texto.trim() === "") {
    return res.status(400).send("El texto de la tarea es obligatorio. <a href='/'>Volver</a>");
  }
  const nueva = { id: siguienteId++, texto: texto.trim(), hecha: false };
  tareas.push(nueva);
  res.redirect("/tareas");
});

app.listen(3000, () => {
  console.log("Servidor corriendo en http://localhost:3000");
});
```

Corre el servidor y abre `http://localhost:3000`. Verás el formulario. Escribe una tarea, dale a Guardar, y observa qué pasa: el navegador te lleva a `/tareas` mostrando el JSON con tu tarea guardada. Agrega dos o tres más y verás la lista crecer. Luego dale al enlace "Ver todas las tareas" o entra directo a `/tareas`.

Analicemos las piezas nuevas. `app.use(express.urlencoded({ extended: true }))` es la línea que permite leer los datos del formulario: sin ella, `req.body` estaría vacío porque los formularios HTML no mandan JSON sino formato `urlencoded`. `res.sendFile(path.join(__dirname, "public", "formulario.html"))` sirve el archivo HTML cuando piden la raíz; `path.join` construye la ruta del archivo de forma segura en cualquier sistema operativo, y `__dirname` es la carpeta donde vive tu `app.js`. En el POST, validamos que el texto no venga vacío y respondemos 400 si el cliente mandó la orden incompleta (el atributo `required` del input ya lo evita en el navegador, pero un cliente malicioso podría saltárselo, así que el servidor siempre debe validar). Finalmente, `res.redirect("/tareass")` en lugar de responder algo directamente: después de procesar el formulario, mandamos al cliente a la lista de tareas. Este patrón se llama POST-Redirect-GET y evita que al recargar la página se reenvíe el formulario duplicando la tarea. Es un detalle profesional que verás en todas las aplicaciones web clásicas.

Prueba romperlo a propósito para aprender: quita la línea de `express.urlencoded` y envía el formulario; verás que la tarea se guarda con texto `undefined` o que la validación del 400 se dispara. Vuelve a poner la línea y todo funciona. Esa es la lección del middleware grabada a fuego.

Ejemplo 4: Router separado para `/api/productos`

El último ejemplo te enseña a organizar un proyecto como lo hacen los profesionales: rutas separadas por recurso usando `express.Router`, con el prefijo `/api` que es convención para indicar "esto es un API de datos, no páginas web".

Estructura de carpetas de proyecto4:

```
proyecto4/  
  app.js  
  rutas/  
    productos.js  
  package.json
```

El archivo `rutas/productos.js` contiene toda la lógica de productos:

```

const express = require("express");
const router = express.Router();

let productos = [
  { id: 1, nombre: "Laptop", precio: 15000 },
  { id: 2, nombre: "Mouse", precio: 350 }
];
let siguienteId = 3;

router.get("/", (req, res) => {
  res.json(productos);
});

router.get("/:id", (req, res) => {
  const producto = productos.find((p) => p.id === Number(req.params.id));
  if (!producto) {
    return res.status(404).json({ error: "Producto no encontrado" });
  }
  res.json(producto);
});

router.post("/", (req, res) => {
  const { nombre, precio } = req.body;
  if (!nombre || precio === undefined) {
    return res.status(400).json({ error: "nombre y precio son obligatorios" });
  }
  const nuevo = { id: siguienteId++, nombre, precio: Number(precio) };
  productos.push(nuevo);
  res.status(201).json(nuevo);
});

module.exports = router;

```

Y app.js queda limpio y legible, como el índice de un libro:

```

const express = require("express");
const app = express();

app.use(express.json());

const rutasProductos = require("./rutas/productos");
app.use("/api/productos", rutasProductos);

app.get("/", (req, res) => {
  res.send("API de la tienda. Prueba /api/productos");
});

app.listen(3000, () => {
  console.log("API corriendo en http://localhost:3000");
});

```

Pruébalo con curl. Lista productos: `curl http://localhost:3000/api/productos`. Ver uno: `curl http://localhost:3000/api/productos/1`. Crear uno: `curl -X POST http://localhost:3000/api/productos -H "Content-Type: application/json" -d '{"nombre": "Teclado", "precio": 800}'`. Verifica que el nuevo producto aparece en la lista.

Lo poderoso de este ejemplo es lo que no se ve a simple vista: `app.js` no sabe nada de productos; solo sabe que existe un router montado en `/api/productos`. Si mañana agregas usuarios, creas `rutas/usuarios.js` con su propio router y agregas dos líneas en `app.js`: el `require` y el `app.use("/api/usuarios", rutasUsuarios)`. Cada recurso vive en su archivo, cada archivo se puede probar y mantener por separado, y `app.js` sigue siendo un índice limpio. Esta es la semilla de la arquitectura que usan las aplicaciones Express reales, y cuando en unidades futuras agregues base de datos y middlewares de autenticación, esta organización te va a salvar la vida.

Un detalle final: observa que dentro del router las rutas se escriben como `"/` y `"/:id"`, sin el prefijo `/api/productos`. El prefijo lo aporta `app.use` al montar el router. Esto significa que el router es reutilizable: podrías montarlo en `/api/productos` hoy y en `/v2/productos` mañana sin tocar el archivo del router. Esa separación entre "qué rutas define" y "dónde se montan" es una de las ideas más elegantes de Express.

Práctica

Leer sobre Express te da el mapa; escribir código te da el territorio. Estos cinco ejercicios están ordenados de menor a mayor dificultad y cubren todo lo esencial de la unidad: rutas GET, parámetros, query strings, POST con JSON, formularios HTML y routers. Intenta resolver cada uno por tu cuenta antes de mirar la solución: la lucha es donde ocurre el aprendizaje. Cada ejercicio incluye la solución explicada paso a paso para que puedas comparar y corregir.

Ejercicio 1: Servidor "acerca de mí"

Crea un servidor Express que responda en el puerto 3000 con tres rutas GET: la raíz "/" debe responder el texto "Bienvenido a mi servidor"; "/nombre" debe responder tu nombre en texto; "/perfil" debe responder un objeto JSON con tu nombre, tu edad y tu ciudad. Usa `res.send` para los textos y `res.json` para el objeto.

Solución paso a paso. Primero, la estructura base que ya dominas: importar express, crear app, definir rutas, escuchar en el puerto. La única decisión es qué método de respuesta usar en cada ruta. Para "/" y "/nombre" el requerimiento pide texto, así que `res.send` es lo natural. Para "/perfil" pide un objeto JSON, así que `res.json` deja clara la intención y pone el encabezado correcto automáticamente.

```
const express = require("express");
const app = express();

app.get("/", (req, res) => {
  res.send("Bienvenido a mi servidor");
});

app.get("/nombre", (req, res) => {
  res.send("Lucas");
});

app.get("/perfil", (req, res) => {
  res.json({ nombre: "Lucas", edad: 30, ciudad: "Ciudad de México" });
});

app.listen(3000, () => {
  console.log("Servidor corriendo en http://localhost:3000");
});
```

Prueba las tres rutas en el navegador. En `"/perfil"` observa cómo el navegador muestra el JSON con formato. Lección del ejercicio: elegir entre `res.send` y `res.json` según lo que quieras comunicar, y que un mismo servidor puede mezclar respuestas de texto y JSON sin problema.

Ejercicio 2: API de películas con parámetros y filtros

Crea un arreglo en memoria con al menos 4 películas (cada una con `id`, `título` y `generó`). Implementa `GET /peliculas` que devuelva todas, pero que acepte un query string opcional `generó` para filtrar (por ejemplo `/peliculas?generó=acción`). Implementa `GET /peliculas/:id` que devuelva una sola película, o un 404 en JSON si el `id` no existe. Recuerda convertir el parámetro a número.

Solución paso a paso. Primero defines los datos. Luego la ruta de lista: lees `req.query.generó`; si viene, filtras con `filter` comparando el género; si no viene, devuelves todo. Usa `return res.json(filtradas)` dentro del `if` para detener la función. Después la ruta de detalle: conviertes `req.params.id` con `Number`, buscas con `find`, y si no hay coincidencia respondes 404 con un objeto de error en JSON; si sí hay, la devuelves con `res.json`.

```

const express = require("express");
const app = express();

const peliculas = [
  { id: 1, titulo: "Misión imposible", genero: "accion" },
  { id: 2, titulo: "Rápidos y furiosos", genero: "accion" },
  { id: 3, titulo: "El diario de una pasión", genero: "romance" },
  { id: 4, titulo: "Interestelar", genero: "ciencia" }
];

app.get("/peliculas", (req, res) => {
  const { genero } = req.query;
  if (genero) {
    const filtradas = peliculas.filter((p) => p.genero === genero);
    return res.json(filtradas);
  }
  res.json(peliculas);
});

app.get("/peliculas/:id", (req, res) => {
  const pelicula = peliculas.find((p) => p.id === Number(req.params.id));
  if (!pelicula) {
    return res.status(404).json({ error: "Película no encontrada" });
  }
  res.json(pelicula);
});

app.listen(3000, () => {
  console.log("API de películas en http://localhost:3000");
});

```

Prueba: /peliculas (las 4), /peliculas?genero=acción (2), /peliculas?genero=terror (arreglo vacío, correcto), /peliculas/3 (una), /peliculas/99 (404). Lección del ejercicio: req.query para filtros opcionales, req.params para identificar un recurso, y la diferencia entre "lista vacía" (200) y "recurso inexistente" (404).

Ejercicio 3: Registro de usuarios con POST y validación

Implementa un POST /usuarios que reciba JSON con nombre y email, los valide (ambos obligatorios, el email debe contener "@"), guarde el usuario en un arreglo en memoria con un id autoincremental, y responda 201 con el usuario creado. Si la validación falla, responde 400 con un mensaje de error en JSON. Además, implementa GET /usuarios que devuelva la lista. No olvides activar express.json().

Solución paso a paso. El primer paso, y el que más se olvida, es `app.use(express.json())` antes de las rutas; sin él, `req.body` será `undefined` y todo lo demás fallará. Luego defines el arreglo y el contador de ids. En el POST, extraes nombre y email de `req.body` con desestructuración. Validas: si falta alguno o el email no contiene "@", respondes 400 con `return` para detener la ejecución. Si pasa la validación, creas el objeto con el id actual, incrementas el contador, lo guardas y respondes 201 con el objeto creado. El GET es una línea.

```
const express = require("express");
const app = express();

app.use(express.json());

let usuarios = [];
let siguienteId = 1;

app.get("/usuarios", (req, res) => {
  res.json(usuarios);
});

app.post("/usuarios", (req, res) => {
  const { nombre, email } = req.body;
  if (!nombre || !email) {
    return res.status(400).json({ error: "nombre y email son obligatorios" });
  }
  if (!email.includes("@")) {
    return res.status(400).json({ error: "El email no es válido" });
  }
  const nuevo = { id: siguienteId++, nombre, email };
  usuarios.push(nuevo);
  res.status(201).json(nuevo);
});

app.listen(3000, () => {
  console.log("Servidor corriendo en http://localhost:3000");
});
```

Pruébalo con curl. Caso feliz: `curl -X POST http://localhost:3000/usuarios -H "Content-Type: application/json" -d '{"nombre": "Ana", "email": "ana@correo.com"}'` debe devolver el usuario con id 1 y código 201. Caso de error: manda solo el nombre o un email sin "@" y verifica el 400 con el mensaje. Luego haz GET /usuarios y confirma que el usuario quedó guardado. Lección del ejercicio: el servidor nunca confía en el cliente, siempre valida; y cada resultado tiene su código de estado correcto (201 creado, 400 datos mal).

Ejercicio 4: Formulario de contacto con redirect

Crea un servidor que sirva en "/" un formulario HTML (guardado en un archivo) con campos nombre y mensaje, que se envíe por POST a "/contacto". El servidor debe leer los datos con `express.urlencoded()`, guardarlos en un arreglo en memoria, y redirigir a "/gracias", donde se muestra un mensaje de agradecimiento en HTML. Además, GET "/mensajes" debe mostrar en JSON todos los mensajes recibidos (útil para verificar que se guardaron).

Solución paso a paso. Primero el HTML (`contacto.html`): un form con `action="/contacto"` y `method="post"`, dos inputs con `name="nombre"` y `name="mensaje"`. En el servidor: activas `express.urlencoded({ extended: true })` al inicio (los formularios no mandan JSON). La ruta GET "/" sirve el HTML con `res.sendFile`. El POST "/contacto" extrae nombre y mensaje de `req.body`, valida que no vengan vacíos (400 si fallan), guarda el objeto con fecha, y hace `res.redirect("/gracias")`. La ruta GET "/gracias" responde un HTML simple de agradecimiento, y GET "/mensajes" devuelve el arreglo en JSON.

```

const express = require("express");
const path = require("path");
const app = express();

app.use(express.urlencoded({ extended: true }));

let mensajes = [];

app.get("/", (req, res) => {
  res.sendFile(path.join(__dirname, "contacto.html"));
});

app.post("/contacto", (req, res) => {
  const { nombre, mensaje } = req.body;
  if (!nombre || !mensaje || nombre.trim() === "" || mensaje.trim() === "") {
    return res.status(400).send("Nombre y mensaje son obligatorios. <a href='/'>Volver</a>");
  }
  mensajes.push({ nombre: nombre.trim(), mensaje: mensaje.trim(), fecha: new Date().toISOString() });
  res.redirect("/gracias");
});

app.get("/gracias", (req, res) => {
  res.send("<h1>Gracias por tu mensaje</h1><p><a href='/'>Enviar otro</a></p>");
});

app.get("/mensajes", (req, res) => {
  res.json(mensajes);
});

app.listen(3000, () => {
  console.log("Servidor corriendo en http://localhost:3000");
});

```

Pruébalo en el navegador: llena el formulario, envíalo y observa cómo terminas en /gracias; luego visita /mensajes y verifica que tu mensaje quedó guardado. Prueba enviar el formulario con un campo vacío (quítale temporalmente el required al HTML si lo pusiste) y observa el 400. Lección del ejercicio: el patrón POST-Redirect-GET completo, la diferencia entre urlencoded y json según el cliente, y servir archivos HTML con sendFile.

Ejercicio 5: Router de libros para /api/libros

Organiza una API de libros usando `express.Router` en un archivo separado. El router debe implementar: `GET "/"` (lista todos), `GET "/:id"` (uno por id o 404), `POST "/"` (recibe título y autor en JSON, valida ambos, responde 201). El archivo principal `app.js` debe activar `express.json()`, montar el router en `"/api/libros"` y tener una ruta raíz de bienvenida. Pruébalo todo con `curl`.

Solución paso a paso. Creas la carpeta `rutas/` y el archivo `rutas/libros.js`: importas `express`, creas el router, defines los datos en memoria y las tres rutas con la lógica ya practicada (filtros no, pero sí `find` con `Number`, validación con 400, 201 al crear). Exportas con `module.exports`. En `app.js`: importas `express`, activas `express.json()`, importas el router con `require("../rutas/libros")`, lo montas con `app.use("/api/libros", rutasLibros)`, agregas la bienvenida y el `listen`. La clave conceptual: dentro del router las rutas son `"/"` y `"/:id"` porque el prefijo lo pone el montaje.

```

const express = require("express");
const router = express.Router();

let libros = [
  { id: 1, titulo: "Cien años de soledad", autor: "García Márquez" },
  { id: 2, titulo: "La casa de los espíritus", autor: "Isabel Allende" }
];
let siguienteId = 3;

router.get("/", (req, res) => {
  res.json(libros);
});

router.get("/:id", (req, res) => {
  const libro = libros.find((l) => l.id === Number(req.params.id));
  if (!libro) {
    return res.status(404).json({ error: "Libro no encontrado" });
  }
  res.json(libro);
});

router.post("/", (req, res) => {
  const { titulo, autor } = req.body;
  if (!titulo || !autor) {
    return res.status(400).json({ error: "titulo y autor son obligatorios" });
  }
  const nuevo = { id: siguienteId++, titulo, autor };
  libros.push(nuevo);
  res.status(201).json(nuevo);
});

module.exports = router;

```

```
const express = require("express");
const app = express();

app.use(express.json());

const rutasLibros = require("./rutas/libros");
app.use("/api/libros", rutasLibros);

app.get("/", (req, res) => {
  res.send("API de libros. Prueba /api/libros");
});

app.listen(3000, () => {
  console.log("API corriendo en http://localhost:3000");
});
```

Verificación con curl: GET lista, GET /api/libros/1, GET /api/libros/99 (404), POST válido (201 y el libro aparece en la lista), POST sin autor (400). Lección del ejercicio: la organización profesional por recursos con Router, reutilizable y escalable; es el patrón que usarás en todos tus proyectos de aquí en adelante.

Resumen de la unidad

Llegaste al final de la unidad 8, y si hiciste los ejemplos y los ejercicios, ya no eres la misma persona que empezó: ahora sabes construir servidores web con el framework más usado de Node.js. Hagamos el recuento de lo conquistado, porque es mucho.

Primero, entiendes qué es un framework y por qué existe: no es magia ni es para flojos, es la decisión profesional de no reinventar lo repetitivo. Con la analogía de la casa (construir a mano contra construir con andamios y revolvedora) te quedó claro que Express te da la estructura para concentrarte en la lógica de tu aplicación en lugar de pelear con detalles mecánicos. Conoces su historia (TJ Holowaychuk, 2010, inspiración en Sinatra de Ruby), su filosofía minimalista resumida en "menos código, más servidor", y por qué es el más usado: comunidad gigante, curva de aprendizaje suave y estabilidad probada durante más de una década.

Segundo, dominas la instalación y tu primera aplicación: `npm init`, `npm install express`, y el flujo `require`, crear `app`, definir rutas, escuchar en un puerto. Desmenuzaste cada línea de tu primera `app` hasta entender qué hace `require("express")`, qué es el objeto `app` y cómo `app.listen` prende el servidor. Conoces los métodos esenciales de `app`: `get`, `post`, `put`, `delete` para los cuatro verbos básicos; `listen` para prender el servidor; `set` para configuraciones; y `use` para `middlewares` (que será la estrella de la unidad 9).

Tercero, trabajas con soltura los objetos `req` y `res` de Express: `req.body` para los datos enviados (y por qué es `undefined` sin `express.json()`), `req.params` para parámetros de ruta como `/tareas/:id`, `req.query` para filtros como `?autor=Ana`, y del lado de la respuesta `res.send` para texto y usos generales, `res.json` para datos estructurados, `res.status` para el código correcto, `res.redirect` para redirecciones y `res.sendFile` para servir archivos. Sabes la diferencia entre `res.send` y `res.json` y cuándo usar cada uno, y tienes grabada la regla de oro: cada petición recibe exactamente una respuesta.

Cuarto, entiendes el diálogo cliente-servidor en su forma más pura: `GET` trae datos (como pedir el menú), `POST` los envía (cómo hacer el pedido), y ambos pueden convivir sobre la misma ruta porque Express distingue por método más ruta. Manejas los códigos de estado como un profesional: 200 para éxito, 201 para recurso creado, 400 para datos mal enviados por el cliente, 404 para recurso inexistente y 500 para errores del servidor. Y sabes probar tus rutas cómo se debe: el navegador para los `GET`, `curl` para los `POST` con `JSON` y encabezados.

Quinto, organizas tu código como un profesional con `express.Router`: un archivo de rutas por recurso, montados en `app.js` con `app.use("/prefijo", router)`, con las rutas internas escritas de forma relativa y reutilizable. Y tienes tu kit de supervivencia contra los errores clásicos: `req.body` `undefined` (falta `express.json()`), puerto ocupado (`EADDRINUSE`), rutas que no coinciden (404 de Express), propiedades de `undefined`, respuestas olvidadas y `Content-Type` incorrecto del cliente.

El mapa hacia adelante: en la unidad 9 vas a conocer los `middlewares`, la característica más poderosa de Express, que te permitirá agregar funcionalidad reutilizable (como el registro de peticiones, la autenticación o el manejo centralizado de errores) a toda tu aplicación con unas pocas líneas. Después construirás APIs REST completas combinando todo lo aprendido aquí. Pero el cimiento ya está puesto, y es sólido: levantar un servidor Express, definir rutas `GET` y `POST`, responder `JSON` con los códigos correctos y organizar el código para que crezca sin romperse. Eso, que hoy te parece natural, es exactamente lo que hacen todos los días miles de desarrolladores backend en producción. Bienvenido al club.

Glosario rápido de la unidad

Para cerrar, un repaso relámpago de los términos que debes dominar al dedillo antes de pasar a la unidad 9. Si alguno no te suena con total claridad, regresa a su sección y repásalo: estos conceptos son el vocabulario diario del backend con Express.

Framework: estructura de herramientas y convenciones donde el framework controla el flujo general y tú aportas el código específico. Express es un framework minimalista: te da lo esencial (rutas, middleware, req/res mejorados) sin imponerte arquitectura.

Middleware: función que se ejecuta en cadena para cada petición antes de llegar al manejador final. En esta unidad usaste dos sin profundizar: `express.json()` y `express.urlencoded()`. La unidad 9 está dedicada por completo a ellos.

Ruta (route): combinación de un método HTTP y un camino URL, con su función manejadora. `GET /tareas` y `POST /tareas` son dos rutas distintas aunque compartan el camino.

Manejador (handler): la función que Express ejecuta cuando una petición coincide con una ruta. Recibe req y res, y su trabajo es terminar respondiendo exactamente una vez.

`req.body`: el cuerpo de la petición ya leído y convertido en objeto. Solo existe si activaste el middleware correspondiente (`express.json()` para JSON, `express.urlencoded()` para formularios).

`req.params`: parámetros variables de la ruta definidos con dos puntos, como `:id` en `/tareas/:id`. Siempre llegan como texto.

`req.query`: los pares clave=valor después del `?` en la URL, usados para filtros, búsquedas y opciones.

`res.json()`: responde con datos en formato JSON y el encabezado correcto. El estándar para APIs.

`res.status()`: fija el código de estado HTTP. Se encadena: `res.status(201).json(nuevo)`.

Códigos de estado: 200 éxito, 201 creado, 400 petición mal formada, 404 no encontrado, 500 error del servidor.

Router: mini-enrutador de Express para organizar rutas por recurso en archivos separados, montados con `app.use("/prefijo", router)`.

POST-Redirect-GET: patrón donde tras procesar un POST se redirige con `res.redirect` a una página GET, evitando reenvíos duplicados al recargar.

REST: estilo de arquitectura donde la ruta nombra el recurso y el método HTTP dice la acción (GET leer, POST crear, PUT actualizar, DELETE eliminar). Lo practicaste sin darte cuenta en cada ejemplo de esta unidad, y será la base de tus APIs profesionales.

UNIDAD 9: Rutas avanzadas, middleware y archivos estáticos

En las unidades anteriores aprendiste a crear rutas básicas con Express, a leer el cuerpo de las peticiones y a conectar tu servidor con una base de datos o con datos en memoria. Hasta ahora tus rutas son fijas: `"/usuarios"`, `"/productos"`, `"/contacto"`. Cada una responde siempre lo mismo, sin importar quién la pida. Pero una API real no funciona así. Piensa en YouTube: la dirección para ver un video es algo como `"/watch?v=dQw4w9WgXcQ"`. Ese código cambia para cada video, y sin embargo es la misma página la que se encarga de mostrarlos todos. Piensa en Amazon: cuando buscas "audífonos" la dirección cambia a `"/s?k=audifonos"`. La página es la misma, pero el contenido depende de lo que escribiste en el buscador. En esta unidad tu servidor va a aprender esos trucos: rutas que capturan datos dentro de la dirección, query strings que mandan información extra, middleware que se interpone entre la petición y la respuesta para revisar, registrar y proteger, y la capacidad de servir archivos estáticos para que tu frontend viva en el mismo servidor que tu backend.

Esta unidad es un punto de inflexión. Después de terminarla, tu servidor dejará de ser un juguete de ejemplos y empezará a parecerse a una aplicación de verdad: una API que entiende direcciones con datos, que tiene guardianes en la puerta, que registra todo lo que pasa y que además sirve la página web que la consume. Son tres clases de contenido condensadas en una sola unidad porque los tres temas se necesitan entre sí: los parámetros y las query strings hacen tus rutas inteligentes, los middleware las hacen seguras y ordenadas, y los archivos estáticos convierten tu servidor en un producto completo.

Antes de empezar, asegúrate de tener instalado Express y de saber crear un servidor básico con `app.listen`, `app.get` y `app.post`. Si algo de eso te suena borroso, repasa las unidades 7 y 8. Aquí vamos a construir sobre esa base, sin repetirla.

Explicación detallada

Parámetros de ruta: direcciones que llevan datos

Hasta ahora, cada ruta que definiste es una dirección fija. Cuando el navegador pide `"/usuarios"`, tu servidor responde la lista completa. Pero ¿qué pasa cuando quieres el usuario número 7? No vas a crear una ruta para cada usuario posible: `"/usuarios/1"`, `"/usuarios/2"`, `"/usuarios/3"`... hasta el infinito. Eso es imposible. La solución son los parámetros de ruta: partes de la dirección que actúan como comodines y capturan lo que el cliente escribió ahí.

En Express, un parámetro se marca con dos puntos. Mira:

```
const express = require("express");
const app = express();

app.get("/usuarios/:id", (req, res) => {
  const id = req.params.id;
  res.send("Pediste el usuario con id: " + id);
});

app.listen(3000, () => {
  console.log("Servidor corriendo en el puerto 3000");
});
```

Si alguien visita `/usuarios/7`, el valor de `req.params.id` será `"7"`. Si visita `/usuarios/42`, será `"42"`. La misma ruta atiende a todos los usuarios posibles porque `:id` es un hueco que se rellena con lo que venga en la dirección. Express guarda esos valores capturados en el objeto `req.params`, que es un objeto normal de JavaScript: `req.params.id`, `req.params.nombre`, lo que hayas nombrado.

Ojo con un detalle importante: `req.params.id` siempre llega como texto, aunque escribas un número en la dirección. `/usuarios/7` te entrega la cadena `"7"`, no el número `7`. Si vas a compararlo con un id numérico de tu base de datos o de tu arreglo, necesitas convertirlo: `Number(req.params.id)` o `parseInt(req.params.id)`. Este es uno de los errores más comunes y más frustrantes, porque `"7" == 7` funciona con el comparador suave pero `"7" === 7` falla con el estricto, y si usas `find` con comparación estricta tu usuario "no existe" aunque sí exista.

Múltiples parámetros en una sola ruta

Una ruta puede tener más de un parámetro. Por ejemplo, en un blog podrías querer la dirección `/blog/2026/09/mi-artículo`, donde el año, el mes y el nombre del artículo son datos:

```
app.get("/blog/:anio/:mes/:articulo", (req, res) => {
  const { anio, mes, articulo } = req.params;
  res.send("Artículo: " + articulo + ", publicado en " + mes + "/" + anio);
});
```

Visitar `/blog/2026/09/mi-artículo` entrega `anio="2026"`, `mes="09"`, `artículo="mi-artículo"`. También puedes mezclar partes fijas con parámetros: `/tienda/:categoría/:productoid` podría responder a `/tienda/electrónica/99`. Los parámetros se leen en el orden en que aparecen en la ruta, y cada uno tiene su nombre.

Parámetros opcionales y validación

A veces un parámetro puede no venir. Express te permite marcar un parámetro como opcional agregando un signo de interrogación al final del nombre: `/artículos/:categoría?`. Si alguien visita `/artículos`, el parámetro `categoría` será `undefined`; si visita `/artículos/deportes`, será `"deportes"`. Úsalo con moderación, porque demasiada opcionalidad vuelve confusas las rutas.

Más importante que la opcionalidad es la validación. Nunca confíes en lo que viene en la dirección. El cliente puede escribir lo que sea: `/usuarios/abc`, `/usuarios/-5`, `/usuarios/../../../../etc/passwd`. Tu código debe defenderse. Un patrón común es validar al inicio del manejador:

```
app.get("/usuarios/:id", (req, res) => {
  const id = Number(req.params.id);
  if (!Number.isInteger(id) || id <= 0) {
    return res.status(400).send("El id debe ser un número entero positivo");
  }
  const usuario = usuarios.find((u) => u.id === id);
  if (!usuario) {
    return res.status(404).send("Usuario no encontrado");
  }
  res.json(usuario);
});
```

Fíjate en los dos `"return"` antes de cada `res.send`. Cuando ya respondiste con un error, no quieres que el código siga ejecutándose y trate de responder dos veces, porque Express se queja si intentas mandar dos respuestas a la misma petición. El patrón `"valida y salte temprano"` te mantiene a salvo.

Query strings: las instrucciones extra de la dirección

Los parámetros de ruta son parte obligatoria de la dirección: `/usuarios/7` no funciona sin el 7. Las query strings son otra cosa: son pares clave=valor que van después de un signo de interrogación al final de la dirección, y son opcionales. Ejemplo: `/buscar?q=react&limite=10`. Aquí `"q=react"` y `"limite=10"` son las query strings. La primera empieza con `"?"`, las siguientes se separan con `"&"`.

Express las lee automáticamente y te las entrega en `req.query`, otro objeto normal de JavaScript:

```
app.get("/buscar", (req, res) => {
  const termino = req.query.q;
  const limite = Number(req.query.limite) || 10;
  res.send("Buscando: " + termino + ", límite: " + limite);
});
```

Visitar `/buscar?q=react` da `terminó="react"` y `limite=10` (el valor por defecto). Visitar `/buscar?q=react&limite=5` da `limite=5`. Si no mandas `"q"`, `terminó` será `undefined`, y tu código debe manejar ese caso.

Piensa en la diferencia con una analogía. La dirección de tu casa es como un parámetro de ruta: es parte esencial de cómo llegar, sin ella no hay destino. Las query strings son como las instrucciones extra que le das al repartidor: "toca dos veces", "deja el paquete con el vecino". No cambian a dónde va, cambian cómo se atiende la entrega. En una API, `/usuarios/7` (parámetro) dice QUÉ recurso quieres: el usuario 7. `/usuarios?rol=admin&activos=true` (query) dice CÓMO lo quieres: filtrado, ordenado, paginado. El recurso es el mismo (`"usuarios"`), las instrucciones cambian la respuesta.

Query strings en la vida real: filtros, búsquedas y paginación

Las query strings brillan en tres usos: búsquedas, filtros y paginación. Una búsqueda: `/productos?buscar=laptop` devuelve solo los productos que coincidan. Un filtro: `/productos?categoría=electrónica&precioMax=5000` devuelve la electrónica de menos de 5000. Y la paginación es el uso más importante de todos: cuando tienes 10,000 productos no los mandas todos de golpe, los mandas de 10 en 10. La convención es usar `"página"` y `"limite"`: `/productos?página=2&limite=10` pide la página 2, de 10 productos cada una.

La lógica de paginación con un arreglo es sencilla una vez que la entiendes:

```
app.get("/productos", (req, res) => {
  const pagina = Number(req.query.pagina) || 1;
  const limite = Number(req.query.limite) || 10;
  const inicio = (pagina - 1) * limite;
  const fin = inicio + limite;
  const resultado = productos.slice(inicio, fin);
  res.json({
    pagina: pagina,
    limite: limite,
    total: productos.length,
    datos: resultado,
  });
});
```

Con `página=2` y `limite=10`: $\text{inicio} = (2-1)*10 = 10$, $\text{fin} = 20$, y `slice(10, 20)` devuelve los elementos del 10 al 19, que son exactamente los 10 productos de la segunda página. Memoriza esa fórmula: $\text{inicio} = (\text{página} - 1) * \text{limite}$. La vas a usar en cada API que pagues en tu vida.

Un detalle fino: las query strings también pueden repetirse o venir como arreglos si usas corchetes, como `"?tag=js&tag=node"`, que Express interpreta como `tag: ["js", "node"]`. No es lo más común, pero conviene saber que existe cuando diseñes filtros múltiples.

Middleware: los retenes en la carretera

Ahora viene el concepto más poderoso de Express. Hasta aquí, cada petición entra por una ruta y sale por su manejador, directo, sin escalas. El middleware cambia eso: son funciones que se ejecutan EN MEDIO, entre que llega la petición y que se manda la respuesta. Imagina una carretera con retenes: antes de llegar a tu destino (la ruta), pasas por varios puntos de revisión. En el primero te anotan en una lista (un logger registra que pasaste). En el segundo revisan tu identificación (un autenticador verifica quién eres). En el tercero revisan que tu carga esté en regla (un validador checa los datos). Si todo está bien, llegas a tu destino. Si algo falla en un retén, te regresan ahí mismo y nunca llegas a la ruta.

Otra analogía que funciona muy bien: el middleware es como la banda que revisa tu boleto antes de entrar al concierto. Tú quieres ver a tu banda favorita (la ruta `/concierto`). Pero antes de entrar, hay personal que revisa tu boleto, te pone un sello en la mano y a veces revisa tu mochila. Ellos no son el concierto, pero sin ellos no entras. Y lo importante: la misma banda de revisión atiende a TODAS las entradas del estadio, no solo a la puerta norte. Así es el middleware: lógica compartida que se aplica a muchas rutas desde un solo lugar.

En código, un middleware es una función que recibe tres argumentos: req, res y next.

```
app.use((req, res, next) => {  
  console.log("Petición recibida: " + req.method + " " + req.url);  
  next();  
});
```

Ese `app.use` registra el middleware para TODAS las rutas y TODOS los métodos. Cada petición que llegue al servidor pasará primero por esa función, imprimirá el mensaje, y luego `next()` le dice a Express: "ya terminé, pásale la petición al siguiente en la fila". Ese "siguiente" puede ser otro middleware o finalmente el manejador de la ruta.

`next()`: la palabra mágica que no debes olvidar

`next` es una función que Express te entrega, y llamarla es lo que permite que la petición siga avanzando. Si tu middleware NO llama a `next()` y TAMPOCO manda una respuesta, la petición se queda colgada para siempre. El navegador se queda esperando, girando la ruedita, hasta que se rinde con un timeout. Es como el retén de la carretera donde el oficial se queda viendo tu identificación sin decir nada y sin dejarte pasar ni regresarte: ahí te quedas, parado, para siempre.

Hay exactamente dos salidas válidas para un middleware: llamar a `next()` para continuar, o mandar una respuesta (`res.send`, `res.json`, `res.status(...).send`) para terminar ahí. Nunca las dos, nunca ninguna. Si llamas a `next()` después de haber mandado una respuesta, Express intentará seguir y probablemente truene con el famoso error "Cannot set headers after they are sent". Si no haces ninguna de las dos, la petición se cuelga. Grábate esta regla: todo middleware termina con `next()` o con una respuesta.

Fíjate que los middleware que ya usaste en unidades anteriores, como `express.json()`, son middleware de verdad: `app.use(express.json())` registra una función que lee el cuerpo de la petición, lo convierte a objeto, lo pone en `req.body` y luego llama a `next()` para que tu ruta lo use. Ahora entiendes qué estaba pasando por debajo todo este tiempo.

El orden de los middleware sí importa (y mucho)

Express procesa los middleware y las rutas en el orden en que los registraste, de arriba hacia abajo, como una fila. Esto significa que un middleware solo afecta a lo que está DEBAJO de él en el código. Mira este ejemplo:

```
app.use(middlewareA);      // se ejecuta para todo lo de abajo
app.get("/uno", manejadorUno);
app.use(middlewareB);      // solo se ejecuta para lo de abajo
app.get("/dos", manejadorDos);
```

Una petición a "/uno" pasa por middlewareA y luego por manejadorUno. Una petición a "/dos" pasa por middlewareA, luego middlewareB, y luego manejadorDos. middlewareB nunca toca a "/uno" porque está registrado después. Esta es la causa número uno de "mi middleware no se ejecuta": lo registraste después de las rutas que querías proteger.

La regla de oro: registra tus middleware globales (logger, seguridad, lectura del body) AL PRINCIPIO, antes de todas las rutas. Y registra el manejador de errores y la ruta 404 AL FINAL, después de todas las rutas, porque solo deben activarse si ninguna ruta respondió.

app.use vs app.get: ¿cuál uso para un middleware?

app.use monta un middleware para una ruta base y todos los métodos HTTP: app.use("/api", miMiddleware) se ejecuta para GET, POST, PUT, DELETE en cualquier dirección que empiece con "/api". En cambio app.get("/ruta", miMiddleware, manejador) monta el middleware solo para peticiones GET a esa ruta exacta. La diferencia práctica:

- app.use: "para todo lo que pase por aquí, sin importar el método". Ideal para loggers, seguridad, lectura del body.
- app.get / app.post con middleware como segundo argumento: "solo para esta ruta y este método". Ideal para validaciones específicas de una ruta.

Puedes encadenar varios middleware en una sola ruta pasándolos como argumentos separados:

```
app.get("/privado", verificarSesion, verificarPermisos, (req, res) => {
  res.send("Contenido secreto");
});
```

La petición pasa por verificarSesion, luego verificarPermisos, y solo si ambos llaman a next() llega al manejador final. Es la fila de retenes aplicada a una sola puerta.

También existe el middleware a nivel de router, que verás cuando organices rutas grandes con express.Router(): router.use(...) aplica el middleware solo a las rutas de ese router. Es el mismo concepto, con alcance más chico. No lo necesitas todavía, pero cuando tu archivo de rutas crezca y lo dividas en módulos, ahí estará.

Middleware caseros: el logger que todo lo ve

El middleware más útil que vas a escribir en tu vida es un logger: una función que registra cada petición que llega al servidor. Cuando algo falla en producción, el log es tu caja negra: te dice qué pidieron, cuándo y cómo. Aquí va uno casero, sin dependencias:

```
function logger(req, res, next) {
  const fecha = new Date().toISOString();
  console.log "[" + fecha + "]" + req.method + " " + req.url);
  next();
}

app.use(logger);
```

Cada petición imprime algo como "[2026-09-19T14:32:10.123Z] GET /usuarios/7". Con eso ya puedes reconstruir qué pasó. Si quieres algo más elegante, mide también cuánto tardó la respuesta. El truco es escuchar el evento "finish" de la respuesta:

```
function loggerConTiempo(req, res, next) {
  const inicio = Date.now();
  res.on("finish", () => {
    const ms = Date.now() - inicio;
    console.log(req.method + " " + req.url + " -> " + res.statusCode + " (" + ms +
"ms)");
  });
  next();
}
```

Ojo: el console.log va DENTRO del evento finish, porque ahí ya se mandó la respuesta y conoces el código de estado. Si lo pusieras después de next() directamente, se ejecutaría antes de que la ruta responda y el statusCode todavía no sería el final. Este patrón de "hacer algo cuando termine la respuesta" lo verás mucho en middleware profesionales.

Middleware caseros: el validador de datos

Otro middleware casero clásico es el validador: revisa que los datos que mandó el cliente tengan sentido ANTES de que lleguen a la ruta. Por ejemplo, para crear un usuario necesitas nombre y correo:


```
function validarUsuario(req, res, next) {
  const { nombre, correo } = req.body;
  if (!nombre || !correo) {
    return res.status(400).json({ error: "Faltan nombre o correo" });
  }
  if (!correo.includes("@")) {
    return res.status(400).json({ error: "El correo no es válido" });
  }
  next(); // todo bien, que siga
}

app.post("/usuarios", validarUsuario, (req, res) => {
  // aquí ya sabes que nombre y correo existen y el correo tiene @
  const nuevo = { id: siguienteId++, nombre: req.body.nombre, correo: req.body.correo };
  usuarios.push(nuevo);
  res.status(201).json(nuevo);
});
```

La belleza de esto es la reutilización: si tienes tres rutas que crean usuarios (registro normal, registro por invitación, importación masiva), las tres usan validarUsuario y la lógica vive en un solo lugar. Si mañana cambias las reglas (por ejemplo, el nombre debe tener mínimo 3 letras), cambias UNA función, no tres rutas. Ese es todo el chiste del middleware: escribir la lógica transversal una vez y aplicarla donde se necesite.

Middleware caseros: el portero de autenticación

El tercer middleware casero esencial es el "portero": decide quién pasa y quién no. En una versión simple, revisa que la petición traiga una clave en los headers:

```
function portero(req, res, next) {
  const clave = req.headers["x-clave"];
  if (clave !== "secreto123") {
    return res.status(401).json({ error: "No tienes permiso, falta la clave" });
  }
  next();
}

app.get("/admin", portero, (req, res) => {
  res.send("Bienvenido al panel de administración");
});
```

Si la petición no trae el header "x-clave" con el valor correcto, el portero responde 401 y la ruta "/admin" jamás se ejecuta. Es el retén que te regresa a tu casa. En aplicaciones reales la clave sería un token firmado (JWT, que verás en otra unidad), pero el mecanismo es idéntico: un middleware que revisa credenciales y decide si llama a next() o responde con error.

Un portero más refinado no solo deja pasar o no, sino que ANOTA quién pasó. Si la clave es válida, puede buscar al usuario y colgarlo en req para que las rutas lo usen:

```
function portero(req, res, next) {
  const clave = req.headers["x-clave"];
  const usuario = usuarios.find((u) => u.clave === clave);
  if (!usuario) {
    return res.status(401).json({ error: "Clave inválida" });
  }
  req.usuario = usuario; // la ruta sabrá quién es
  next();
}

app.get("/perfil", portero, (req, res) => {
  res.json({ mensaje: "Hola " + req.usuario.nombre });
});
```

Este patrón de "enriquecer el req" es poderosísimo: los middleware pueden agregar datos al objeto de la petición y todo lo que venga después los puede leer. express.json() hace exactamente eso con req.body.

Middleware de terceros: no reinventes la rueda

Todo lo anterior lo puedes escribir tú, pero la comunidad de Node.js ya resolvió los casos comunes con paquetes probados por millones de servidores. Los instalas con npm y los montas con app.use.

morgan es el logger profesional. En vez de tu console.log casero, te da formatos listos y colores:

```
const morgan = require("morgan");
app.use(morgan("dev"));
```

Con el formato "dev" cada petición imprime algo como "GET /usuarios/7 200 3.214 ms". El "200" es el código de estado y "3.214 ms" el tiempo. Existen otros formatos ("combined", "tiny") para producción. Si tu logger casero te enseñó el concepto, morgan te da la herramienta de todos los días.

cors resuelve un problema que te va a morder en cuanto tu frontend y tu backend vivan en lugares distintos. Los navegadores bloquean por seguridad las peticiones de una página hacia un servidor de otro origen (distinto dominio, puerto o protocolo). Si tu página está en "http://localhost:5500" y tu API en "http://localhost:3000", el navegador bloquea las peticiones aunque tu servidor responda bien. El paquete cors agrega los headers que le dicen al navegador "está permitido":

```
const cors = require("cors");
app.use(cors()); // permite peticiones desde cualquier origen
```

En desarrollo, cors() abierto está bien. En producción lo configuras para permitir solo tus dominios: cors({ origin: "https://miapp.com" }). Si algún día tu fetch funciona en Postman pero no en el navegador, el culpable casi seguro es CORS, y ahora sabes dónde mirar.

helmet es un paquete que configura automáticamente varios headers de seguridad en tus respuestas: protege contra ataques comunes como clickjacking, inyección de scripts en ciertos encabezados y adivinación de tecnología. Se usa en una línea:

```
const helmet = require("helmet");
app.use(helmet());
```

No necesitas entender cada header que pone; basta saber que es la práctica estándar ponerlo en todo servidor Express que vaya a internet. Es como ponerle seguro a las puertas: no piensas en cada cerradura, solo activas el sistema.

Manejo centralizado de errores: el hospital al final de la fila

En una aplicación real las cosas fallan: la base de datos se cae, un archivo no existe, un cálculo divide entre cero. Si cada ruta maneja sus errores por su cuenta, terminas con try/catch repetidos por todos lados y respuestas de error inconsistentes. Express ofrece una mejor salida: el middleware de manejo de errores, una función especial que se reconoce porque recibe CUATRO parámetros en vez de tres: err, req, res, next.

```
app.use((err, req, res, next) => {
  console.error("Error atrapado:", err.message);
  res.status(500).json({ error: "Algo salió mal en el servidor" });
});
```

¿Cómo llega un error hasta ahí? Desde cualquier ruta o middleware, llamas a next(err) pasándole el error:

```
app.get("/datos", (req, res, next) => {
  try {
    const resultado = operacionRiesgosa();
    res.json(resultado);
  } catch (err) {
    next(err); // mándalo al manejador central
  }
});
```

Cuando Express ve que llamaste a next con un argumento, se salta todos los middleware y rutas normales y va directo al primer middleware de error que encuentre. Por eso el manejador de errores se registra AL FINAL de todo, después de las rutas: es el hospital al final de la carretera, el último recurso.

En Express 5 (y con un pequeño truco en Express 4), si tu manejador es async y la promesa se rechaza, el error también llega al middleware de errores. Pero en Express 4 clásico, un async que truena sin try/catch deja la petición colgada. La práctica segura: envuelve tus rutas async en try/catch y manda el error con next(err). Es un poco verboso, pero te salva de los cuelgues misteriosos.

El manejador central también es donde decides qué mostrarle al cliente. En desarrollo puedes mandar el stack completo del error para depurar; en producción JAMÁS mandes detalles internos (rutas de archivos, consultas SQL, stack traces), porque le regalas información a un atacante. Un error genérico "Algo salió mal" con el detalle solo en tus logs es la práctica correcta.

La ruta 404: el comodín que va hasta el final

¿Qué pasa si alguien pide "/rutaimposible"? Ninguna de tus rutas coincide, ningún middleware responde... y la petición se queda colgada. Para evitarlo, al FINAL de tu archivo (después de todas las rutas, antes del manejador de errores) pones un comodín que atrapa todo lo que no coincidió:

```
app.use((req, res) => {
  res.status(404).send("Página no encontrada");
});
```

Sin ruta especificada, app.use coincide con TODO. Cómo está al final, solo se ejecuta si ninguna ruta anterior respondió. Es tu red de seguridad. Y aquí va la advertencia clásica: si pones este 404 ANTES de tus rutas, las tapa a todas, porque coincide con todo y responde

antes de que las rutas tengan oportunidad. El orden vuelve a aparecer: el 404 siempre al final, sin excepciones.

Un 404 más útil devuelve JSON en una API:

```
app.use((req, res) => {
  res.status(404).json({ error: "Ruta no encontrada: " + req.url });
});
```

Así el cliente de tu API recibe un error estructurado en vez de un HTML genérico, y puede programar contra él.

Archivos estáticos: tu servidor también sirve páginas

Hasta ahora tu servidor solo responde datos: textos, JSON, mensajes. Pero un servidor web completo también sirve archivos tal cómo están: páginas HTML, hojas de estilo CSS, scripts de JavaScript para el navegador, imágenes, fuentes. A esos se les llama archivos estáticos porque se mandan tal cual, sin procesarlos ni generarlos al momento. Express trae esto integrado con `express.static`, que de hecho es un middleware más:

```
const express = require("express");
const path = require("path");
const app = express();

app.use(express.static("public"));

app.listen(3000, () => {
  console.log("Servidor corriendo en el puerto 3000");
});
```

Con esa sola línea, todo lo que pongas en la carpeta "public" queda disponible por HTTP. Si creas "public/index.html", visitar "http://localhost:3000/" lo muestra. Si creas "public/estilos.css", la dirección "http://localhost:3000/estilos.css" lo sirve. Si creas "public/js/app.js", se pide como "/js/app.js". Express mapea la estructura de carpetas a direcciones URL automáticamente, y además detecta el tipo de archivo para mandar el Content-Type correcto: text/html para HTML, text/css para CSS, application/javascript para JS, image/png para PNG. El navegador recibe cada archivo con su tipo correcto y sabe qué hacer con él.

El detalle del path: "public" es relativo a la carpeta desde donde ejecutas node, lo cual puede traicionarte si corres el servidor desde otro directorio. La forma robusta usa el módulo path de Node.js:

```
app.use(express.static(path.join(__dirname, "public")));
```

`__dirname` es la carpeta donde vive tu archivo de código, así que `path.join(__dirname, "public")` siempre apunta al lugar correcto sin importar desde dónde ejecutes node. Acostúmbrate a escribirlo así desde el primer día.

Sirviendo un frontend completo junto al backend

Aquí es donde todo se junta y se pone emocionante. Puedes tener tu API (las rutas que devuelven JSON) Y tu página web (HTML/CSS/JS) en el mismo servidor, en el mismo puerto. La estructura típica:

```
mi-proyecto/  
  servidor.js  
  public/  
    index.html  
    estilos.css  
    app.js  
    imagenes/  
      logo.png
```

Y el servidor:

```
const express = require("express");  
const path = require("path");  
const app = express();  
  
app.use(express.json());  
app.use(express.static(path.join(__dirname, "public")));  
  
// Tu API vive bajo /api para no chocar con los archivos  
app.get("/api/usuarios", (req, res) => {  
  res.json(usuarios);  
});  
  
app.get("/api/usuarios/:id", (req, res) => {  
  // ... buscar por id  
});  
  
app.listen(3000, () => {  
  console.log("Servidor corriendo en el puerto 3000");  
});
```

El index.html puede tener un fetch a "/api/usuarios" y todo funciona sin problemas de CORS, porque la página y la API están en el mismo origen. Este es el despliegue más simple posible: un solo proceso de Node.js sirve todo. Para proyectos pequeños y medianos es perfecto y es cómo funcionan miles de aplicaciones reales.

Un ejemplo de index.html que consume tu propia API:

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  <title>Mi app</title>
  <link rel="stylesheet" href="/estilos.css">
</head>
<body>
  <h1>Usuarios</h1>
  <ul id="lista"></ul>
  <script src="/app.js"></script>
</body>
</html>
```

Y el app.js del navegador:

```
fetch("/api/usuarios")
  .then((res) => res.json())
  .then((usuarios) => {
    const lista = document.getElementById("lista");
    usuarios.forEach((u) => {
      const li = document.createElement("li");
      li.textContent = u.nombre;
      lista.appendChild(li);
    });
  });
```

Fíjate en las rutas de los archivos: en el HTML se escriben como "/estilos.css" y "/app.js", con diagonal al inicio, porque son direcciones URL servidas por express.static, no rutas de archivo del disco. El navegador las pide al servidor y el servidor las encuentra en la carpeta public. Esta distinción entre "ruta URL" y "ruta de archivo" es clave para no confundirte.

Rutas virtuales: un prefijo para tus estáticos

A veces quieres que los archivos estáticos vivan bajo un prefijo, por organización o para no mezclarlos con tu API. Le pasas una ruta base a `app.use`:

```
app.use("/estatico", express.static(path.join(__dirname, "public")));
```

Ahora `"public/estilos.css"` se sirve como `"/estático/estilos.css"`. El prefijo `"/estático"` es virtual: no existe como carpeta, es solo la parte de la URL. Esto es útil cuando tu API usa la raíz y quieres separar visualmente: `"/api/..."` para datos, `"/estático/..."` para archivos. También puedes montar varias carpetas estáticas:

```
app.use(express.static(path.join(__dirname, "public")));  
app.use(express.static(path.join(__dirname, "archivos")));
```

Express las revisa en orden: si el archivo no está en `"public"`, lo busca en `"archivos"`. Es una cascada de carpetas.

Errores comunes y cómo resolverlos

Ya tienes todas las piezas; ahora los tropiezos clásicos para que no te quiten horas.

El `next()` olvidado. Síntoma: el navegador se queda cargando eternamente y nunca responde. Causa: un middleware que no llama a `next()` ni manda respuesta. Solución: revisa cada middleware y verifica que todos los caminos terminen en `next()` o en una respuesta. Un truco de depuración es poner un `console.log` al inicio y al final de cada middleware para ver dónde se atora la fila.

El orden incorrecto de middleware. Síntoma: `"mi middleware no se ejecuta"` o `"mi ruta nunca responde"`. Causas típicas: registraste el logger después de las rutas (no las ve), o pusiste el 404 antes de las rutas (las tapa todas), o montaste `express.json()` después de las rutas que leen `req.body` (llega undefined). Solución: sigue el orden canónico de arriba hacia abajo: 1) middleware globales (helmet, cors, morgan, express.json, tu logger), 2) archivos estáticos, 3) tus rutas y APIs, 4) el 404 comodín, 5) el manejador de errores. Memorízalo como una receta.

El 404 que tapa rutas. Es un caso especial del anterior pero tan común que merece su línea: si tu comodín `app.use((req, res) => res.status(404)...) está antes de alguna ruta, esa ruta jamás se ejecutará. El comodín siempre va después de la última ruta real.`

Static antes o después: ¿importa? Sí, un poco. Si montas `express.static` ANTES de tus rutas, una petición a `"/api/usuarios"` primero busca un archivo llamado `"api/usuarios"` en `public`; como

no existe, pasa a la siguiente capa y llega a tu ruta. Funciona, pero hay un riesgo: si por accidente tienes en public un archivo cuya ruta coincide con una de tu API (por ejemplo "public/api/usuarios.html" no choca, pero "public/login.html" sí podría chocar con tu ruta "/login"), el archivo estático gana porque está primero y tu ruta nunca se ejecuta. La práctica segura: monta tus rutas de API bajo un prefijo como "/api" y los estáticos en la raíz, o viceversa, y mantén los dos mundos separados por diseño. Si un día una ruta "desaparece" misteriosamente, revisa si un archivo estático la está opacando.

El parámetro que llega como texto. Ya lo viste, pero se repite tanto que va de nuevo: req.params.id es "7", no 7. Si comparas con === contra un número, falla. Convierte siempre con Number() y valida con Number.isInteger o isNaN antes de usarlo.

La query string que no existe. req.query.limite es undefined si no la mandaron, y Number(undefined) es NaN. Por eso el patrón Number(req.query.limite) || 10: si la conversión falla, NaN es falsy y cae al valor por defecto. Pero cuidado: si alguien manda limite=0, Number da 0 que también es falsy y cae al 10. Si el cero fuera un valor válido para tu caso, valida explícitamente en vez de usar ||.

El doble next() o next() después de responder. Síntoma: error "Cannot set headers after they are sent". Causa: llamaste a next() y también mandaste una respuesta, o tu código sigue ejecutándose después de un res.send sin return. Solución: usa return res.status(...).json(...) para salir temprano, y decide por camino: o next(), o respuesta, nunca ambos.

CORS en el navegador pero no en Postman. Síntoma: tu API responde perfecto en Postman o curl, pero el fetch del navegador falla con error de CORS. Causa: el navegador aplica la política de mismo origen; Postman no. Solución: instala y configura cors en tu servidor. No es un bug de tu código, es seguridad del navegador funcionando como debe.

Poniendo todo junto: el mapa mental de una petición

Para cerrar la explicación, sigue mentalmente una petición completa a un servidor bien armado. Llega GET "/api/usuarios/7?detallado=true". Primero pasa por helmet (pone headers de seguridad) y llama a next(). Luego cors (autoriza el origen) y next(). Luego morgan (registra la petición en el log) y next(). Luego express.json() (no hay body en un GET, pero revisa) y next(). Luego express.static busca un archivo "api/usuarios/7" en public, no lo encuentra, y next(). Luego llega a tu ruta "/api/usuarios/:id": el middleware portero revisa la clave, la encuentra válida, cuelga el usuario en req y next(). El manejador valida que id sea número, busca el usuario, lee req.query.detallado, arma la respuesta y hace res.json. Si en cualquier punto algo truena y llama a next(err), la petición salta directo al manejador de errores del final, que responde 500. Si ninguna ruta hubiera coincidido, el comodín 404 habría respondido.

Cada pieza tiene su lugar en la fila, cada una hace una sola cosa, y el código queda limpio y predecible. Esa es la arquitectura Express bien hecha, y ya la dominas en teoría. Vamos a la práctica.

Ejemplos paso a paso

Ejemplo 1: API de usuarios con :id y validación

Vamos a construir una mini API de usuarios que usa parámetros de ruta con validación seria. Crea un archivo servidor.js:

```
const express = require("express");
const app = express();
app.use(express.json());

const usuarios = [
  { id: 1, nombre: "Ana", correo: "ana@ejemplo.com" },
  { id: 2, nombre: "Luis", correo: "luis@ejemplo.com" },
  { id: 3, nombre: "Marta", correo: "marta@ejemplo.com" },
];

// Listar todos
app.get("/api/usuarios", (req, res) => {
  res.json(usuarios);
});

// Obtener uno por id, con validación
app.get("/api/usuarios/:id", (req, res) => {
  const id = Number(req.params.id);
  if (!Number.isInteger(id) || id <= 0) {
    return res.status(400).json({ error: "El id debe ser un número entero positivo" });
  }
  const usuario = usuarios.find((u) => u.id === id);
  if (!usuario) {
    return res.status(404).json({ error: "Usuario no encontrado" });
  }
  res.json(usuario);
});

// Crear uno nuevo
app.post("/api/usuarios", (req, res) => {
  const { nombre, correo } = req.body;
```

```

    if (!nombre || !correo) {
      return res.status(400).json({ error: "Faltan nombre o correo" });
    }
    const nuevo = { id: usuarios.length + 1, nombre, correo };
    usuarios.push(nuevo);
    res.status(201).json(nuevo);
  });

  app.use((req, res) => {
    res.status(404).json({ error: "Ruta no encontrada" });
  });

  app.listen(3000, () => console.log("Servidor en http://localhost:3000"));

```

Pruébalo paso a paso. Primero GET `/api/usuarios`: devuelve los tres. Luego GET `/api/usuarios/2`: devuelve a Luis. Luego GET `/api/usuarios/abc`: responde 400 con "El id debe ser un número entero positivo". Luego GET `/api/usuarios/99`: responde 404 con "Usuario no encontrado". Luego POST `/api/usuarios` con body `{ "nombre": "Pedro", "correo": "pedro@ejemplo.com" }`: responde 201 con el nuevo usuario y su id 4. Cada código de estado cuenta una historia distinta: 200 todo bien, 201 creado, 400 tu petición está mal, 404 no existe. Esa es una API que se comporta como las profesionales.

Ejemplo 2: buscador con query strings y paginación

Ahora un catálogo de productos con búsqueda, filtro y paginación, todo con query strings:

```

const express = require("express");
const app = express();

const productos = [
  { id: 1, nombre: "Laptop gamer", categoria: "electronica", precio: 18000 },
  { id: 2, nombre: "Mouse inalámbrico", categoria: "electronica", precio: 450 },
  { id: 3, nombre: "Silla de oficina", categoria: "muebles", precio: 3200 },
  { id: 4, nombre: "Teclado mecánico", categoria: "electronica", precio: 1500 },
  { id: 5, nombre: "Escritorio", categoria: "muebles", precio: 2800 },
  { id: 6, nombre: "Monitor 27 pulgadas", categoria: "electronica", precio: 6000 },
  { id: 7, nombre: "Lámpara de escritorio", categoria: "muebles", precio: 700 },
  { id: 8, nombre: "Audífonos bluetooth", categoria: "electronica", precio: 1200 },
];

app.get("/api/productos", (req, res) => {
  // 1. Búsqueda por texto
  let resultado = productos;
  if (req.query.buscar) {

```

```

    const termino = req.query.buscar.toLowerCase();
    resultado = resultado.filter((p) => p.nombre.toLowerCase().includes(termino));
  }
  // 2. Filtro por categoría
  if (req.query.categoria) {
    resultado = resultado.filter((p) => p.categoria === req.query.categoria);
  }
  // 3. Filtro por precio máximo
  if (req.query.precioMax) {
    const max = Number(req.query.precioMax);
    if (!isNaN(max)) {
      resultado = resultado.filter((p) => p.precio <= max);
    }
  }
  // 4. Paginación
  const pagina = Number(req.query.pagina) || 1;
  const limite = Number(req.query.limite) || 5;
  const inicio = (pagina - 1) * limite;
  const datos = resultado.slice(inicio, inicio + limite);

  res.json({
    pagina,
    limite,
    total: resultado.length,
    totalPaginas: Math.ceil(resultado.length / limite),
    datos,
  });
});

app.listen(3000, () => console.log("Servidor en http://localhost:3000"));

```

Pruébalo con estas direcciones y observa qué cambia: `/api/productos` trae los primeros 5. `/api/productos?página=2` trae los 3 restantes. `/api/productos?buscar=escritorio` encuentra el escritorio y la lámpara de escritorio. `/api/productos?categoría=electrónica&precioMax=2000` trae mouse, teclado y audífonos. `/api/productos?categoría=muebles&limite=2&página=1` trae silla y escritorio, con `totalPaginas: 2`. Una sola ruta, combinaciones infinitas. Así funcionan los catálogos reales.

Ejemplo 3: logger y validador como middleware reutilizable

Este ejemplo muestra middleware caseros aplicados a varias rutas, para que veas el ahorro de código:

```
const express = require("express");
```

```

const app = express();
app.use(express.json());

// Middleware 1: logger global (aplica a todo)
function logger(req, res, next) {
  const fecha = new Date().toLocaleString("es-MX");
  console.log("[ " + fecha + " ] " + req.method + " " + req.url);
  next();
}
app.use(logger);

// Middleware 2: validador reutilizable
function validarProducto(req, res, next) {
  const { nombre, precio } = req.body;
  if (!nombre || typeof nombre !== "string") {
    return res.status(400).json({ error: "El nombre es obligatorio y debe ser texto"
});
  }
  if (typeof precio !== "number" || precio <= 0) {
    return res.status(400).json({ error: "El precio debe ser un número mayor a cero"
});
  }
  next();
}

// Middleware 3: portero simple
function portero(req, res, next) {
  if (req.headers["x-clave"] !== "admin123") {
    return res.status(401).json({ error: "Acceso denegado" });
  }
  next();
}

const productos = [];

// Ruta pública con validación
app.post("/api/productos", validarProducto, (req, res) => {
  const nuevo = { id: productos.length + 1, ...req.body };
  productos.push(nuevo);
  res.status(201).json(nuevo);
});

// Ruta protegida: pasa por el portero Y el validador
app.post("/api/admin/productos", portero, validarProducto, (req, res) => {
  const nuevo = { id: productos.length + 1, ...req.body, destacado: true };

```

```
    productos.push(nuevo);  
    res.status(201).json(nuevo);  
  });  
  
app.listen(3000, () => console.log("Servidor en http://localhost:3000"));
```

Observa la fila en `/api/admin/productos`: primero el logger global (registra), luego el portero (revisa la clave), luego el validador (revisa los datos), y al final el manejador. Si pruebas POST sin el header `x-clave`, el portero responde 401 y el validador ni se ejecuta. Si mandas la clave pero un precio negativo, el portero deja pasar y el validador responde 400. Cada retén hace su trabajo y la ruta final solo se ejecuta cuando todo está en regla. Y el logger, registrado con `app.use` al principio, ve pasar absolutamente todo.

Ejemplo 4: sitio estático servido por Express junto a una API

El ejemplo final junta los dos mundos: una página web servida por `express.static` que consume una API del mismo servidor. Estructura de archivos:

```
tienda/  
  servidor.js  
  public/  
    index.html  
    estilos.css  
    app.js
```

El servidor:

```

const express = require("express");
const path = require("path");
const app = express();

app.use(express.static(path.join(__dirname, "public")));

const productos = [
  { id: 1, nombre: "Laptop gamer", precio: 18000 },
  { id: 2, nombre: "Mouse inalámbrico", precio: 450 },
  { id: 3, nombre: "Teclado mecánico", precio: 1500 },
];

app.get("/api/productos", (req, res) => {
  res.json(productos);
});

app.get("/api/productos/:id", (req, res) => {
  const id = Number(req.params.id);
  const producto = productos.find((p) => p.id === id);
  if (!producto) {
    return res.status(404).json({ error: "Producto no encontrado" });
  }
  res.json(producto);
});

app.listen(3000, () => console.log("Abre http://localhost:3000"));

```

El public/index.html:

```

<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  <title>Mi tiendita</title>
  <link rel="stylesheet" href="/estilos.css">
</head>
<body>
  <h1>Catálogo</h1>
  <div id="catalogo"></div>
  <script src="/app.js"></script>
</body>
</html>

```

El public/estilos.css:

```
body { font-family: Arial, sans-serif; max-width: 600px; margin: 40px auto; }
.tarjeta { border: 1px solid #ccc; padding: 12px; margin: 8px 0; border-radius: 8px; }
.precio { color: green; font-weight: bold; }
```

El public/app.js (corre en el navegador):

```
fetch("/api/productos")
  .then((res) => res.json())
  .then((productos) => {
    const contenedor = document.getElementById("catalogo");
    productos.forEach((p) => {
      const div = document.createElement("div");
      div.className = "tarjeta";
      div.innerHTML = "<h3>" + p.nombre + "</h3>" +
        "<p class='precio'>$" + p.precio + "</p>";
      contenedor.appendChild(div);
    });
  })
  .catch((err) => console.error("Error:", err));
```

Abre "http://localhost:3000" y verás la página con el catálogo cargado desde tu propia API, sin CORS, sin dos servidores, sin complicaciones. El HTML, el CSS y el JS los sirvió express.static; los datos los sirvieron tus rutas. Frontend y backend conviviendo en un solo proceso de Node.js. Cuando despliegues esto a un servicio de hosting, subes una sola carpeta y funciona. Este ejemplo es, en miniatura, la arquitectura de miles de aplicaciones productivas.

Práctica

Ejercicio 1: ruta con dos parámetros y validación

Crea una ruta GET "/tienda/:categoría/:id" que responda con el producto correspondiente de este catálogo. Valida que el id sea un número entero positivo (400 si no) y que el producto exista Y pertenezca a la categoría pedida (404 si no).

```
const productos = [
  { id: 1, categoria: "libros", nombre: "El Quijote" },
  { id: 2, categoria: "libros", nombre: "Cien años de soledad" },
  { id: 3, categoria: "musica", nombre: "Guitarra acústica" },
];
```


Solución paso a paso. Primero extraemos los dos parámetros de req.params. Luego convertimos el id a número y validamos que sea entero positivo; si no, respondemos 400 y salimos con return. Después buscamos el producto que coincida en id Y en categoría al mismo tiempo; si no lo encontramos, respondemos 404. Si todo sale bien, respondemos el producto en JSON:

```
app.get("/tienda/:categoria/:id", (req, res) => {
  const categoria = req.params.categoria;
  const id = Number(req.params.id);
  if (!Number.isInteger(id) || id <= 0) {
    return res.status(400).json({ error: "El id debe ser un número entero positivo"
  });
  }
  const producto = productos.find((p) => p.id === id && p.categoria === categoria);
  if (!producto) {
    return res.status(404).json({ error: "Producto no encontrado en esa categoría"
  });
  }
  res.json(producto);
});
```

Prueba: "/tienda/libros/1" devuelve El Quijote. "/tienda/música/1" devuelve 404, porque el producto 1 es de libros aunque el id exista: la categoría también debe coincidir. "/tienda/libros/abc" devuelve 400. Este ejercicio te entrena en la validación doble que toda ruta con parámetros necesita: que el dato tenga el formato correcto y que el recurso realmente exista.

Ejercicio 2: paginación con valores por defecto y límites sanos

Tienes un arreglo "artículos" con 50 elementos. Crea GET "/api/artículos" que acepte "página" y "limite" por query string, con valores por defecto página=1 y limite=10, y que nunca permita un límite mayor a 50 (para que nadie pida un millón de registros de golpe). La respuesta debe incluir página, limite, total y datos.

Solución paso a paso. Primero leemos página y limite con sus valores por defecto usando el patrón Number(...) || valor. Después blindamos el limite: si es mayor a 50, lo recortamos a 50; si es menor a 1, lo subimos a 1. Lo mismo con página para que no sea menor a 1. Luego aplicamos la fórmula inicio = (página - 1) * limite y cortamos con slice. Finalmente respondemos el objeto con la metadata:

```

app.get("/api/articulos", (req, res) => {
  let pagina = Number(req.query.pagina) || 1;
  let limite = Number(req.query.limite) || 10;
  if (limite > 50) limite = 50;
  if (limite < 1) limite = 10;
  if (pagina < 1) pagina = 1;
  const inicio = (pagina - 1) * limite;
  const datos = articulos.slice(inicio, inicio + limite);
  res.json({
    pagina: pagina,
    limite: limite,
    total: articulos.length,
    totalPaginas: Math.ceil(articulos.length / limite),
    datos: datos,
  });
});

```

Prueba `"/api/artículos?limite=100"`: el servidor lo recorta a 50 y responde con limite: 50. Prueba `"?página=3&limite=10"`: trae los artículos del 20 al 29. El blindaje del limite es una práctica de seguridad real: sin él, un cliente malicioso podría pedir `limite=999999999` y tumbar tu servidor o tu base de datos. Nunca confíes en los números que manda el cliente.

Ejercicio 3: middleware logger con tiempo de respuesta

Escribe un middleware global que registre método, URL, código de estado y milisegundos que tardó cada petición, usando el evento "finish" de la respuesta. Móntalo con `app.use` antes de dos rutas de ejemplo y verifica en la consola que funciona.

Solución paso a paso. El middleware guarda la hora de inicio con `Date.now()`. Luego se suscribe al evento "finish" del objeto `res`: ese evento se dispara cuando la respuesta ya se envió por completo, así que dentro del callback ya conocemos `res.statusCode` real. Después llama a `next()` para que la petición continúe (importante: `next()` va fuera del callback del evento, no dentro). Finalmente lo registramos con `app.use` antes de las rutas:

```
function logger(req, res, next) {
  const inicio = Date.now();
  res.on("finish", () => {
    const ms = Date.now() - inicio;
    console.log(req.method + " " + req.url + " -> " + res.statusCode + " (" + ms +
"ms)");
  });
  next();
}

app.use(logger);

app.get("/rapido", (req, res) => {
  res.send("Respuesta inmediata");
});

app.get("/lento", (req, res) => {
  setTimeout(() => res.send("Respuesta con retraso"), 500);
});
```

Al pedir `/rápido` verás algo como `GET /rápido -> 200 (3ms)` y al pedir `/lento` verás `GET /lento -> 200 (505ms)`. Este middleware te enseña dos lecciones en una: que el middleware puede hacer trabajo DESPUÉS de que la ruta responda (gracias a los eventos), y que `next()` debe llamarse de inmediato para no frenar la fila. Es el mismo principio que usan morgan y los loggers profesionales.

Ejercicio 4: portero con roles y rutas protegidas

Crea un middleware "portero" que lea el header `"x-rol"` de la petición. Si el rol es `"admin"`, deja pasar y guarda el rol en `req.rol`. Si no, responde 403 con un mensaje de error. Úsalo solo en la ruta GET `/admin/reporte` (no globalmente), y deja pública la ruta GET `/info`.

Solución paso a paso. El middleware lee `req.headers["x-rol"]`; recuerda que los nombres de headers llegan en minúsculas. Si el valor no es exactamente `"admin"`, responde 403 (prohibido: el cliente sí se identificó pero no tiene permiso; el 401 se usa cuando falta autenticación). Si es `admin`, guarda el rol en `req.rol` para que la ruta lo aproveche y llama a `next()`. Luego montamos el portero solo en la ruta protegida, pasándolo como segundo argumento de `app.get`:

```
function portero(req, res, next) {
  const rol = req.headers["x-rol"];
  if (rol !== "admin") {
    return res.status(403).json({ error: "Solo los administradores pueden ver este reporte" });
  }
  req.rol = rol;
  next();
}

app.get("/info", (req, res) => {
  res.json({ mensaje: "Información pública, todos pueden verla" });
});

app.get("/admin/reporte", portero, (req, res) => {
  res.json({ mensaje: "Reporte secreto", vistoPor: req.rol });
});
```

Prueba con curl o Postman: GET `/info` responde sin headers especiales. GET `/admin/reporte` sin header responde 403. GET `/admin/reporte` con header `x-rol: admin` responde el reporte. Este ejercicio muestra el middleware a nivel de ruta: protege exactamente lo que necesita protección y deja lo demás libre, sin afectar al resto de la aplicación.

Ejercicio 5: API + página estática que la consume

Crea un proyecto con esta estructura: `servidor.js` y carpeta `public/` con `index.html` y `app.js`. El servidor debe servir los archivos estáticos y exponer GET `/api/saludo/:nombre` que responda `{ "mensaje": "Hola, <nombre>" }`. La página debe tener un input, un botón y un párrafo; al hacer clic, el `app.js` del navegador pide `/api/saludo/<lo que escribió el usuario>` y muestra el mensaje en el párrafo.

Solución paso a paso. Primero el servidor: montamos `express.static` con `path.join(__dirname, "public")` para servir la página, y definimos la ruta con el parámetro `:nombre`. Como el nombre viene de la URL, Express ya lo decodifica (si escriben "María José" llega bien).

```

const express = require("express");
const path = require("path");
const app = express();

app.use(express.static(path.join(__dirname, "public")));

app.get("/api/saludo/:nombre", (req, res) => {
  const nombre = req.params.nombre;
  res.json({ mensaje: "Hola, " + nombre });
});

app.listen(3000, () => console.log("Abre http://localhost:3000"));

```

Ahora public/index.html:

```

<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  <title>Saludador</title>
</head>
<body>
  <h1>Saludador</h1>
  <input id="nombre" placeholder="Escribe tu nombre">
  <button id="boton">Saludar</button>
  <p id="resultado"></p>
  <script src="/app.js"></script>
</body>
</html>

```

Y public/app.js (código del navegador):

```

document.getElementById("boton").addEventListener("click", () => {
  const nombre = document.getElementById("nombre").value.trim();
  if (!nombre) return;
  fetch("/api/saludo/" + encodeURIComponent(nombre))
    .then((res) => res.json())
    .then((data) => {
      document.getElementById("resultado").textContent = data.mensaje;
    })
    .catch((err) => console.error("Error:", err));
});

```

Fíjate en `encodeURIComponent`: convierte espacios y caracteres especiales en un formato seguro para URLs, así "María José" viaja como "Mar%C3%ADA%20Jos%C3%A9" y Express lo decodifica de vuelta. Al abrir "`http://localhost:3000`", escribir tu nombre y hacer clic, verás el saludo. Acabas de construir una aplicación full-stack mínima: frontend servido por Express, backend en Express, comunicándose por tu propia API con parámetros de ruta. Todo lo de esta unidad, trabajando junto.

Proyecto integrador: API de biblioteca completa

Ya viste cada pieza por separado. Ahora vamos a armar el rompecabezas completo: una API de biblioteca con libros, parámetros de ruta, query strings para buscar y paginar, middleware de logger, portero y validación, manejo centralizado de errores, ruta 404 y una página estática que consume la API. Este proyecto usa TODO lo de la unidad en un solo archivo, y es el mejor repaso posible.

Paso 1: la base y los datos. Creamos el servidor, registramos `express.json()` y definimos un arreglo de libros:

```
const express = require("express");
const path = require("path");
const app = express();

app.use(express.json());

const libros = [
  { id: 1, titulo: "Cien años de soledad", autor: "García Márquez", genero: "novela", anio: 1967 },
  { id: 2, titulo: "La casa de los espíritus", autor: "Isabel Allende", genero: "novela", anio: 1982 },
  { id: 3, titulo: "Ficciones", autor: "Jorge Luis Borges", genero: "cuentos", anio: 1944 },
  { id: 4, titulo: "Rayuela", autor: "Julio Cortázar", genero: "novela", anio: 1963 },
  { id: 5, titulo: "El Aleph", autor: "Jorge Luis Borges", genero: "cuentos", anio: 1949 },
  { id: 6, titulo: "Como agua para chocolate", autor: "Laura Esquivel", genero: "novela", anio: 1989 },
];
```

Paso 2: los middleware globales. El logger va primero para verlo todo:

```
function logger(req, res, next) {
  const inicio = Date.now();
  res.on("finish", () => {
    console.log(req.method + " " + req.url + " -> " + res.statusCode + " (" +
(Date.now() - inicio) + "ms)");
  });
  next();
}
app.use(logger);
```

Paso 3: los middleware reutilizables. El validador de libros nuevos y el portero para las rutas de administración:

```
function validarLibro(req, res, next) {
  const { titulo, autor } = req.body;
  if (!titulo || !autor) {
    return res.status(400).json({ error: "El libro necesita título y autor" });
  }
  next();
}

function soloAdmin(req, res, next) {
  if (req.headers["x-rol"] !== "admin") {
    return res.status(403).json({ error: "Se requiere rol de administrador" });
  }
  next();
}
```

Paso 4: las rutas de lectura con parámetros y query strings. Aquí conviven :id con búsqueda, filtro y paginación:

```

app.get("/api/libros", (req, res) => {
  let resultado = libros;
  if (req.query.buscar) {
    const t = req.query.buscar.toLowerCase();
    resultado = resultado.filter((l) =>
      l.titulo.toLowerCase().includes(t) || l.autor.toLowerCase().includes(t)
    );
  }
  if (req.query.genero) {
    resultado = resultado.filter((l) => l.genero === req.query.genero);
  }
  const pagina = Number(req.query.pagina) || 1;
  let limite = Number(req.query.limite) || 5;
  if (limite > 20) limite = 20;
  if (limite < 1) limite = 5;
  const inicio = (pagina - 1) * limite;
  res.json({
    pagina, limite, total: resultado.length,
    datos: resultado.slice(inicio, inicio + limite),
  });
});

app.get("/api/libros/:id", (req, res, next) => {
  try {
    const id = Number(req.params.id);
    if (!Number.isInteger(id) || id <= 0) {
      return res.status(400).json({ error: "El id debe ser un número entero positivo" });
    }
    const libro = libros.find((l) => l.id === id);
    if (!libro) {
      return res.status(404).json({ error: "Libro no encontrado" });
    }
    res.json(libro);
  } catch (err) {
    next(err);
  }
});

```

Paso 5: las rutas de escritura, protegidas y validadas. Fíjate en la cadena de middleware:


```

app.post("/api/libros", soloAdmin, validarLibro, (req, res) => {
  const nuevo = { id: libros.length + 1, ...req.body };
  libros.push(nuevo);
  res.status(201).json(nuevo);
});

app.delete("/api/libros/:id", soloAdmin, (req, res, next) => {
  try {
    const id = Number(req.params.id);
    const indice = libros.findIndex((l) => l.id === id);
    if (indice === -1) {
      return res.status(404).json({ error: "Libro no encontrado" });
    }
    const eliminado = libros.splice(indice, 1)[0];
    res.json({ mensaje: "Libro eliminado", libro: eliminado });
  } catch (err) {
    next(err);
  }
});

```

Paso 6: los estáticos, el 404 y el manejador de errores, en ese orden estricto:

```

app.use(express.static(path.join(__dirname, "public")));

app.use((req, res) => {
  res.status(404).json({ error: "Ruta no encontrada: " + req.url });
});

app.use((err, req, res, next) => {
  console.error("Error interno:", err.message);
  res.status(500).json({ error: "Algo salió mal en el servidor" });
});

app.listen(3000, () => console.log("Biblioteca en http://localhost:3000"));

```

Recorre mentalmente la fila una vez más con una petición real: POST `/api/libros` con header `"x-rol: admin"` y body válido. Entra por `express.json` (parsea el body), pasa por el logger (empieza a medir), los estáticos buscan un archivo `"api/libros"` y no lo encuentran, llega a la ruta: `soloAdmin` revisa el rol y deja pasar, `validarLibro` revisa título y autor y deja pasar, el manejador crea el libro y responde 201. Al terminar, el evento `finish` del logger imprime `"POST /api/libros -> 201 (4ms)"`. Si el rol no fuera `admin`, `soloAdmin` respondería 403 y nada más se ejecutaría. Si la ruta no existiera, el 404 respondería. Si algo tronara adentro, `next(err)` llevaría al manejador de errores. Cada capa hace su trabajo, en orden, sin repetirse.

Para la página estática, crea `public/index.html` con un buscador que use las query strings:

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  <title>Biblioteca</title>
</head>
<body>
  <h1>Biblioteca</h1>
  <input id="buscar" placeholder="Buscar por título o autor">
  <button id="btn">Buscar</button>
  <ul id="resultados"></ul>
  <script>
    document.getElementById("btn").addEventListener("click", () => {
      const q = document.getElementById("buscar").value.trim();
      fetch("/api/libros?buscar=" + encodeURIComponent(q) + "&limite=10")
        .then((res) => res.json())
        .then((data) => {
          const ul = document.getElementById("resultados");
          ul.innerHTML = "";
          data.datos.forEach((l) => {
            const li = document.createElement("li");
            li.textContent = l.titulo + " - " + l.autor + " (" + l.anio + ")";
            ul.appendChild(li);
          });
        });
    });
  </script>
</body>
</html>
```

Abre `"http://localhost:3000"`, escribe `"borges"` y verás sus dos libros. Escribe `"novela"` en el buscador... no funcionará, porque la búsqueda solo mira título y autor, no género. ¿Ves el

reto? Modifica el servidor para que también busque en el género, o agrega un select de género que use "?genero=novela". Ese tipo de mejoras pequeñas es exactamente como crecen las aplicaciones reales: una query string a la vez.

Este proyecto integrador resume la unidad entera en unas 120 líneas: parámetros que identifican recursos, query strings que filtran y paginan, middleware que registran, protegen y validan, errores que se centralizan, un 404 que atrapa lo desconocido y un frontend servido por el mismo servidor. Si entiendes cada línea de este archivo y puedes explicar por qué está en ese orden, dominas la unidad.

Repaso rápido: preguntas y respuestas

Antes del resumen final, un repaso en formato de preguntas, como si te estuvieras preparando para un examen o una entrevista de trabajo. Si puedes responder cada una sin mirar arriba, la unidad es tuya.

¿Qué es un parámetro de ruta y cómo se declara? Es una parte variable de la dirección URL que captura un dato, y se declara con dos puntos delante del nombre, como en "/usuarios/:id". Cuando alguien visita "/usuarios/7", Express guarda "7" en req.params.id. Sirve para identificar UN recurso específico: un usuario, un producto, un artículo. Sin parámetros tendrías que crear una ruta distinta por cada recurso posible, lo cual es imposible. El parámetro convierte una ruta fija en una plantilla que atiende infinitas direcciones con el mismo código.

¿En qué se diferencia req.params de req.query? req.params contiene los datos que son parte obligatoria de la ruta ("/usuarios/7" no existe sin el 7), mientras que req.query contiene pares clave=valor opcionales después del signo "?" ("/usuarios?rol=admin"). La analogía: el parámetro es la dirección de la casa, la query string son las instrucciones extra para el repartidor. En diseño de APIs, los parámetros identifican el QUÉ (qué recurso quiero) y las query strings modifican el CÓMO (filtrado, ordenado, paginado, buscado).

¿Por qué req.params.id llega como texto y por qué importa? Porque todo lo que viaja en una URL es texto; no existe el concepto de "número" en una dirección web. Si tu arreglo guarda ids numéricos y comparas con === sin convertir, "7" === 7 es falso y tu búsqueda falla aunque el usuario exista. La solución es convertir siempre: Number(req.params.id), y validar el resultado con Number.isInteger o isNaN antes de usarlo. Este es uno de esos errores que te hacen perder una hora la primera vez y cinco segundos todas las demás, una vez que lo aprendes.

¿Cómo funciona la paginación con query strings? El cliente pide "?pagina=2&limite=10" y el servidor calcula qué pedazo del total le toca: inicio = (pagina - 1) * limite, y devuelve slice(inicio, inicio + limite). Con página 2 y límite 10, inicio = 10, y se devuelven los elementos

10 a 19. Además se responde con metadata (página, limite, total, totalPaginas) para que el cliente sepa cuántas páginas hay. Y siempre se blindan los valores: un límite máximo para que nadie pida un millón de registros, y valores por defecto cuando no mandan nada.

¿Qué es un middleware en Express? Es una función que se ejecuta entre la petición y la respuesta, recibiendo (req, res, next). Forma una fila: cada middleware hace su trabajo y luego llama a next() para pasar la petición al siguiente, que puede ser otro middleware o el manejador final de la ruta. Sirve para lógica transversal: registrar peticiones, validar datos, verificar permisos, leer el body. La gracia es escribir esa lógica una vez y aplicarla a muchas rutas, en vez de repetirla en cada manejador.

¿Qué pasa si un middleware no llama a next() ni responde? La petición se queda colgada para siempre: el cliente espera hasta que se rinde con un timeout. Es el error más desconcertante para un principiante porque no hay mensaje de error, solo silencio. La regla de oro: todo middleware termina con next() para continuar o con una respuesta (res.send, res.json) para terminar ahí. Nunca las dos cosas, nunca ninguna.

¿Importa el orden en que registro middleware y rutas? Muchísimo: Express los ejecuta de arriba hacia abajo, en el orden de registro. Un middleware solo afecta a lo que está debajo de él. Por eso los middleware globales van al principio (antes de las rutas), el 404 comodín va después de todas las rutas (si va antes, las tapa a todas) y el manejador de errores va al final de todo. Si "tu middleware no se ejecuta", lo primero que debes revisar es dónde lo registraste.

¿Cuál es la diferencia entre app.use y app.get para montar un middleware? app.use lo monta para todos los métodos HTTP y para la ruta base más todo lo que cuelgue de ella: app.use("/api", m) corre en GET, POST, PUT y DELETE bajo "/api". En cambio, pasar el middleware como argumento de app.get("/ruta", m, manejador) lo limita a peticiones GET a esa ruta exacta. Usa app.use para lo global (loggers, seguridad) y middleware por ruta para lo específico (validar los datos de un formulario concreto, proteger solo el panel de admin).

¿Qué es el middleware de manejo de errores y cómo se reconoce? Es una función con CUATRO parámetros (err, req, res, next) en vez de tres, registrada al final de todo. Express salta directo a ella cuando alguna ruta o middleware llama a next(err) pasándole un error. Centraliza el manejo de fallos: en vez de try/catch repetidos en cada ruta con respuestas inconsistentes, todos los errores terminan en un solo lugar que los registra y responde un 500 genérico. En producción nunca debe exponer detalles internos como stack traces o rutas de archivos.

¿Para qué sirve la ruta 404 comodín? Atrapa todas las peticiones que no coincidieron con ninguna ruta, para que no se queden colgadas. Se escribe como app.use((req, res) => ...) sin ruta específica, y OBLIGATORIAMENTE va después de la última ruta real y antes del

manejador de errores. En una API conviene responder JSON con el error en vez de HTML genérico, para que los clientes puedan programar contra él.

¿Qué hacen morgan, cors y helmet? morgan es el logger profesional: con `app.use(morgan("dev"))` registra método, URL, código de estado y tiempo de cada petición sin que escribas nada. cors agrega los headers que permiten a páginas de otro origen consumir tu API; sin él, el fetch del navegador falla aunque Postman funcione, por la política de mismo origen del navegador. helmet configura automáticamente headers de seguridad en las respuestas, protegiendo contra ataques comunes; es práctica estándar en cualquier servidor expuesto a internet.

¿Qué hace `express.static` y por qué se usa `path.join(__dirname, "public")`? `express.static` es un middleware que sirve archivos tal cual están en disco (HTML, CSS, JS, imágenes), mapeando la estructura de carpetas a URLs y poniendo el Content-Type correcto automáticamente. Se usa `path.join(__dirname, "public")` porque `__dirname` siempre apunta a la carpeta del archivo de código, mientras que una ruta relativa como "public" depende de desde dónde ejecutes node y puede romperse. Es la forma robusta que debes usar siempre.

¿Cómo conviven frontend y backend en el mismo servidor? Montas `express.static` para servir la página y defines tus rutas de API bajo un prefijo como `"/api"`. La página usa fetch a `"/api/..."` y, al estar en el mismo origen, no hay problemas de CORS. Es el despliegue más simple: un solo proceso de Node.js sirve todo. El único cuidado es no dejar que un archivo estático opaque una ruta de la API: mantén separados los dos mundos por diseño (`"/api/..."` para datos, raíz para archivos).

¿Cuál es el error "Cannot set headers after they are sent"? Aparece cuando intentas responder dos veces a la misma petición: por ejemplo, llamaste a `next()` después de hacer `res.send`, o tu código siguió ejecutándose después de un `res.send` sin un `return`. La solución es el patrón "valida y salte temprano": `return res.status(...).json(...)` para salir de la función en cuanto respondes un error, y decidir por cada camino si toca `next()` o respuesta, nunca ambos.

¿Cuándo conviene un parámetro opcional (`:categoría?`)? Cuando la misma ruta debe funcionar con y sin ese dato, como `"/artículos"` (todos) y `"/artículos/deportes"` (filtrados). Se declara agregando `"?"` al nombre del parámetro y, si no viene, su valor es `undefined`, así que el manejador debe contemplar ambos casos. Úsalo con moderación: demasiados parámetros opcionales vuelven las rutas ambiguas y difíciles de documentar. En la mayoría de los diseños, lo opcional va mejor como query string (`"?categoría=deportes"`) y los parámetros se reservan para lo obligatorio.

¿Cómo enriquece un middleware el objeto req y para qué sirve? Cualquier middleware puede colgar datos en req (`req.usuario`, `req.rol`, `req.inicio`) y todo lo que venga después en la fila los

puede leer. `express.json()` hace exactamente eso con `req.body`: parsea el cuerpo y lo deja listo para las rutas. El portero del ejemplo lo usa para anotar quién pasó la revisión. Este patrón evita pasar datos como argumentos entre funciones y es la forma estándar de comunicar información "hacia adelante" en la cadena de Express.

¿Qué diferencia hay entre 400, 401, 403 y 404? El 400 (Bad Request) dice "tu petición está mal formada": un id que no es número, un body sin los campos requeridos. El 401 (Unauthorized) dice "no sé quién eres": falta la clave o el token. El 403 (Forbidden) dice "sé quién eres pero no tienes permiso": el rol no es admin. El 404 (Not Found) dice "eso no existe": el usuario 99 no está, o la ruta no coincide con nada. Usar el código correcto no es decoración: los clientes programan su lógica contra estos códigos, así que un 404 cuando en realidad fue un 400 confunde a quien consume tu API.

¿Por qué el 404 comodín y el manejador de errores van al final, en ese orden? Porque Express evalúa de arriba hacia abajo y el comodín coincide con todo: si estuviera antes, ninguna ruta debajo de él se ejecutaría jamás. El manejador de errores va después del 404 porque es el último recurso: solo se activa con `next(err)`, y ponerlo al final garantiza que cualquier error de cualquier capa llegue a él. El orden canónico completo es: middleware globales, archivos estáticos, rutas, 404 comodín, manejador de errores. Si memorizas esa receta, la mitad de los bugs de Express desaparecen.

Mi API funciona en Postman pero el navegador bloquea el fetch, ¿qué hago? Casi con seguridad es CORS: el navegador aplica la política de mismo origen y bloquea peticiones de tu página hacia un servidor de distinto origen (otro puerto cuenta como distinto origen), mientras que Postman no aplica esa política. La solución es instalar el paquete `cors` y montarlo con `app.use(cors())` en desarrollo, o con `cors({ origin: "https://miapp.com" })` en producción para permitir solo tus dominios. No es un bug de tu código: es el navegador protegiendo al usuario, y ahora sabes exactamente dónde se configura el permiso.

Resumen de la unidad

En esta unidad tu servidor aprendió tres superpoderes. Primero, las rutas inteligentes: los parámetros de ruta (`:id`) capturan datos que son parte de la dirección y llegan en `req.params` siempre como texto (convíértelos con `Number` y válidalos antes de usarlos), mientras que las query strings (`?q=react&limite=10`) mandan instrucciones opcionales que llegan en `req.query` y sirven para búsquedas, filtros y paginación con la fórmula $\text{inicio} = (\text{página} - 1) * \text{limite}$. La analogía que te debe quedar: el parámetro es la dirección de la casa, la query string son las instrucciones para el repartidor.

Segundo, el middleware: funciones entre la petición y la respuesta que forman una fila de retenes. Todo middleware termina con `next()` o con una respuesta, nunca con las dos y nunca con ninguna, porque olvidar `next()` cuelga la petición para siempre. El orden de registro es el orden de ejecución: los middleware globales van al principio, las rutas en medio, el 404 comodín después de todas las rutas y el manejador de errores (el de cuatro parámetros) al final. Con middleware caseros resolviste el logger, el validador y el portero de autenticación; con paquetes de terceros conociste morgan para logs profesionales, cors para permitir peticiones entre orígenes y helmet para headers de seguridad.

Tercero, los archivos estáticos: con `app.use(express.static(path.join(__dirname, "public")))` tu servidor sirve HTML, CSS, JS e imágenes tal cual, mapeando carpetas a URLs, lo que te permite hospedar el frontend y el backend en el mismo proceso y olvidarte de CORS en tus propios proyectos.

Y la lección transversal de toda la unidad: nunca confíes en lo que manda el cliente. Valida los parámetros, blindas los límites de paginación, revisa las claves en el portero y no expongas detalles internos en los errores. Un servidor que valida es un servidor que sobrevive. En la siguiente unidad seguirás endureciendo tu backend con temas que se construyen directamente sobre estos cimientos.

Antes de cerrar, un consejo práctico para cuando salgas a construir por tu cuenta. Cuando algo no funcione en tu servidor, depura siguiendo la fila en orden: primero verifica que la petición llega (mira el log de morgan o de tu logger casero); después revisa que cada middleware llame a `next()` (un `console.log` al inicio y al final de cada uno revela dónde se atora); luego confirma que el orden sea el canónico (globales, estáticos, rutas, 404, errores); y al final revisa los datos (¿el parámetro llegó como texto?, ¿la query trae lo que esperas?, ¿el body se parseó?). Nueve de cada diez bugs de Express caen en una de esas cuatro revisiones. Con esa rutina de diagnóstico y todo lo que practicaste aquí, tienes las herramientas para construir APIs ordenadas, seguras y completas. Nos vemos en la unidad 10.

Recuerda también guardar este proyecto integrador de la biblioteca en tu portafolio: demuestra parámetros de ruta, query strings, middleware propio, manejo de errores y archivos estáticos en un solo programa, que es justo lo que un reclutador quiere ver en un backend junior de Node.js. Éxito en tu camino como desarrollador backend.

UNIDAD 10: APIs REST y proyecto final: tu backend completo

Bienvenido a la Unidad 10. Esta es la última unidad del "Curso Completo de Node.js" de Lucas EGA Academy, y eso significa algo importante: llegaste al final de un camino que empezaste desde cero. En las primeras unidades encendiste tu primer servidor con el módulo http nativo y viste el famoso "Hola mundo" aparecer en el navegador. Después aprendiste a organizar tu código con módulos, a leer y escribir archivos con fs, a reaccionar a eventos con EventEmitter y a dominar lo asincrono con callbacks, promesas y async/await. Y en las unidades recientes llegó la joya de la corona: Express, el framework que convierte a Node.js en una máquina de construir servidores web. Ahora, en esta unidad final, todas esas piezas se unen en una sola construcción: vas a levantar una API REST completa, un backend funcional de verdad, con operaciones de crear, leer, actualizar y borrar (el famoso CRUD), datos que sobreviven a los reinicios gracias a la persistencia en archivo JSON, validación de datos, códigos de estado correctos y hasta un frontend mínimo que consume tu API usando fetch. Al terminar esta unidad no solo sabrás que es una API REST: tendrás una en tus manos, hecha por ti, y entenderas cómo se conecta el mundo del backend con el mundo del frontend. Esa es la habilidad que separa a quien "sabe un poco de JavaScript" de quien puede construir aplicaciones reales de principio a fin.

Piénsalo así: durante el curso aprendiste las piezas por separado, como quien aprende a usar el martillo, la sierra y el destornillador. Ahora vas a construir el mueble completo. Una API REST es el puente más común entre lo que ve el usuario (el frontend: una página web, una app de celular) y los datos que viven en el servidor (el backend). Cuando abres tu app de notas en el teléfono y ves tu lista, hubo una API que entregó esos datos. Cuando marcas una tarea como hecha, hubo una API que recibió esa orden. Cuando borras algo, hubo una API que lo eliminó. Las APIs REST son, literalmente, el idioma con el que las aplicaciones modernas se comunican entre sí, y después de esta unidad serás una de las personas que sabe construirlas.

Esta unidad cubre las clases 27 a 30 del curso. En la clase 27 aprenderás que es JSON y que son las APIs REST, con la analogía del mesero que te ayudara a entender los verbos HTTP y el diseño de rutas. En la clase 28 pondrás manos a la obra con el proyecto central: una API de tareas con CRUD completo, donde los datos viven en memoria en un arreglo de JavaScript: tu primer backend funcional. En la clase 29 conectaras esa API con un frontend usando fetch, con la línea `fetch("http://localhost:3000/api/tareas").then(r => r.json())`, y vivirás el momento mágico del full stack: tu propio frontend hablando con tu propia API, hecho con tus propias

manos. Y en la clase 30 celebraremos tu graduación: un repaso de todo lo que dominas (módulos, Express, APIs REST) y la ruta clara de lo que sigue, porque un buen curso no termina con un adiós, termina con un "ahora ve por más". El siguiente paso natural: bases de datos con SQL, y todo el camino sigue en lucasegaacademy.com.

Una advertencia honesta antes de empezar: esta unidad es la más larga y la más densa del curso, y es a propósito. Construir un backend completo no se aprende con tres párrafos y un ejemplo de juguete; se aprende viendo el código real, entendiendo cada decisión y practicando con ejercicios que te obligan a pensar. Así que ponte cómodo, abre tu editor, ten tu terminal a la mano y trabaja esta unidad cómo se debe: leyendo, escribiendo código y rompiendo cosas para aprender a arreglarlas. Lo que construyas aquí será tu primer backend funcional, y nadie te lo va a quitar.

Explicación detallada

Que es una API REST

API significa "Application Programming Interface", interfaz de programación de aplicaciones. En lenguaje sencillo: es un contrato que dice "si me hablas de esta forma, te respondo de esta otra forma". Una API es la manera en que un programa le pide cosas a otro programa sin necesidad de saber cómo está hecho por dentro. Tu no necesitas saber cómo funciona el motor de un coche para manejarlo: necesitas saber que el volante gira, el acelerador acelera y el freno frena. Una API es exactamente eso: el volante, el acelerador y el freno de un sistema de software.

REST significa "Representational State Transfer", transferencia de estado representacional. Es un estilo de arquitectura, es decir, un conjunto de reglas y buenas prácticas para diseñar APIs que funcionen sobre HTTP, el mismo protocolo que usa la web. Cuando una API sigue esas reglas, la llamamos API REST o API RESTful. La enorme mayoría de las APIs del mundo real son REST: las de las redes sociales, las de GitHub, las de los servicios de mapas, las de las tiendas en línea. Si aprendes a construir APIs REST, aprendes el dialecto más hablado del backend moderno.

La analogía clásica para entender una API REST es la del mesero con menú fijo en un restaurante. Imagínate que tu (el cliente, que en nuestro caso es el frontend o cualquier programa que consume la API) entras a un restaurante. Tu no entras a la cocina a cocinar tu propia comida: eso sería un caos y un riesgo. En su lugar, hay un mesero (la API) que te atiende. El mesero te da un menú fijo (la documentación de la API: la lista de cosas que puedes pedir y cómo pedir las). Tu le dices "quiero la ensalada número 3" (haces una petición

HTTP a una ruta específica), el mesero va a la cocina (el servidor, los datos), trae tu plato (la respuesta con los datos) y te lo sirve en la mesa. Tu no sabes ni necesitas saber cómo se cocina la ensalada, que sartenes se usaron ni quien es el chef. Solo sabes que pediste del menú y recibiste lo que pediste. Y ojo con el detalle del "menú fijo": no puedes pedirle al mesero "sorprendeme con algo": solo puedes pedir lo que está en el menú. Igual con una API: solo puedes usar las rutas y operaciones que el backend expone; si intentas pedir algo fuera del menú, recibes un error.

Esta analogía explica varias cosas a la vez. Primero: el cliente y el servidor están separados. El frontend no toca los datos directamente, igual que tu no entras a la cocina. Segundo: la comunicación sigue un formato acordado. El menú dice exactamente que puedes pedir; la API dice exactamente que rutas existen y que datos acepta. Tercero: el mesero no recuerda tu pedido anterior a menos que se lo digas. Cada vez que pides, tienes que decirle todo lo que necesita saber: tu mesa, tu plato, tus detalles. Esto nos lleva a uno de los principios más importantes de REST, que veremos en un momento: cada petición debe ser independiente y completa por sí misma.

Ahora, hablemos de dos conceptos clave: recursos y representaciones. En REST, todo lo que la API maneja se modela como un "recurso". Un recurso es cualquier cosa con nombre a la que le puedes pedir información o sobre la que puedes actuar: una tarea, un usuario, un producto, un pedido. Cada recurso tiene una dirección única, su URL. Por ejemplo, en nuestra API de tareas, el recurso "tareas" vive en "/api/tareas", y una tarea específica, digamos la número 5, vive en "/api/tareas/5". Fíjate que la URL es un sustantivo, no un verbo: dice QUE cosa quieres, no QUE hacer con ella. Lo que quieres hacer se lo dices con el verbo HTTP, que es como el tono de voz con el que le hablas al mesero: "traeme" (GET), "agrega" (POST), "cambia" (PUT), "quita" (DELETE).

La "representación" es la forma en que el recurso viaja del servidor al cliente. El mismo recurso "tarea número 5" podría representarse como JSON, como XML o como texto plano, igual que la misma ensalada se puede servir en plato hondo o en plato plano: la ensalada es la misma, cambia la presentación. En la práctica, el formato universal de las APIs REST modernas es JSON, y por eso le dedicamos una sección completa más adelante. Cuando haces GET a "/api/tareas/5", el servidor te responde con la representación JSON de esa tarea: algo como {"id": 5, "título": "Comprar pan", "completada": false}. El recurso es la tarea; la representación es ese texto JSON que viaja por la red.

Los 6 principios REST explicados de forma sencilla

REST no es una tecnología que instalas; es una filosofía de diseño con seis principios. No necesitas memorizarlos como un examen, pero entenderlos te convierte en alguien que diseña APIs bien, no en alguien que solo copia ejemplos. Vamos uno por uno, en lenguaje de todos los días.

1. **Cliente-servidor.** La interfaz de usuario (el cliente) y los datos (el servidor) están separados y se comunican solo a través de la API. Esto permite que el mismo backend atienda a una página web, a una app de Android y a una app de iPhone al mismo tiempo, sin cambiar nada del servidor. Cada parte evoluciona por su cuenta: puedes rediseñar todo el frontend sin tocar el backend, y viceversa. Es como el restaurante: puedes cambiar toda la decoración del comedor sin remodelar la cocina.
2. **Sin estado (stateless).** Cada petición del cliente al servidor debe contener toda la información necesaria para ser entendida, sin depender de peticiones anteriores. El servidor no "recuerda" nada entre petición y petición. Volviendo al mesero: cada vez que le pides algo, le dices la mesa, el plato y los detalles, porque el mesero atiende a veinte mesas y no puede guardar en su cabeza el historial completo de cada una. En la práctica esto significa que si tu API necesita saber quien eres (autenticación), el cliente debe enviar su credencial en cada petición, normalmente en un encabezado. La ventaja es enorme: como el servidor no guarda estado de cada cliente, puede atender a miles de ellos y es fácil poner más servidores a ayudar cuando crece la demanda.
3. **Cacheable.** Las respuestas deben decir si se pueden guardar en cache o no. Si un dato no cambia seguido (por ejemplo, la lista de categorías de productos), el cliente o un intermediario puede guardarlo y no volver a pedirlo, lo que hace todo más rápido y reduce la carga del servidor. Es como cuando el mesero ya sabe que siempre pides agua natural al sentarte: no necesita ir a preguntar a la cocina cada vez. En HTTP esto se controla con encabezados de cache, y aunque en este curso no lo implementaremos a fondo, debes saber que existe y que diseñar pensando en cache es parte de ser un buen diseñador de APIs.

4. Interfaz uniforme. Este es el corazon de REST y el principio más importante para ti ahora. Significa que todas las APIs REST se usan de la misma manera: los recursos se identifican con URLs que son sustantivos, las acciones se expresan con los verbos HTTP estandar (GET, POST, PUT, DELETE), las respuestas usan JSON y los códigos de estado HTTP tienen significados acordados (200 salio bien, 404 no existe, etc.). Gracias a esta uniformidad, cuando aprendes a usar una API REST, sabes usarlas todas. Es cómo aprender a manejar: todos los coches tienen volante, acelerador y freno en los mismos lugares. Por eso dedicaremos tanto tiempo a los verbos y al diseno de rutas: son el "volante y los pedales" de las APIs.
5. Sistema en capas. El cliente no necesita saber si esta hablando directamente con el servidor final o con intermediarios (un balanceador de carga, un proxy, una capa de seguridad). Cada capa solo ve a la capa inmediata. Para ti como consumidor de la API, da igual si del otro lado hay un servidor o cien; tu pides del menu y recibes tu plato. Este principio es el que permite que las aplicaciones crezcan de cientos a millones de usuarios agregando capas intermedias sin cambiar el contrato con el cliente.
6. Código bajo demanda (opcional). El servidor puede enviar código ejecutable al cliente cuando sea necesario, como scripts de JavaScript. Es el único principio opcional, y en las APIs REST puras casi no se usa, así que lo mencionamos por completitud y seguimos adelante.

Con estos seis principios en la cabeza ya piensas como un diseñador de APIs. Ahora bajemos al detalle práctico: los verbos HTTP, que son las acciones que puedes pedir del menu.

Verbos HTTP y su correspondencia con CRUD

CRUD son las siglas de Create, Read, Update, Delete: crear, leer, actualizar y borrar. Son las cuatro operaciones fundamentales que se hacen con datos en cualquier sistema del mundo, desde una libreta de papel hasta la base de datos de un banco. Todo lo que una aplicación hace con datos cabe en esas cuatro operaciones: cuando te registras en un sitio, se CREA tu usuario; cuando ves tu perfil, se LEE; cuando cambias tu foto, se ACTUALIZA; cuando cierras tu cuenta, se BORRA. Si dominas el CRUD, dominas el 90 por ciento de lo que un backend hace en el día a día.

HTTP ya trae verbos pensados exactamente para estas operaciones, y REST los usa con significados precisos. Aquí esta la correspondencia que debes grabarte:

- GET significa leer. Pide una representación de un recurso sin modificar nada en el servidor. Es el verbo más usado y el más seguro: hacer GET dos veces seguidas da el mismo resultado y no cambia nada. Ejemplo: GET `"/api/tareas"` trae la lista de tareas; GET `"/api/tareas/5"` trae la tarea con id 5.
- POST significa crear. Envía datos al servidor para que cree un recurso nuevo. Ejemplo: POST a `"/api/tareas"` con `{"titulo": "Estudiar Node"}` crea una tarea nueva. Fíjate que POST va a la colección (`"/api/tareas"`), no a un id específico, porque el recurso aun no existe y el servidor le asignara su id.
- PUT significa actualizar (reemplazar). Envía la versión nueva y completa de un recurso que ya existe. Ejemplo: PUT a `"/api/tareas/5"` con `{"titulo": "Comprar pan integral", "completada": true}` reemplaza la tarea 5. PUT va a la URL del recurso específico, porque ya sabes cuál quieres cambiar.
- DELETE significa borrar. Elimina el recurso indicado. Ejemplo: DELETE a `"/api/tareas/5"` borra la tarea 5. Normalmente no lleva cuerpo en la petición: la URL ya dice todo lo necesario.

Hay un quinto verbo que verás mencionado: PATCH, que significa actualización parcial (cambiar solo un campo, como marcar `"completada": true` sin enviar todo lo demás). PATCH es útil en el mundo real, pero en este curso nos quedaremos con los cuatro grandes para mantener el proyecto claro. Lo importante es la idea: el verbo dice la acción, la URL dice el recurso.

Dos propiedades técnicas de los verbos que vale la pena conocer, aunque sea de pasada: GET es "seguro" (no modifica nada) e "idempotente" (repetirlo no cambia el resultado). PUT y DELETE son idempotentes (borrar dos veces la misma tarea deja el mismo estado final: la tarea no existe). POST no es idempotente: si envías dos veces el mismo POST de crear tarea, creas dos tareas. Esto no es trivia de examen; explica por qué, por ejemplo, debes tener cuidado con que el usuario no haga doble clic en el botón de "crear", porque cada clic es un POST y cada POST crea un recurso nuevo.

En Express, cada verbo tiene su método: `app.get()`, `app.post()`, `app.put()`, `app.delete()`. Ya los conoces de unidades anteriores; ahora les daras su uso más profesional. La línea más famosa de esta unidad, la que resume todo, es esta:

```
app.get("/api/tareas", (req, res) => res.json(tareas));
```

Leela en voz alta: "cuando llegue un GET a `/api/tareas`, responde con el JSON del arreglo `tareas`". Esa sola línea es una API REST funcionando: un recurso (`tareas`), una representación (JSON) y un verbo (GET para leer). Todo lo demás que veremos son extensiones de esa idea.

Diseno de rutas RESTful: el menu bien organizado

Ya sabes que la URL dice QUE recurso quieres y el verbo dice QUE hacer con el. Ahora veamos cómo se combinan: cada pareja de verbo más ruta es un "platillo del menu", un endpoint. Un endpoint es simplemente una combinación de verbo HTTP y URL que hace algo específico. Nuestra API de tareas, completa y bien diseñada, tiene estos cinco endpoints:

Endpoint 1: GET /api/tareas. Lee la colección completa. Devuelve el arreglo de todas las tareas en JSON. Código de estado 200. Es como decirle al mesero "traeme el menu de postres completo".

Endpoint 2: POST /api/tareas. Crea una tarea nueva. El cliente envia en el cuerpo de la petición los datos de la tarea (por ejemplo {"titulo": "Estudiar Node"}), el servidor le asigna un id único, la guarda y la devuelve creada. Código de estado 201 (qué significa "creado", lo veremos en la sección de códigos). Es como decir "agrega este platillo a mi orden".

Endpoint 3: GET /api/tareas/:id. Lee UNA tarea especifica. El ":id" es un parámetro de ruta: Express lo captura y te lo entrega en req.params.id. Si la tarea existe, la devuelve con 200; si no existe, responde 404 ("no encontrado"). Es como decir "traeme el platillo número 5 de mi orden".

Endpoint 4: PUT /api/tareas/:id. Actualiza (reemplaza) la tarea indicada. El cliente envia la versión nueva completa en el cuerpo. Si la tarea existe, se actualiza y se devuelve con 200; si no existe, 404. Es como decir "cambia mi ensalada número 3 por esta otra".

Endpoint 5: DELETE /api/tareas/:id. Borra la tarea indicada. Si existe, se elimina y se responde 200 (o 204, lo veremos); si no existe, 404. Es como decir "quita el postre de mi orden".

Fíjate en el patrón: la URL siempre habla del recurso ("tareas", en plural, como colección), nunca de la acción. Los errores típicos de principiante son poner verbos en la URL, como "/api/crearTarea" o "/api/borrarTarea/5". Eso rompe la interfaz uniforme: si cada API inventa sus propios verbos en las URLs, ya no hay un lenguaje comun. La forma RESTful es dejar que el verbo HTTP haga su trabajo: POST /api/tareas crea, DELETE /api/tareas/5 borra. La URL nombra la cosa; el verbo nombra la acción. Grabatelo: URLs con sustantivos en plural para colecciones, verbo HTTP para la acción.

El prefijo "/api" es una convención muy extendida: marca que estas rutas son la API (datos en JSON) y no páginas HTML. Así, en el mismo servidor puedes tener "/" que sirve tu página web y "/api/tareas" que sirve datos. Cuando crezca tu proyecto y tengas versiones, verás "/api/v1/tareas": la versión en la URL permite cambiar la API sin romper a los clientes viejos. No lo implementaremos aquí, pero ya sabes que existe y por qué.

Otro detalle de diseño: los parámetros de ruta (":id") sirven para identificar UN recurso, mientras que los query strings ("?completada=true") sirven para filtrar, ordenar o paginar una colección. Por ejemplo, GET "/api/tareas?completada=true" podría devolver solo las tareas completadas. En Express, los query strings llegan en req.query. Es una distinción útil: ":id" es "cual", "?filtro=" es "como la quiero". En los ejercicios de práctica implementarás un filtro por query para que quede clarísimo.

Y una regla de oro del diseño RESTful: la API debe ser predecible. Si alguien conoce tus endpoints de tareas, debería poder adivinar los de usuarios: GET /api/usuarios, POST /api/usuarios, GET /api/usuarios/:id, PUT /api/usuarios/:id, DELETE /api/usuarios/:id. Mismo menú, distinto restaurante. Esa predecibilidad es el premio de seguir las convenciones: tu API se vuelve fácil de aprender, fácil de documentar y fácil de usar.

JSON a fondo: el idioma entre apps

JSON significa "JavaScript Object Notation", notación de objetos de JavaScript. Es un formato de texto para representar datos estructurados, y es el idioma universal en el que las aplicaciones se hablan entre sí. La frase clave de esta clase es: "JSON es el idioma entre apps". Cuando tu frontend le pide datos a tu backend, viajan en JSON. Cuando una app de celular habla con un servidor, hablan JSON. Cuando dos empresas conectan sus sistemas, el contrato casi siempre es JSON. Nació de la sintaxis de objetos de JavaScript, pero hoy lo usan todos los lenguajes: Python, Java, PHP, Go, todos hablan JSON. Aprenderlo a fondo es aprender el esperanto del software moderno.

La sintaxis de JSON es deliberadamente simple. Un documento JSON es texto plano que representa UN valor, que puede ser un objeto (llaves {}) o un arreglo (corchetes []). Dentro van pares de "clave": valor. Las reglas estrictas que debes respetar:

- Las claves SIEMPRE van entre comillas dobles. {"nombre": "Ana"} es válido; {nombre: "Ana"} NO es JSON válido (aunque en JavaScript puro sí funciona).
- Los textos (strings) SIEMPRE van entre comillas dobles. "Hola" es válido; 'Hola' con comillas simples NO es JSON válido.
- Los tipos de datos permitidos son solo seis: string (texto), number (número, sin comillas, pueden ser enteros o decimales), boolean (true o false, sin comillas), null (vacío intencional, sin comillas), object ({}), y array ([]).
- NO se permiten comentarios, ni funciones, ni undefined, ni fechas como tipo nativo (las fechas se mandan como strings, por ejemplo "2026-09-19").
- No puede haber comas al final: {"a": 1,} es inválido. La última propiedad no lleva coma.

Ejemplo de una tarea en JSON:

```
{
  "id": 5,
  "titulo": "Comprar pan",
  "completada": false,
  "etiquetas": ["compras", "urgente"],
  "fechaLimite": null
}
```

Lee cada campo con su tipo: id es number, título es string, completada es boolean, etiquetas es un array de strings, fechaLimite es null (aun no tiene fecha). Seis tipos, cero sorpresas. Esa simplicidad es su poder: cualquier lenguaje puede leer y escribir estos seis tipos sin ambigüedad.

Ahora, la relación entre JSON y JavaScript, que es donde muchos se confunden. JSON es TEXTO; un objeto de JavaScript es una ESTRUCTURA en memoria. Se parecen muchísimo a simple vista, pero son cosas distintas: uno viaja por la red, el otro vive en tu programa. JavaScript te da dos herramientas para convertir entre ambos:

- `JSON.stringify(objeto)` convierte un objeto o arreglo de JavaScript en un string JSON. Lo usas cuando vas a ENVIAR datos: al guardar en un archivo, al responder una API, al mandar por la red.
- `JSON.parse(stringJSON)` convierte un string JSON en un objeto o arreglo de JavaScript. Lo usas cuando RECIBES datos: al leer un archivo, al recibir la respuesta de una API, al leer lo que mandó un cliente.

Mira el ciclo completo:

```
let tarea = { id: 5, titulo: "Comprar pan", completada: false };

// 1. Convertir a texto JSON para enviarlo o guardarlo
let texto = JSON.stringify(tarea);
console.log(texto); // '{"id":5,"titulo":"Comprar pan","completada":false}'
console.log(typeof texto); // "string": ahora es texto plano

// 2. Convertir de vuelta a objeto para trabajar con el
let recuperada = JSON.parse(texto);
console.log(recuperada.titulo); // "Comprar pan": otra vez objeto usable
console.log(typeof recuperada); // "object"
```


Este ciclo (objeto -> texto -> objeto) es el latido de toda API REST: el servidor toma datos de su memoria, los convierte a JSON con `stringify` (o con `res.json`, que lo hace por ti), los manda por la red como texto, y el cliente los convierte de vuelta a objetos con `parse` (o con `response.json()`, que lo hace por ti). Entender este ciclo es entender el 50 por ciento de cómo funciona la web moderna.

En Express, `res.json()` hace el `stringify` por ti y además pone el encabezado correcto (`Content-Type: application/json`) para que el cliente sepa que lo que recibe es JSON. Por eso `app.get("/api/tareas", (req, res) => res.json(tareas))` es tan elegante: le pasas el arreglo y Express se encarga de convertirlo y etiquetarlo. Del otro lado, para LEER el JSON que envía el cliente (por ejemplo, en un POST), necesitas el middleware `express.json()`, que toma el texto JSON del cuerpo de la petición y te lo entrega ya convertido en `req.body`. Sin ese middleware, `req.body` sería `undefined` y tu POST no recibiría nada. Es un error clásico: "mi POST no recibe datos" casi siempre significa "olvide `app.use(express.json())`".

`JSON.stringify` tiene dos trucos útiles. El primero: `JSON.stringify(objeto, null, 2)` genera el JSON "bonito", con indentación de 2 espacios, ideal para guardar archivos legibles por humanos o para depurar. El segundo: `stringify` ignora silenciosamente los valores `undefined` y las funciones. Si tu objeto tiene una función, simplemente no aparece en el JSON. Esto es importante: JSON solo transporta DATOS, no comportamiento. No puedes mandar una función por JSON; solo datos.

`JSON.parse` tiene un peligro: si el texto no es JSON válido, LANZA un error y tu programa se cae si no lo atrapas. Por eso, cuando parseas datos que vienen de fuera (un archivo que edita un humano, una petición de un cliente), envuélvelo en `try/catch`. En nuestro proyecto, al leer el archivo de persistencia usaremos `try/catch` alrededor de `JSON.parse`: si el archivo está vacío o corrupto, empezamos con un arreglo vacío en lugar de tronar el servidor. Esa es la diferencia entre un servidor robusto y uno frágil.

Por qué JSON ganó y es "el idioma entre apps"? Tres razones. Una: es texto plano, así que viaja por cualquier protocolo, se lee en cualquier editor y se depura a simple vista. Dos: es ligero y fácil de parsear; las computadoras lo convierten a estructuras nativas en microsegundos. Tres: es independiente del lenguaje pero nació de JavaScript, el lenguaje de la web, así que el navegador lo habla de forma nativa. XML hacía lo mismo pero con mucho más ruido (etiquetas de apertura y cierre para todo); JSON dijo lo mismo con la mitad de caracteres. La industria votó con los pies y hoy JSON es el estándar.

Cuidado con tres confusiones comunes. Una: JSON no es lo mismo que un objeto literal de JavaScript, aunque se parezcan. La diferencia visible: en JSON las claves llevan comillas dobles obligatorias. Dos: los números en JSON no llevan comillas; `{"id": "5"}` y `{"id": 5}` son

cosas distintas (string vs number) y mezclarlos causa bugs sutiles, como que el id 5 no coincida con el id "5" al buscar. Tres: JSON no tiene concepto de "indefinido": si un campo no tiene valor, se usa null o simplemente se omite el campo. undefined es de JavaScript, no de JSON.

Códigos de estado HTTP: el semaforo de la API

Cada respuesta HTTP lleva un número de tres digitos que dice, de un vistazo, que paso con la petición. Son el "semaforo" de la API: el cliente no necesita leer todo el cuerpo para saber si su pedido salio bien, mal o más o menos. Los códigos se agrupan por familias según su primer dígito: 2xx significa éxito, 4xx significa error del cliente (pidió mal), 5xx significa error del servidor (algo se rompio adentro). Para nuestra API REST nos bastan cinco códigos, pero debes usarlos con precisión porque son parte del contrato:

- 200 OK: "salio bien". Es la respuesta estandar de éxito para GET, PUT y DELETE. Cuando pides la lista de tareas y todo sale bien, respondes 200 con el JSON. En Express, `res.json()` usa 200 por defecto, así que la mayoría de tus respuestas exitosas ya lo tienen sin que hagas nada.
- 201 Created: "salio bien Y se creó algo nuevo". Es el código correcto para un POST exitoso. Le dice al cliente "no solo te entendi: ya existe el recurso que pediste". En Express lo usas así: `res.status(201).json(nuevaTarea)`. Usar 201 en lugar de 200 para creaciones es una de esas senales de profesionalismo: cualquiera que consuma tu API sabra distinguir "aquí tienes datos" de "acabo de crear esto".
- 204 No Content: "salio bien y no tengo nada más que decirte". Se usa tipicamente en DELETE exitosos cuando no quieres devolver cuerpo. En Express: `res.status(204).send()`. Ojo: con 204 NO debes enviar cuerpo; algunos clientes se confunden si lo haces. En nuestro proyecto devolveremos 200 con un mensaje en el DELETE para que sea más didactico, pero debes saber que 204 existe y es la opción "elegante".
- 400 Bad Request: "me pediste algo mal formado". El cliente envió datos invalidos: falta el título, el título no es texto, el JSON venia roto. Es TU trabajo como backend detectar esto (validación, siguiente sección) y responder 400 con un mensaje que explique que está mal. Nunca dejes que datos invalidos entren a tu sistema en silencio: responde 400 y di por qué.
- 404 Not Found: "eso que pides no existe". El cliente pidió GET `/api/tareas/999` y no hay tarea 999. También aplica a PUT y DELETE sobre ids inexistentes. Responder 404 con un mensaje claro ("Tarea no encontrada") es mucho mejor que responder 200 con un null misterioso o, peor, que el servidor truene.

Hay más códigos en el mundo real (401 no autorizado, 403 prohibido, 500 error interno), y los verás cuando agregues autenticación. Por ahora, con estos cinco tu API habla correctamente. La regla de oro: el código de estado cuenta la HISTORIA de la petición; el cuerpo JSON cuenta los DETALLES. Un cliente bien hecho primero mira el código (¿2xx? sígo; ¿4xx? le muestro el error al usuario) y luego lee el cuerpo.

En Express, el patrón es siempre `res.status(código).json(datos)`. Fíjate que `status()` va ANTES de `json()` o `send()`: primero dices el código, luego mandas el cuerpo. Si lo pones al revés, no funciona como esperas. Y recuerda: si no llamas a `status()`, Express asume 200, lo cual está bien para lecturas y actualizaciones exitosas.

Validación de datos de entrada: el portero del restaurante

La validación es el "portero" de tu API: revisa lo que el cliente quiere meter antes de dejarlo pasar. Si en un POST a `/api/tareas` el cliente manda `{}` (sin título), o `{"título": 123}` (título numerico), o `{"título": ""}` (título vacío), tu API no debe crear una tarea rota: debe responder 400 con un mensaje que explique el problema. La regla de seguridad más importante del backend es: NUNCA confíes en los datos que vienen de fuera. El cliente puede ser tu propio frontend (confiable) o un atacante con una herramienta que manda peticiones a mano (nada confiable). Tu validación protege tus datos en ambos casos.

Que validar en nuestra API de tareas:

1. Que el título exista: si `req.body.título` es `undefined`, responde 400 con `{error: "El título es obligatorio"}`.
2. Que el título sea texto: si `typeof req.body.título !== "string"`, responde 400.
3. Que el título no este vacío ni sea puro espacio: si `req.body.título.trim() === ""`, responde 400. El `trim()` quita espacios al inicio y final; un título de " " no es un título real.
4. En el PUT, que "completada", si viene, sea booleano: si alguien manda `{"completada": "si"}`, responde 400. Los tipos importan, como vimos en la sección de JSON.

El patrón de código es una función validadora que devuelve un mensaje de error o null si todo está bien:

```
function validarTarea(datos) {
  if (!datos.titulo || typeof datos.titulo !== "string") {
    return "El titulo es obligatorio y debe ser texto";
  }
  if (datos.titulo.trim() === "") {
    return "El titulo no puede estar vacio";
  }
  return null; // todo bien
}

// En el endpoint POST:
app.post("/api/tareas", (req, res) => {
  const error = validarTarea(req.body);
  if (error) {
    return res.status(400).json({ error });
  }
  // ... crear la tarea
});
```

Fíjate en el "return" antes de `res.status(400)`: es crucial. Sin ese return, el código seguiría ejecutándose después de responder el error e intentaría crear la tarea de todos modos, o peor, intentaría responder dos veces y Express lanzaría el famoso error "Cannot set headers after they are sent". Patrón grabado: valida, si hay error responde y RETORNA.

La validación también aplica a los ids de la URL. Si alguien pide `GET /api/tareas/hola`, `req.params.id` será "hola". Tu código debe buscar la tarea con ese id, no encontrarla y responder 404. Eso ya es un manejo correcto. Pero si usas ids numericos, conviene convertir con `Number(req.params.id)` y, si el resultado es `NaN` (no es número), responder 400. Así distingues "pediste mal el id" (400) de "el id está bien formado pero no existe" (404). Es un detalle fino que muestra dominio.

IDs únicos: `Date.now()` y el contador

Cada recurso necesita un identificador único, su "CURP": un valor que lo distingue de todos los demás para siempre. Sin ids únicos no puedes pedir "la tarea 5", ni actualizarla, ni borrarla. Hay dos estrategias sencillas para generarlos en nuestro proyecto:

Estrategia 1: `Date.now()`. Devuelve el número de milisegundos desde el 1 de enero de 1970 (el "tiempo Unix" en milisegundos). Cada vez que lo llamas obtienes un número distinto, porque el tiempo siempre avanza: `Date.now()` te da algo como 1726684800123. Es fácil, no necesitas guardar ningún contador y funciona bien para proyectos pequeños o prototipos. Su debilidad: si creas dos tareas en el MISMO milisegundo (raro, pero posible en operaciones automáticas), obtendrías el mismo id. Para nuestro proyecto es aceptable, pero debes conocer el límite.

Estrategia 2: un contador en memoria. Guardas una variable "siguienteld" que empieza en 1; cada vez que creas una tarea, le asignas el valor actual y lo incrementas. Los ids quedan 1, 2, 3, limpios y predecibles, perfectos para aprender y para probar con herramientas. Su debilidad: si reinicias el servidor, el contador vuelve a 1 y podrías repetir ids... a menos que, al iniciar, calcules el siguiente id a partir de los datos guardados (por ejemplo, el máximo id existente más 1). En nuestro proyecto con persistencia haremos exactamente eso: al arrancar, leemos el archivo y ajustamos el contador.

En el mundo profesional se usan UUIDs (identificadores larguísimo y aleatorios, como "550e8400-e29b-41d4-a716-446655440000") o los ids automáticos de la base de datos. Pero para tu primer backend, el contador es perfecto: simple, legible y fácil de depurar. Lo importante es el concepto: el id lo asigna EL SERVIDOR, nunca el cliente. Si dejaras que el cliente eligiera el id, dos clientes podrían elegir el mismo y romperían tu base de datos. El servidor manda en los ids; el cliente solo manda los datos.

Persistencia en archivo JSON: que los datos sobrevivan al reinicio

Si guardas las tareas solo en un arreglo en memoria, todo funciona... hasta que reinicias el servidor. Al apagar Node, la memoria se borra y tus tareas desaparecen. Eso está bien para practicar, pero un backend real debe conservar sus datos. La solución profesional es una base de datos (tema de tu siguiente curso, el de SQL), pero hay un paso intermedio perfecto para este proyecto: guardar el arreglo en un archivo JSON en disco usando el módulo `fs` que ya dominas. Es una "base de datos de juguete" honesta y muy didáctica: te enseña el concepto de persistencia sin la complejidad de instalar un motor de base de datos.

La idea es simple y tiene dos momentos:

Momento 1: al ARRANCAR el servidor, lees el archivo (por ejemplo "tareas.json"). Si existe y tiene JSON válido, lo conviertes con `JSON.parse` y ese es tu arreglo inicial. Si no existe (primera vez) o está corrupto, empiezas con un arreglo vacío. Todo esto envuelto en `try/catch` para que el servidor nunca truene al arrancar.

Momento 2: cada vez que los datos CAMBIAN (crear, actualizar o borrar una tarea), escribes el arreglo completo al archivo con `fs.writeFileSync` y `JSON.stringify(tareas, null, 2)`. El "null, 2"

genera el JSON bonito con indentación, para que puedas abrir el archivo y leerlo como humano. Usamos la versión sincrónica (`writeFileSync`) a propósito: es simple, y como estas escrituras son pequeñas y poco frecuentes, no hay problema de rendimiento. (En un servidor con miles de escrituras por segundo usarías otra estrategia, pero no es nuestro caso.)

El código central:

```
const fs = require("fs");
const ARCHIVO = "./tareas.json";

// Cargar al arrancar
let tareas = [];
try {
  const texto = fs.readFileSync(ARCHIVO, "utf8");
  tareas = JSON.parse(texto);
} catch (e) {
  tareas = []; // no existe o esta corrupto: empezamos de cero
}

// Guardar cada vez que cambian los datos
function guardar() {
  fs.writeFileSync(ARCHIVO, JSON.stringify(tareas, null, 2));
}

// Y en cada endpoint que modifica:
app.post("/api/tareas", (req, res) => {
  // ... validacion ...
  const nueva = { id: siguienteId++, titulo: req.body.titulo.trim(), completada: false };
  tareas.push(nueva);
  guardar(); // persistir el cambio
  res.status(201).json(nueva);
});
```

Tres detalles finos. Uno: el contador de ids debe sobrevivir también. Al cargar, calculalo como el máximo id existente más 1 (o 1 si no hay tareas). Así nunca repites ids entre reinicios. Dos: la lectura inicial usa `readFileSync` porque ocurre UNA vez al arrancar, antes de atender peticiones; no hay nada que bloquear. Tres: si dos peticiones de escritura llegaran exactamente al mismo tiempo podría haber una condición de carrera, pero Node.js atiende una petición a la vez en su hilo principal y nuestras escrituras son instantaneas, así que para este proyecto es seguro.

Abre el archivo tareas.json después de crear un par de tareas y miralo: ahí están tus datos, en texto plano, legibles. Ese archivo ES tu base de datos por ahora. Cuando en el siguiente curso aprendas SQL, entenderas que una base de datos hace lo mismo pero con superpoderes: búsquedas rápidas, relaciones, seguridad y millones de registros. El concepto de persistencia, ya lo tienes.

CORS: por qué el navegador bloquea tu fetch y como permitirlo

Aquí viene uno de los errores más confusos para principiantes, así que prestame toda tu atención. Imagina este escenario: tu API corre en `http://localhost:3000` y tu frontend (un archivo HTML con JavaScript) lo abres en el navegador, quizá servido desde otro puerto o como archivo local. Tu código hace `fetch("http://localhost:3000/api/tareas")...` y el navegador lo BLOQUEA con un error de CORS en la consola. Tu API funciona perfecto si la pruebas con una herramienta o con curl, pero el navegador se niega. ¿Que esta pasando? ¿Tu código está mal? No. Es el navegador protegiendote.

CORS significa "Cross-Origin Resource Sharing", compartir recursos entre orígenes cruzados. Un "origen" es la combinación de protocolo + dominio + puerto. `http://localhost:3000` y `http://localhost:5500` son orígenes DISTINTOS (cambia el puerto). La regla de seguridad del navegador, llamada "política del mismo origen", dice: una página web solo puede pedir datos al MISMO origen del que vino, a menos que el servidor diga explícitamente "permite peticiones de otros orígenes". ¿Por qué existe esta regla? Imagínate que estas logueado en tu banco y visitas una página maliciosa: sin esta protección, esa página podría hacer fetch al sitio del banco usando tu sesión y robar tus datos. El navegador bloquea esas peticiones cruzadas por defecto para protegerte. Es una buena regla; solo que durante el desarrollo te estorba.

La solución tiene dos caminos. Camino fácil: sirve tu frontend DESDE tu API. Es decir, que Express sirva los archivos estáticos (HTML, JS, CSS) con `express.static`, así frontend y API comparten el mismo origen (`http://localhost:3000`) y no hay problema de CORS. Es lo que haremos en el ejemplo de la sección "Ejemplos paso a paso": el mismo servidor entrega la página y los datos. Simple y sin configuración extra.

Camino flexible: permitir CORS explícitamente en tu API. El servidor responde con un encabezado especial, `Access-Control-Allow-Origin`, que dice "acepto peticiones de estos orígenes". La forma más sencilla es instalar el paquete "cors" de npm y usarlo como middleware:

```
const cors = require("cors");
app.use(cors()); // permite peticiones de cualquier origen (solo para desarrollo)
```

Con `app.use(cors())`, tu API acepta fetch desde cualquier página. Ojo: "cualquier origen" está bien mientras desarrollas en tu máquina, pero en producción debes restringirlo a los dominios de tu frontend real, porque abrir tu API al mundo entero es un riesgo de seguridad. El paquete `cors` permite configurarlo: `cors({ origin: "https://miapp.com" })`. Para este curso, con saber que existe el paquete y que en desarrollo puedes usarlo abierto, es suficiente.

Como diagnosticar un problema de CORS: el síntoma es inconfundible. Tu fetch falla, en la consola del navegador aparece un error que menciona "CORS", "Access-Control-Allow-Origin" o "has been blocked by CORS policy", y en la pestaña de red verás la petición marcada en rojo. Si ves eso, no es tu lógica ni tu JSON: es CORS. Soluciones en orden: 1) sirve el frontend desde el mismo servidor (`express.static`), 2) agrega el middleware `cors`, 3) verifica que no sea otro error disfrazado (a veces un 404 también se ve raro en la consola; revisa la pestaña de red para ver el código real).

Un detalle técnico que te ahorrara confusión: para peticiones "simples" (GET, o POST con JSON... bueno, POST con JSON ya no es tan simple) el navegador a veces envía primero una petición de "pre-vuelo" (preflight) con el verbo `OPTIONS`, preguntando "¿me permites?". El middleware `cors` responde a esos `OPTIONS` automáticamente. Si implementaras CORS a mano (posible pero innecesario), tendrías que manejar `OPTIONS` tú mismo. Usa el paquete y olvídate del preflight.

Fetch desde el frontend: tu página habla con tu API

Hasta ahora el "cliente" de tu API eras tu, probando con el navegador o con herramientas. Ahora el cliente será una página web real: tu propio frontend hablando con tu propio backend. La herramienta es `fetch`, la función del navegador para hacer peticiones HTTP desde JavaScript. Ya la conoces de haber consumido APIs ajenas; ahora la usaras para consumir la TUYA. La línea emblemática de esta clase es:

```
fetch("http://localhost:3000/api/tareas").then(r => r.json())
```

Leela por partes: `fetch()` hace la petición GET a tu API (fetch usa GET por defecto) y devuelve una promesa que se cumple con el objeto "response". Ese `.then(r => r.json())` toma la respuesta y la convierte de texto JSON a objeto de JavaScript (`r.json()` también devuelve una promesa, por eso se encadena otro `then`). Al final tienes tu arreglo de tareas, listo para mostrar en la página. Es el mismo ciclo objeto -> texto -> objeto que vimos en JSON, ahora cruzando la red entre tus dos programas.

Fetch con `async/await` se lee aun más claro, y es lo que usaremos en el frontend del proyecto:


```

async function cargarTareas() {
  try {
    const respuesta = await fetch("http://localhost:3000/api/tareas");
    if (!respuesta.ok) {
      throw new Error("Error del servidor: " + respuesta.status);
    }
    const tareas = await respuesta.json();
    mostrarTareas(tareas);
  } catch (error) {
    console.log("No se pudieron cargar las tareas:", error.message);
    // Aqui mostrarias un mensaje amable en la pagina
  }
}

```

Tres lecciones en este código. Una: fetch SOLO rechaza su promesa si hay un error de RED (el servidor esta apagado, no hay internet). Si el servidor responde 404 o 500, fetch considera que "la petición salio" y te entrega la respuesta: por eso SIEMPRE debes revisar respuesta.ok (qué es true para códigos 2xx) y lanzar tu propio error si no lo es. Este es el error número uno de quienes empiezan con fetch: pensar que un 404 entrara al catch. No entra. Revisa .ok. Dos: el try/catch envuelve todo, incluyendo la conversión a JSON, porque cualquiera de esos pasos puede fallar. Tres: en el catch no basta con console.log en una app real: debes mostrarle algo al usuario en la página ("No se pudo conectar con el servidor"), porque el usuario no tiene la consola abierta.

Para CREAR, ACTUALIZAR y BORRAR desde fetch, debes especificar el verbo y el cuerpo:

```

// CREAR (POST): hay que decir el metodo y mandar el JSON en el cuerpo
async function crearTarea(titulo) {
  const respuesta = await fetch("http://localhost:3000/api/tareas", {
    method: "POST",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify({ titulo })
  });
  if (!respuesta.ok) throw new Error("No se pudo crear");
  return await respuesta.json(); // la tarea creada, con su id
}

// BORRAR (DELETE): solo el metodo, la URL ya dice cual
async function borrarTarea(id) {
  const respuesta = await fetch("http://localhost:3000/api/tareas/" + id, {
    method: "DELETE"
  });
  if (!respuesta.ok) throw new Error("No se pudo borrar");
}

// ACTUALIZAR (PUT): metodo mas cuerpo con los datos nuevos
async function actualizarTarea(id, datos) {
  const respuesta = await fetch("http://localhost:3000/api/tareas/" + id, {
    method: "PUT",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify(datos)
  });
  if (!respuesta.ok) throw new Error("No se pudo actualizar");
  return await respuesta.json();
}

```

El encabezado "Content-Type: application/json" es OBLIGATORIO cuando envías JSON: le dice al servidor "lo que te mandó en el cuerpo es JSON, interpretalo como tal". Sin ese encabezado, el middleware `express.json()` ignora el cuerpo y `req.body` llega `undefined`. Es el error clásico gemelo del "olvide `express.json()`": uno es del lado del servidor, el otro del lado del cliente. Cuando un POST "no recibe datos", revisa AMBOS: ¿el servidor tiene `app.use(express.json())`? ¿el fetch manda el Content-Type correcto? Nueve de cada diez veces, el problema es uno de esos dos.

Manejo de errores de red en el frontend, resumido en reglas prácticas: envuelve cada llamada en try/catch; revisa respuesta.ok después de cada fetch; muestra mensajes en la página (no solo en consola) cuando algo falle; y considera estados de "cargando" (un texto "Cargando tareas..." mientras esperas) para que el usuario no piense que la página se trabó. Un frontend que maneja bien sus errores se siente profesional aunque sea simple; uno que deja la pantalla en blanco cuando falla la red se siente roto aunque su código sea correcto.

Armar el frontend mínimo: HTML y JS que consumen tu API

Un "frontend mínimo" no necesita frameworks ni diseño espectacular: necesita una página que liste las tareas, un formulario para crearlas, botones para completarlas y borrarlas, y el JavaScript que conecta todo con tu API via fetch. El patrón es el mismo ciclo que ya conoces de proyectos anteriores (estado -> mostrar -> escuchar -> actualizar), solo que ahora "actualizar" significa "hablar con la API y luego refrescar".

La estructura del HTML es sencilla: un título, un formulario con un input de texto y un botón "Agregar", y una lista vacía (ul) donde JavaScript pintara las tareas. Cada tarea pintada lleva su texto, una casilla o botón para marcarla como completada y un botón para borrarla. Nada más. La magia está en el JavaScript:

1. Al cargar la página, llamas a cargarTareas(): fetch GET, conviertes a JSON y pintas la lista.
2. La función pintarTareas(tareas) vacía la lista y crea un elemento por cada tarea, con sus botones. Cada botón lleva el id de la tarea para saber a cual afecta.
3. El formulario escucha el evento submit: evita la recarga (preventDefault), lee el título, hace fetch POST, y si sale bien vuelve a cargar la lista.
4. El botón de completar hace fetch PUT con {completada: true} (o alternando el valor actual) y recarga.
5. El botón de borrar hace fetch DELETE y recarga.

El truco arquitectónico: después de CADA operación que cambia datos (crear, actualizar, borrar), vuelves a pedir la lista completa al servidor y la repintas. Es simple y siempre correcto: la pantalla refleja exactamente lo que el servidor tiene. En apps grandes se optimiza actualizando solo lo necesario, pero para tu primer full stack, "recargar y repintar" es la estrategia correcta: simple, predecible y fácil de depurar.

Donde vive este frontend? Como vimos en CORS, lo más fácil es que Express sirva los archivos estáticos: pones tu index.html (y tu app.js del frontend) en una carpeta "public", agregas app.use(express.static("public")) en tu servidor, y abres http://localhost:3000 en el

navegador. Mismo origen, cero problemas de CORS, y tu proyecto queda autocontenido: un solo servidor que entrega la página Y los datos. Esa es la arquitectura de nuestro ejemplo en la siguiente sección.

Despliegue básico: subir tu API a internet

"Despliegue" (deploy) significa poner tu aplicación en un servidor en internet para que cualquiera pueda usarla, no solo tu en tu máquina. Pasar de "funciona en mi localhost" a "funciona en internet" es un rito de iniciación: tu API deja de ser un ejercicio y se convierte en un servicio real. No vamos a hacer un despliegue paso a paso aquí (cada plataforma cambia sus pantallas cada año), pero debes entender los conceptos, porque son los mismos en todas partes.

Concepto 1: tu código debe escuchar en el puerto que te asignen. En tu máquina usas el puerto 3000 porque tu lo elegiste. En un servidor en la nube, la plataforma te asigna un puerto dinámico y te lo pasa en una variable de entorno llamada PORT. Si tu código tiene el puerto fijo en 3000, no funcionara allá. La solución es una línea que ya debes usar desde ahora:

```
const PORT = process.env.PORT || 3000;  
app.listen(PORT, () => console.log("Servidor en puerto " + PORT));
```

Lee: "usa el puerto que me diga el entorno, y si no me dice nada (estoy en mi máquina), usa el 3000". process.env es el objeto de Node.js con las variables de entorno del sistema. Esta línea es obligatoria en cualquier proyecto que aspire a salir de tu computadora. Grabala.

Concepto 2: las variables de entorno. Son configuraciones que cambian según donde corre tu app (puerto, claves, URLs) y que NO deben estar escritas en tu código, porque tu código se comparte (por ejemplo, en GitHub) y las claves no deben compartirse. En desarrollo se usa un archivo ".env" (que nunca se sube al repositorio) con pares CLAVE=valor, y el paquete "dotenv" para leerlo. En producción, la plataforma te da un panel para definir esas variables. La idea es una sola: el código lee process.env.MI_CLAVE; el valor lo pone el entorno, no el código.

Concepto 3: opciones gratuitas. Existen plataformas con planes gratuitos para hospedar proyectos pequeños de Node.js (los nombres y condiciones cambian con el tiempo, así que menciónalas de forma genérica: busca "free Node.js hosting" actualizado al año en que leas esto). Todas funcionan parecido: conectas tu repositorio de código, les dices el comando de arranque ("node servidor.js" o "npm start") y ellas te dan una URL pública. El flujo típico es: tu código en GitHub -> la plataforma lo toma -> lo instala (npm install) -> lo arranca -> te da https://tu-app.algo.com. Tu API ya está en internet.

Checklist antes de publicar (repasalo como piloto antes de despegar):

1. El puerto sale de process.env.PORT con fallback a 3000.
2. No hay claves ni secretos escritos en el código (todo por variables de entorno).
3. Las dependencias están en package.json (la plataforma corre npm install; si algo solo está instalado en tu máquina, allá no existirá).
4. Tienes un script de arranque claro ("start": "node servidor.js" en package.json).
5. El CORS esta configurado para tu frontend real (no abierto al mundo entero).
6. Probaste los 5 endpoints en local y todos responden los códigos correctos.
7. Los archivos que no deben subirse están ignorados (un archivo .gitignore con node_modules y .env, cuando uses Git).

Si marcas las siete, tu despliegue tiene el 95 por ciento de probabilidades de salir bien a la primera. El 5 por ciento restante son los detalles de cada plataforma, y para eso están sus documentaciones.

Errores comunes y sus soluciones

Esta sección es oro puro: la lista de los errores que VAS a cometer (todos los cometemos) y como salir de cada uno rápido.

Error 1: CORS bloquea mi fetch. Sintoma: en la consola del navegador, error mencionando CORS o Access-Control-Allow-Origin; la API funciona si la pruebas directo. Solución: sirve el frontend desde el mismo servidor con express.static, o instala y usa el middleware cors. No es tu lógica; es seguridad del navegador.

Error 2: "Olvide el Content-Type". Sintoma: el POST llega al servidor pero req.body es undefined o vacío. Solución en dos frentes: el cliente debe mandar headers: {"Content-Type": "application/json"} en el fetch, y el servidor debe tener app.use(express.json()) antes de las rutas. Revisa ambos; el culpable suele ser uno solo.

Error 3: IDs duplicados o que se repiten al reiniciar. Sintoma: creas tareas, reinicias el servidor y las nuevas tareas repiten ids de las viejas; luego PUT o DELETE afectan a la tarea equivocada. Solución: al arrancar, calcula siguienteId como el máximo id guardado más 1 (o 1 si no hay tareas). Nunca reinicies el contador a 1 si ya hay datos persistidos.

Error 4: no validar y el servidor truena o guarda basura. Sintoma: POST sin título crea tareas con título undefined; o el servidor lanza un error raro al hacer `.trim()` sobre undefined. Solución: valida SIEMPRE antes de usar los datos (la función `validarTarea` de la sección correspondiente), responde 400 y haz return. La validación va primero, la lógica después.

Error 5: "Cannot set headers after they are sent". Sintoma: ese mensaje en la consola del servidor. Significa que tu código intentó responder DOS veces a la misma petición (por ejemplo, respondiste el 400 de validación pero no hiciste return y luego intentaste responder 201). Solución: cada vez que respondes dentro de un condicional, pon return delante. Un endpoint, una respuesta.

Error 6: el archivo JSON se corrompe y el servidor no arranca. Sintoma: al iniciar, `JSON.parse` lanza error. Solución: el try/catch al cargar ya lo maneja (empieza con arreglo vacío), pero además no editas `tareas.json` a mano mientras el servidor corre, y si se corrompe, bórralo y deja que se regenere. En desarrollo, borrar el archivo es un "reset" válido.

Error 7: el frontend no se actualiza después de crear o borrar. Sintoma: haces clic en "Agregar", la API crea la tarea (la ves si recargas), pero la lista en pantalla no cambia. Solución: después de cada operación de escritura, vuelve a llamar a `cargarTareas()` y repinta. O olvidaste ese paso, o tu función de pintar no limpia la lista antes de repintar (y entonces verías duplicados).

Error 8: el servidor no arranca porque el puerto está ocupado. Sintoma: error `EADDRINUSE` al hacer `node servidor.js`. Significa que ya hay otro proceso usando el puerto 3000 (quizá tu servidor anterior sigue corriendo en otra terminal). Solución: cierra el otro proceso o usa otro puerto. En desarrollo es de lo más común: antes de arrancar, verifica que no tengas un servidor viejo corriendo.

Graduación: lo que dominas y lo que sigue

Llegamos a la clase 30, y esta sección es tu ceremonia de graduación. Vamos a decir en voz alta, sin modestia falsa, todo lo que dominas después de este curso:

Dominas los módulos de Node.js: sabes organizar código en archivos, exportar con `module.exports` e importar con `require`, y entiendes por qué el código modular es código profesional. Dominas el sistema de archivos: lees, escribes y persistes datos con `fs`. Dominas los eventos y lo asíncrono: callbacks, promesas y `async/await` ya no te asustan; entiendes el event loop lo suficiente para no bloquearlo. Dominas Express: levantas servidores, defines rutas, usas middleware como `express.json()` y `express.static`, y capturas parámetros de ruta y query strings. Y ahora dominas las APIs REST: diseñas recursos con URLs de sustantivos, usas los verbos HTTP con su significado correcto (GET leer, POST crear, PUT actualizar,

DELETE borrar), hablas JSON con fluidez en ambas direcciones (stringify y parse), respondes con los códigos de estado adecuados (200, 201, 204, 400, 404), validas la entrada, generas ids únicos, persistes en archivo, resuelves CORS y conectas un frontend con fetch. Eso, en una línea: sabes construir un backend completo y funcional.

Tu proyecto final lo demuestra: una API de tareas con CRUD completo, persistencia en JSON y un frontend que la consume. Eso no es un ejercicio de juguete: es la arquitectura real de miles de aplicaciones en producción, solo que en versión pequeña y sin base de datos. Cuando alguien te pregunte "¿sabes backend?", tu respuesta ya no es teoría: "construí una API REST con Node.js y Express, con CRUD, validación y persistencia, y un frontend que la consume". Esa frase vale en entrevistas, en freelancing y en tus propios proyectos.

¿Y que sigue? La ruta natural, sin rodeos:

Paso 1: bases de datos con SQL. Tu persistencia en archivo JSON fue el entrenamiento; ahora toca lo profesional: un motor de base de datos real (tablas, consultas SELECT, INSERT, UPDATE, DELETE, relaciones). El curso de SQL de lucasegaacademy.com es tu siguiente parada, y llegarás con ventaja porque ya entiendes qué es persistir, que es un id único y que es validar.

Paso 2: autenticación. Que los usuarios se registren, inicien sesión y que tu API sepa quien es quien en cada petición (recuerda el principio "sin estado": credenciales en cada petición). Es el siguiente superpoder de backend y el tema que convierte tus APIs en aplicaciones multiusuario de verdad.

Paso 3: más proyectos, cada vez más grandes. Un blog con API, una tienda con carrito, un clon simplificado de tu app favorita. Cada proyecto te va a doler un poco y te va a enseñar mucho: esa es la fórmula y no hay atajos.

Sigue en lucasegaacademy.com: este curso de Node.js es uno de los 22 cursos de la academia, y el ecosistema está diseñado para llevarte de principiante a desarrollador completo, con cada curso conectando con los demás. Explora el curso de SQL, repasa los de JavaScript cuando necesites reforzar, y sigue construyendo. Felicidades, graduado: dominas módulos, Express y APIs REST. El backend ya no es un misterio para ti. Ahora ve por las bases de datos.

Ejemplos paso a paso

Ejemplo 1: API REST de tareas completa (servidor.js)

Este es el ejemplo central de la unidad: el servidor completo con los 5 endpoints, validación, ids únicos y persistencia en archivo JSON. Léelo despacio, bloque por bloque; cada bloque usa algo que ya aprendiste en el curso. Guárdalo como `servidor.js`, corre `"npm install express"` una vez en la carpeta, y arrancalo con `"node servidor.js"`.

```
const express = require("express");
const fs = require("fs");
const path = require("path");

const app = express();
const PORT = process.env.PORT || 3000;
const ARCHIVO = path.join(__dirname, "tareas.json");

// ----- Middleware -----
app.use(express.json()); // para leer el JSON que mandan los clientes
app.use(express.static(path.join(__dirname, "public"))); // sirve el frontend

// ----- Cargar datos al arrancar (persistencia) -----
let tareas = [];
try {
  const texto = fs.readFileSync(ARCHIVO, "utf8");
  const datos = JSON.parse(texto);
  if (Array.isArray(datos)) {
    tareas = datos;
  }
} catch (e) {
  tareas = []; // primera vez o archivo corrupto: empezamos de cero
}

// El siguiente id nunca se repite, ni siquiera entre reinicios
let siguienteId = tareas.length > 0
  ? Math.max(...tareas.map(t => t.id)) + 1
  : 1;

function guardar() {
  fs.writeFileSync(ARCHIVO, JSON.stringify(tareas, null, 2));
}

// ----- Validacion -----
function validarTarea(datos) {
  if (!datos || typeof datos.titulo !== "string") {
    return "El titulo es obligatorio y debe ser texto";
  }
}
```



```

    if (datos.titulo.trim() === "") {
        return "El titulo no puede estar vacio";
    }
    return null;
}

function buscarIndice(id) {
    return tareas.findIndex(t => t.id === id);
}

// ----- Endpoints -----

// 1. GET /api/tareas -> leer todas
app.get("/api/tareas", (req, res) => {
    res.json(tareas);
});

// 2. POST /api/tareas -> crear una
app.post("/api/tareas", (req, res) => {
    const error = validarTarea(req.body);
    if (error) {
        return res.status(400).json({ error });
    }
    const nueva = {
        id: siguienteId++,
        titulo: req.body.titulo.trim(),
        completada: false
    };
    tareas.push(nueva);
    guardar();
    res.status(201).json(nueva);
});

// 3. GET /api/tareas/:id -> leer una
app.get("/api/tareas/:id", (req, res) => {
    const id = Number(req.params.id);
    if (Number.isNaN(id)) {
        return res.status(400).json({ error: "El id debe ser un numero" });
    }
    const tarea = tareas.find(t => t.id === id);
    if (!tarea) {
        return res.status(404).json({ error: "Tarea no encontrada" });
    }
    res.json(tarea);
});

```

```

// 4. PUT /api/tareas/:id -> actualizar una
app.put("/api/tareas/:id", (req, res) => {
  const id = Number(req.params.id);
  if (Number.isNaN(id)) {
    return res.status(400).json({ error: "El id debe ser un numero" });
  }
  const indice = buscarIndice(id);
  if (indice === -1) {
    return res.status(404).json({ error: "Tarea no encontrada" });
  }
  const error = validarTarea(req.body);
  if (error) {
    return res.status(400).json({ error });
  }
  if (typeof req.body.completada !== "undefined" && typeof req.body.completada !==
"boolean") {
    return res.status(400).json({ error: "completada debe ser true o false" });
  }
  tareas[indice].titulo = req.body.titulo.trim();
  if (typeof req.body.completada === "boolean") {
    tareas[indice].completada = req.body.completada;
  }
  guardar();
  res.json(tareas[indice]);
});

// 5. DELETE /api/tareas/:id -> borrar una
app.delete("/api/tareas/:id", (req, res) => {
  const id = Number(req.params.id);
  if (Number.isNaN(id)) {
    return res.status(400).json({ error: "El id debe ser un numero" });
  }
  const indice = buscarIndice(id);
  if (indice === -1) {
    return res.status(404).json({ error: "Tarea no encontrada" });
  }
  const borrada = tareas.splice(indice, 1)[0];
  guardar();
  res.json({ mensaje: "Tarea eliminada", tarea: borrada });
});

app.listen(PORT, () => {
  console.log("API de tareas en http://localhost:" + PORT);
});

```

Vamos por bloques, sin prisa. El bloque de requires trae express, fs y path: las tres herramientas del servidor. El middleware tiene dos líneas clave: `express.json()` para leer cuerpos JSON (sin esto, `req.body` no existe) y `express.static("public")` para servir el frontend desde la misma app (adios, CORS). El bloque de carga lee `tareas.json` al arrancar con `try/catch`: si el archivo no existe (primera ejecución), el `catch` deja el arreglo vacío y el servidor arranca igual. El cálculo de `siguienteld` usa `Math.max` sobre los ids existentes más 1: así los ids nunca se repiten entre reinicios. La función `guardar()` escribe el arreglo completo al disco con formato bonito.

La función `validarTarea` devuelve un mensaje de error o `null`: revisa que el título exista, sea texto y no este vacío. `buscarIndice` usa `findIndex` para localizar la posición de una tarea por su id (devuelve -1 si no existe, que convertimos en 404).

Cada endpoint sigue el mismo ritual: 1) convertir y validar el id si lo hay, 2) validar los datos si los hay, 3) operar sobre el arreglo, 4) guardar en disco, 5) responder con el código correcto. GET colección responde 200 con el arreglo (`res.json` usa 200 por defecto). POST valida, crea con id único, guarda y responde 201 con la tarea creada. GET por id distingue 400 (id mal formado) de 404 (id bien formado pero inexistente). PUT valida todo, actualiza título y (opcionalmente) completada, guarda y responde 200. DELETE busca, elimina con `splice`, guarda y responde 200 con un mensaje y la tarea borrada.

Fíjate en los "return" antes de cada `res.status` de error: sin ellos, el código seguiría y trataría de responder dos veces. Y fíjate en que `guardar()` se llama en los tres endpoints que modifican datos (POST, PUT, DELETE), nunca en los GET: leer no cambia nada, así que no hay nada que persistir.

Para probarlo sin frontend, usa tu navegador para los GET (abre `http://localhost:3000/api/tareas`) y una herramienta de peticiones o `curl` para los demás. Prueba también los errores a propósito: POST sin título (debe dar 400), GET `/api/tareas/999` (debe dar 404), GET `/api/tareas/hola` (debe dar 400). Probar los errores es tan importante como probar el éxito: un endpoint que maneja bien sus errores es un endpoint profesional.

Ejemplo 2: cliente fetch que lista, crea y borra tareas

Este ejemplo es un script de Node.js (o código que puedes pegar en la consola del navegador) que consume tu API usando `fetch`. Demuestra el ciclo completo: leer la lista, crear una tarea, volver a leer, y borrar. Guardalo como `cliente.js` y correlo con "node cliente.js" mientras tu servidor esta encendido. (Node.js 18 o superior trae `fetch` incluido; si tu versión es más vieja, este ejemplo va en el navegador.)

```
const API = "http://localhost:3000/api/tareas";
```

```

async function listar(mensaje) {
  const respuesta = await fetch(API);
  if (!respuesta.ok) throw new Error("Error al listar: " + respuesta.status);
  const tareas = await respuesta.json();
  console.log("\n" + mensaje);
  if (tareas.length === 0) {
    console.log("(no hay tareas)");
  }
  tareas.forEach(t => {
    const marca = t.completada ? "[x]" : "[ ]";
    console.log(marca + " " + t.id + ": " + t.titulo);
  });
  return tareas;
}

async function crear(titulo) {
  const respuesta = await fetch(API, {
    method: "POST",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify({ titulo })
  });
  if (!respuesta.ok) {
    const detalle = await respuesta.json();
    throw new Error("Error al crear: " + detalle.error);
  }
  const nueva = await respuesta.json();
  console.log("Creada tarea " + nueva.id + ": " + nueva.titulo);
  return nueva;
}

async function borrar(id) {
  const respuesta = await fetch(API + "/" + id, { method: "DELETE" });
  if (!respuesta.ok) throw new Error("Error al borrar: " + respuesta.status);
  console.log("Borrada la tarea " + id);
}

async function principal() {
  try {
    await listar("Tareas al inicio:");
    const t1 = await crear("Estudiar Node.js");
    await crear("Hacer la API de tareas");
    await listar("Despues de crear dos:");
    await borrar(t1.id);
    await listar("Despues de borrar la primera:");
  }
}

```

```

    } catch (error) {
      console.log("Fallo algo:", error.message);
    }
  }
}

principal();

```

Observa los patrones que se repiten en las tres funciones: fetch con su configuración, revisión de respuesta.ok (porque fetch no rechaza con 404 o 400), conversión con await respuesta.json(), y errores claros. En crear(), fíjate como leemos el cuerpo del error (detalle.error) para mostrar el mensaje de validación que manda el servidor: así el cliente aprovecha los mensajes útiles del backend. En borrar(), no hay cuerpo que enviar: la URL con el id lo dice todo. La función principal encadena todo con await y un solo try/catch: si cualquier paso falla, el catch lo reporta. Correlo dos veces seguidas y observa algo interesante: la segunda vez, las tareas de la primera ejecución siguen ahí, porque la persistencia funciona. Tus datos sobreviven al reinicio del cliente y del servidor.

Ejemplo 3: frontend HTML mínimo conectado a la API

Este es el momento full stack: una página web servida por tu propio servidor que consume tu propia API. Crea la carpeta "public" junto a servidor.js, y dentro guarda estos dos archivos. Luego abre <http://localhost:3000> en el navegador.

El HTML (public/index.html):

```

<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  <title>Mis tareas</title>
</head>
<body>
  <h1>Mis tareas</h1>
  <p id="estado">Cargando tareas...</p>
  <form id="formulario">
    <input id="titulo" type="text" placeholder="Nueva tarea..." required>
    <button type="submit">Agregar</button>
  </form>
  <ul id="lista"></ul>
  <script src="app.js"></script>
</body>
</html>

```

El JavaScript del frontend (public/app.js):

```
const API = "/api/tareas"; // mismo origen: sin host, sin CORS
const lista = document.getElementById("lista");
const estado = document.getElementById("estado");
const formulario = document.getElementById("formulario");
const inputTitulo = document.getElementById("titulo");

async function cargarTareas() {
  try {
    estado.textContent = "Cargando tareas...";
    const respuesta = await fetch(API);
    if (!respuesta.ok) throw new Error("codigo " + respuesta.status);
    const tareas = await respuesta.json();
    pintar(tareas);
    estado.textContent = tareas.length === 0
      ? "No hay tareas. Agrega la primera."
      : "Tienes " + tareas.length + " tarea(s).";
  } catch (error) {
    estado.textContent = "No se pudo conectar con el servidor.";
    console.log(error.message);
  }
}

function pintar(tareas) {
  lista.innerHTML = "";
  tareas.forEach(t => {
    const li = document.createElement("li");
    li.textContent = t.titulo + (t.completada ? " (hecha)" : "");

    const btnToggle = document.createElement("button");
    btnToggle.textContent = t.completada ? "Desmarcar" : "Completar";
    btnToggle.addEventListener("click", () => alternar(t));

    const btnBorrar = document.createElement("button");
    btnBorrar.textContent = "Borrar";
    btnBorrar.addEventListener("click", () => borrar(t.id));

    li.appendChild(document.createTextNode(" "));
    li.appendChild(btnToggle);
    li.appendChild(document.createTextNode(" "));
    li.appendChild(btnBorrar);
    lista.appendChild(li);
  });
}
```

```

formulario.addEventListener("submit", async (e) => {
  e.preventDefault();
  const titulo = inputTitulo.value.trim();
  if (titulo === "") return;
  try {
    const respuesta = await fetch(API, {
      method: "POST",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify({ titulo })
    });
    if (!respuesta.ok) {
      const detalle = await respuesta.json();
      throw new Error(detalle.error || "no se pudo crear");
    }
    inputTitulo.value = "";
    await cargarTareas(); // recargar y repintar
  } catch (error) {
    estado.textContent = "Error: " + error.message;
  }
});

async function alternar(t) {
  try {
    const respuesta = await fetch(API + "/" + t.id, {
      method: "PUT",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify({ titulo: t.titulo, completada: !t.completada })
    });
    if (!respuesta.ok) throw new Error("no se pudo actualizar");
    await cargarTareas();
  } catch (error) {
    estado.textContent = "Error: " + error.message;
  }
}

async function borrar(id) {
  try {
    const respuesta = await fetch(API + "/" + id, { method: "DELETE" });
    if (!respuesta.ok) throw new Error("no se pudo borrar");
    await cargarTareas();
  } catch (error) {
    estado.textContent = "Error: " + error.message;
  }
}

```

```
cargarTareas();
```

Tres decisiones de diseño que debes notar. Una: la constante API es `"/api/tareas"`, una ruta RELATIVA, sin host ni puerto. Como la página la sirve el mismo servidor, el navegador completa el origen solo: funciona en `localhost:3000` y funcionara igual cuando despliegues, sin cambiar código. Dos: después de cada operación de escritura (crear en el submit, alternar, borrar) se llama a `cargarTareas()`: recargar y repintar, la estrategia simple y correcta. Tres: el párrafo "estado" siempre le dice al usuario que esta pasando: cargando, cuantas hay, errores de conexión. Nunca dejes al usuario frente a una pantalla muda.

Pruébalo de principio a fin: abre la página, agrega tres tareas, completa una, borra otra, recarga la página (las tareas siguen ahí: persistencia), reinicia el servidor (siguen ahí), abre `tareas.json` y mira tus datos en disco. Acabas de hacer full stack con tus propias manos: frontend y backend, tuyos, hablando entre si.

Práctica

Ejercicio 1: filtro por query string (`?completada=true`)

Enunciado: modifica el endpoint `GET /api/tareas` para que acepte un query string opcional "completada". Si el cliente pide `GET /api/tareas?completada=true`, debe devolver solo las completadas; si pide `?completada=false`, solo las pendientes; si no manda el query, devuelve todas como antes.

Solución paso a paso. Paso 1: recuerda que Express entrega los query strings en `req.query`, siempre como strings. `req.query.completada` será `"true"`, `"false"` o `undefined`. Paso 2: si es `undefined`, responde con el arreglo completo (comportamiento actual, no rompemos nada). Paso 3: si es `"true"` o `"false"`, filtra el arreglo comparando `t.completada` con el booleano correspondiente. Paso 4: si es cualquier otro valor (`"si"`, `"1"`, `"hola"`), responde 400 porque el filtro está mal formado. Así queda:


```

app.get("/api/tareas", (req, res) => {
  const filtro = req.query.completada;
  if (typeof filtro === "undefined") {
    return res.json(tareas);
  }
  if (filtro !== "true" && filtro !== "false") {
    return res.status(400).json({ error: "completada debe ser true o false" });
  }
  const soloCompletadas = filtro === "true";
  const filtradas = tareas.filter(t => t.completada === soloCompletadas);
  res.json(filtradas);
});

```

Pruébalo: crea tareas, completa algunas, y abre en el navegador `/api/tareas?completada=true` y `?completada=false`. Observa que el endpoint viejo (sin query) sigue funcionando: eso se llama compatibilidad hacia atrás y es una virtud profesional.

Ejercicio 2: validación más estricta del título

Enunciado: endurece `validarTarea` para que el título tenga entre 3 y 100 caracteres después de quitar espacios al inicio y final. Si no cumple, responde 400 con un mensaje específico para cada caso.

Solución paso a paso. Paso 1: calcula el título limpio una sola vez con `trim()` y guardalo en una variable, para no repetir la operación. Paso 2: revisa la longitud mínima primero (más específico), luego la máxima. Paso 3: devuelve mensajes distintos para que el cliente sepa exactamente que corregir. Código:

```

function validarTarea(datos) {
  if (!datos || typeof datos.titulo !== "string") {
    return "El titulo es obligatorio y debe ser texto";
  }
  const limpio = datos.titulo.trim();
  if (limpio.length < 3) {
    return "El titulo debe tener al menos 3 caracteres";
  }
  if (limpio.length > 100) {
    return "El titulo no puede pasar de 100 caracteres";
  }
  return null;
}

```

Y no olvides usar el título LIMPIO al crear y actualizar (`req.body.título.trim()`), para no guardar espacios basura en los bordes. Prueba con POST de `{"título": "ab"}` (debe dar 400), `{"título": " "}` (400) y un título de 101 caracteres (400). La validación estricta es lo que separa un prototipo de un servicio serio.

Ejercicio 3: paginación con ?página y ?limite

Enunciado: cuando haya cientos de tareas, devolverlas todas de golpe es pesado. Agrega paginación al GET `/api/tareas`: acepta `?página=N` y `?limite=M` (ambos opcionales, por defecto página 1 y limite 10) y devuelve solo ese "pedazo" del arreglo, junto con metadatos: total de tareas, página actual y total de páginas.

Solución paso a paso. Paso 1: lee los query strings y conviértelos a número con `Number()`; si no vienen, usa los valores por defecto. Paso 2: valida que sean enteros positivos; si no, 400. Paso 3: calcula el inicio con $(\text{página} - 1) * \text{limite}$ y corta el arreglo con `slice(inicio, inicio + limite)`. Paso 4: responde un objeto con los datos y los metadatos (ya no el arreglo pelado: documenta el cambio).

```
app.get("/api/tareas", (req, res) => {
  const pagina = req.query.pagina ? Number(req.query.pagina) : 1;
  const limite = req.query.limite ? Number(req.query.limite) : 10;
  if (!Number.isInteger(pagina) || pagina < 1 || !Number.isInteger(limite) ||
  limite < 1) {
    return res.status(400).json({ error: "pagina y limite deben ser enteros positivos" });
  }
  const inicio = (pagina - 1) * limite;
  const datos = tareas.slice(inicio, inicio + limite);
  res.json({
    datos,
    total: tareas.length,
    pagina,
    limite,
    totalPaginas: Math.ceil(tareas.length / limite)
  });
});
```

Nota: este ejercicio cambia la forma de la respuesta del GET (de arreglo a objeto), así que tu frontend tendría que adaptarse (usar `respuesta.datos` en lugar de la respuesta directa). En el mundo real, un cambió así se haría en una nueva versión de la API (`/api/v2/tareas`): por eso existen las versiones. Pruébalo con `?página=2&limite=5` después de crear 12 tareas.

Ejercicio 4: endpoint de estadísticas

Enunciado: agrega GET `/api/tareas/stats` que devuelva un resumen: total de tareas, cuantas completadas, cuantas pendientes y el porcentaje completado (redondeado). No debe confundirse con GET `/api/tareas/:id`.

Solución paso a paso. Paso 1: define la ruta ANTES que `/api/tareas/:id` en tu código, porque Express revisa las rutas en orden y "stats" podría interpretarse como un `:id`. (Con nuestra validación de id numerico daría 400 en lugar de las stats: por eso el orden importa.) Paso 2: cuenta con filter o reduce. Paso 3: calcula el porcentaje cuidando la división entre cero.

```
// IMPORTANTE: esta ruta va ANTES de app.get("/api/tareas/:id")
app.get("/api/tareas/stats", (req, res) => {
  const total = tareas.length;
  const completadas = tareas.filter(t => t.completada).length;
  const pendientes = total - completadas;
  const porcentaje = total === 0 ? 0 : Math.round((completadas / total) * 100);
  res.json({ total, completadas, pendientes, porcentaje });
});
```

La lección del orden de rutas es sutil y valiosa: en Express, las rutas específicas van antes que las parametrizadas. `/api/tareas/stats` antes que `/api/tareas/:id`. Si lo pones al revés, "stats" cae en `:id` y tu endpoint de estadísticas jamás se ejecuta. Pruébalo en ambos ordenes para ver la diferencia con tus propios ojos.

Ejercicio 5: borrado múltiple de completadas

Enunciado: agrega DELETE `/api/tareas?completada=true` que borre TODAS las tareas completadas de una vez y responda cuantas borró. Si no se manda el query, debe responder 400 (no queremos un borrado accidental de todo sin filtro).

Solución paso a paso. Paso 1: exige el query `completada=true`; cualquier otro caso es 400 con mensaje claro ("para borrar todo usa el query `?completada=true`"). Paso 2: cuenta cuantas hay completadas antes de borrar, para informar. Paso 3: reasigna el arreglo quedandote solo con las pendientes (filter) y guarda en disco. Ojo: "tareas" debe estar declarado con `let`, no `const`, para poder reasignarlo.

```

app.delete("/api/tareas", (req, res) => {
  if (req.query.completada !== "true") {
    return res.status(400).json({ error: "Indica ?completada=true para borrar las completadas" });
  }
  const antes = tareas.length;
  tareas = tareas.filter(t => !t.completada);
  const borradas = antes - tareas.length;
  guardar();
  res.json({ mensaje: "Tareas completadas eliminadas", borradas });
});

```

Este endpoint es destructivo, así que el requisito del query explícito es una protección deliberada: obligas al cliente a decir "se lo que hago". Es un patrón real: las operaciones peligrosas piden confirmación explícita. Prueba: completa dos tareas, llama al DELETE con el query, verifica el conteo en la respuesta y la lista final.

Ejercicio 6: respaldo automatico del archivo de datos

Enunciado: cada vez que guardar() escriba tareas.json, crea antes una copia de seguridad (tareas.respaldo.json) con el contenido anterior. Así, si algo se corrompe, puedes recuperar.

Solución paso a paso. Paso 1: antes de escribir el archivo nuevo, verifica si existe el actual con fs.existsSync. Paso 2: si existe, copialo al respaldo con fs.copyFileSync. Paso 3: escribe el nuevo contenido como antes. Todo envuelto para no tronar si algo falla.

```

function guardar() {
  try {
    if (fs.existsSync(ARCHIVO)) {
      fs.copyFileSync(ARCHIVO, path.join(__dirname, "tareas.respaldo.json"));
    }
    fs.writeFileSync(ARCHIVO, JSON.stringify(tareas, null, 2));
  } catch (e) {
    console.log("Error al guardar:", e.message);
  }
}

```

Prueba: crea una tarea, revisa que exista el respaldo; crea otra, abre el respaldo y confirma que tiene el estado anterior (una tarea menos). Este ejercicio te ensena un habito profesional: los datos valiosos siempre tienen respaldo, y el respaldo se hace ANTES del cambio, no después. En el mundo real esto evoluciona a respaldos programados y bases de datos con replicas; el principio es el mismo que acabas de programar.

Resumen de la unidad

Llegaste al final de la Unidad 10 y del curso completo. Respiremos y pongamos en orden todo lo que esta unidad te dio, porque es mucho:

Una API es un contrato entre programas; REST es el estilo de diseño más usado para APIs sobre HTTP, guiado por seis principios (cliente-servidor, sin estado, cacheable, interfaz uniforme, sistema en capas y código bajo demanda opcional). La analogía del mesero con menú fijo te ayuda a recordarlo: el cliente pide del menú (rutas y verbos documentados), el mesero (la API) va a la cocina (los datos) y sirve el plato (la respuesta), sin que el cliente entre jamás a la cocina.

Todo se modela como recursos con URLs de sustantivos (`/api/tareas`, `/api/tareas/:id`) y las acciones las dicen los verbos HTTP en correspondencia con CRUD: GET lee, POST crea, PUT actualiza, DELETE borra. Cada combinación de verbo más ruta es un endpoint, y una API bien diseñada es predecible: quien conoce tus endpoints de tareas adivina los de usuarios.

JSON es el idioma entre apps: texto plano con seis tipos de datos (string, number, boolean, null, object, array), claves siempre entre comillas dobles, sin comentarios ni funciones. `JSON.stringify` convierte objetos a texto para enviar; `JSON.parse` convierte texto a objetos para trabajar. En Express, `res.json()` serializa por ti y `express.json()` te entrega el cuerpo ya convertido en `req.body`.

Los códigos de estado cuentan la historia de cada petición: 200 éxito, 201 creado, 204 sin contenido, 400 petición mal formada, 404 no encontrado. La validación es el portero: revisa título obligatorio, tipo texto, no vacío, y responde 400 con mensajes claros, siempre con `return` después de responder. Los ids los asigna el servidor (contador con ajuste al arrancar, o `Date.now()`), nunca el cliente.

La persistencia en archivo JSON con `fs` te enseñó el concepto que las bases de datos llevan al nivel profesional: cargar al arrancar con `try/catch`, guardar en cada cambio, y calcular el siguiente id desde los datos existentes. CORS te explicó por qué el navegador bloquea `fetch` entre orígenes distintos y sus dos soluciones: servir el frontend desde el mismo servidor (`express.static`) o el middleware `cors`.

Del lado del cliente, `fetch` con `async/await`, revisión obligatoria de `respuesta.ok` (`fetch` no rechaza con 404), encabezado `Content-Type: application/json` al enviar, y manejo de errores visible para el usuario. El frontend mínimo sigue el ciclo estado -> mostrar -> escuchar -> actualizar, con recargar y repintar después de cada escritura. Y el despliegue: PORT desde `process.env`, secretos en variables de entorno, dependencias en `package.json`, checklist de siete puntos.

Tu proyecto final une todo: servidor Express con 5 endpoints, validación, ids únicos, persistencia en JSON, frontend servido por el mismo servidor que consume la API con fetch. Es tu primer backend funcional, y es arquitectura real.

Y ahora, la graduación. Felicidades: terminaste el "Curso Completo de Node.js" de Lucas EGA Academy. Empezaste encendiendo un servidor que decía "Hola mundo" y terminas con una API REST completa que crea, lee, actualiza, borra, valida, persiste y se habla con un frontend. Dominas los módulos, el sistema de archivos, los eventos, lo asincrono, Express y las APIs REST. Eso no es poca cosa: es el conjunto de habilidades con el que miles de personas consiguen su primer trabajo como desarrolladoras backend.

El camino no termina aquí; cambia de nivel. Tu siguiente parada es el curso de bases de datos con SQL en lucasegaacademy.com: tablas, consultas y relaciones, el paso profesional después de tu persistencia en archivo. Después, autenticación: usuarios reales con sesiones reales. Y entre un curso y otro, proyectos: un blog con API, una tienda con carrito, el clon simplificado de tu app favorita. Cada proyecto te va a doler un poco y te va a enseñar mucho. Esa es la fórmula y no hay atajos.

Guarda este proyecto. Subelo a un repositorio cuando aprendas Git. Dentro de un año vas a volver a ver tu API de tareas y te va a dar ternura lo simple que era, y esa ternura será la medida exacta de cuanto creciste. Todos los programadores tenemos ese primer backend que nos da orgullo y risa a la vez. El tuyo ya existe, lo construiste tu, con tus propias manos: full stack, de principio a fin.

Nos vemos en el curso de SQL, en lucasegaacademy.com. El backend ya no es un misterio para ti. Ahora ve por las bases de datos. Felicidades, graduado.

EXAMEN FINAL

Instrucciones

Bienvenido al examen final del libro "Curso Completo de Node.js" de Lucas EGA Academy. Este examen evalúa todo lo que aprendiste a lo largo de las 10 unidades del curso, desde los fundamentos de Node.js y su instalación hasta la creación de APIs REST y el despliegue de aplicaciones en la nube.

El examen consta de 50 preguntas repartidas de forma equilibrada entre las 10 unidades del libro, con aproximadamente 5 preguntas por unidad. Encontrarás tres tipos de preguntas. Primero, 20 preguntas de opción múltiple con cuatro posibles respuestas marcadas como a), b), c) y d), de las cuales solo una es correcta. Segundo, 15 preguntas de verdadero o falso, marcadas con (V/F), donde debes decidir si la afirmación es verdadera o falsa. Tercero, 15 preguntas abiertas o de código, donde se te pide escribir un fragmento corto de código, predecir la salida de un programa, encontrar un error o completar líneas faltantes.

Cada pregunta vale 2 puntos, así que el puntaje máximo es de 100 puntos. La calificación se interpreta de la siguiente manera. Si obtienes entre 90 y 100 puntos tu desempeño es excelente: dominas los temas y estás listo para construir proyectos reales con Node.js. Si obtienes entre 70 y 89 puntos tu desempeño es bien: tienes una base sólida, pero conviene repasar las unidades donde fallaste más. Si obtienes menos de 70 puntos necesitas repasar las unidades del libro con calma antes de seguir adelante, poniendo especial atención a los temas de las preguntas que respondiste mal.

Se recomienda presentar el examen en un lugar tranquilo y con tiempo suficiente. Calcula alrededor de 90 minutos para responder con calma las 50 preguntas. Puedes tener a la mano tu editor de código y la terminal para probar los fragmentos de las preguntas de programación, porque experimentar es parte del aprendizaje. Sin embargo, intenta responder primero por tu cuenta y usa la computadora solo para verificar, no para buscar las respuestas en internet. Sé honesto contigo mismo: el objetivo del examen no es la calificación, sino descubrir qué temas ya dominas y cuáles necesitas reforzar.

Lee cada pregunta con atención antes de responder. En las preguntas de opción múltiple, descarta primero las opciones que claramente son incorrectas. En las de verdadero o falso, fíjate en palabras absolutas como "siempre", "nunca" o "solo", porque muchas veces esconden la trampa. En las preguntas de código, analiza línea por línea y recuerda cómo funciona el modelo asíncrono de Node.js. Cuando termines, compara tus respuestas con la sección de

"Respuestas y soluciones" que está al final del documento y lee cada explicación, incluso las de las preguntas que acertaste, porque ahí hay detalles que refuerzan lo aprendido.

Mucho éxito. Si llegaste hasta aquí es porque ya recorriste todo el curso, y eso ya es un logro.

Para calificarte, cuenta cuántas preguntas respondiste correctamente y multiplica ese número por 2. Ese es tu puntaje final sobre 100. Por ejemplo, si acertaste 42 preguntas, tu puntaje es 84. Anota además en qué unidades fallaste más, porque ese dato vale más que el número: te dice exactamente dónde enfocar tu repaso. Las preguntas están repartidas así: las preguntas 1 a 5 corresponden a la unidad 1 (fundamentos e instalación), las preguntas 6 a 10 a la unidad 2 (módulos, process y globales), las preguntas 11 a 15 a la unidad 3 (fs y path), las preguntas 16 a 20 a la unidad 4 (módulo os y servidor HTTP), las preguntas 21 a 25 a la unidad 5 (npm, package.json y nodemon), las preguntas 26 a 30 a la unidad 6 (EventEmitter, streams y buffers), las preguntas 31 a 35 a la unidad 7 (callbacks, promesas, async/await y event loop), las preguntas 36 a 40 a la unidad 8 (Express y rutas GET/POST), las preguntas 41 a 45 a la unidad 9 (parámetros, query strings, middleware y archivos estáticos), y las preguntas 46 a 50 a la unidad 10 (APIs REST, CRUD, fetch y despliegue).

Si tu puntaje quedó entre 90 y 100, tu desempeño es excelente. Dominas los fundamentos, el trabajo con archivos, el modelo asíncrono y la creación de servidores y APIs. Estás listo para construir proyectos reales, como una API completa con base de datos o una aplicación de tiempo real. Como siguiente paso, te conviene practicar integrando todo en un proyecto grande y aprender sobre autenticación y despliegue avanzado.

Si tu puntaje quedó entre 70 y 89, tu desempeño es bien. Tienes una base sólida y entiendes la mayoría de los conceptos, pero hay unidades donde todavía dudas. Identifica esas unidades con la lista de arriba y vuelve a leerlas en el libro, prestando atención a los ejemplos de código. Repite los ejercicios de esas unidades y vuelve a presentar solo esas preguntas una semana después. Con ese repaso enfocado llegarás al nivel excelente sin mucho esfuerzo extra.

Si tu puntaje quedó por debajo de 70, necesitas repasar las unidades con calma antes de seguir adelante. No es un fracaso: Node.js tiene conceptos nuevos para quien viene del navegador, como el modelo asíncrono, los módulos y el event loop, y es normal que tomen tiempo. Empieza por las unidades 1 a 5 para afianzar la base, luego ataca las unidades 6 y 7 que son el corazón del curso, y termina con las unidades 8 a 10 que son la parte más práctica y divertida. Vuelve a presentar el examen completo cuando termines el repaso.

Hay trampas frecuentes que conviene tener presentes. En las preguntas de verdadero o falso, desconfía de las palabras absolutas como "siempre", "nunca", "solo" o "únicamente", porque muchas afirmaciones falsas se construyen con ellas. En las preguntas de código, no confundas las versiones síncronas y asíncronas de fs: las que terminan en Sync devuelven el

valor directamente y las demás usan callbacks o promesas. Recuerda también que en http puro se responde con `res.end` y no con `res.send`, que es exclusivo de Express. Y en las preguntas del event loop, aplica la regla de oro: primero todo el código síncrono, después las microtareas de las promesas y al final las macrotareas como `setTimeout`.

Un último consejo: las preguntas abiertas y de código no se califican con todo o nada. Si tu solución funciona aunque esté escrita de forma distinta a la propuesta en las soluciones, date el punto completo. Lo importante es que el programa haga lo pedido y que entiendas por qué funciona. La programación se trata de resolver problemas, no de memorizar respuestas exactas.

Antes de empezar, usa esta lista de verificación para saber qué debes dominar de cada unidad. Unidad 1: debes saber qué es Node.js, qué motor usa, cómo instalarlo y cómo ejecutar un archivo con el comando `node`. Unidad 2: debes dominar `require` y `module.exports`, el objeto `process` con `process.argv`, y las variables globales como `__dirname`. Unidad 3: debes manejar `fs.readFile` y `fs.writeFile` con callbacks, sus versiones Sync con `try/catch`, y `path.join` para rutas portables. Unidad 4: debes conocer los métodos básicos del módulo `os` y saber levantar un servidor con `http.createServer`, responder según `req.url` y usar `server.listen`. Unidad 5: debes entender qué es el `package.json`, para qué sirven `npm install` y `npm init`, por qué `node_modules` va en el `.gitignore` y qué hace `nodemon`. Unidad 6: debes saber registrar eventos con `on` y emitirlos con `emit`, entender qué es un Buffer y usar streams con pipe para archivos grandes. Unidad 7: debes dominar el orden del event loop, convertir callbacks a promesas y usar `async/await` con `try/catch` sin dudar. Unidad 8: debes crear rutas GET y POST con Express, usar `res.send` y `res.json`, y levantar el servidor con `app.listen`. Unidad 9: debes obtener parámetros con `req.params`, query strings con `req.query`, escribir middlewares con `next` y servir archivos estáticos con `express.static`. Unidad 10: debes aplicar los métodos HTTP según REST, usar los códigos 200 y 201 correctamente, consumir APIs con `fetch` y conocer los pasos del despliegue en la nube. Si puedes explicar cada punto de esta lista con tus propias palabras y escribir un ejemplo de código, el examen será un trámite.

Preguntas

A continuación encontrarás las 50 preguntas del examen, mezcladas por tipo pero ordenadas por unidad para que puedas ubicar fácilmente a qué tema pertenece cada una. Las primeras 20 preguntas de cada bloque de unidad son de opción múltiple, seguidas por las de verdadero o falso y las abiertas o de código, aunque el orden exacto varía para mantener el examen dinámico. Lee cada enunciado completo antes de mirar las opciones, y en las preguntas de código tómate el tiempo de simular mentalmente la ejecución línea por línea. Si una pregunta

te parece muy difícil, márcala y sigue adelante: a veces una pregunta posterior te refresca la memoria sobre el tema. En las preguntas abiertas no busques la redacción perfecta; busca que el código funcione y que la explicación demuestre que entiendes el concepto. Y en las de predecir salida, escribe primero tu predicción en un papel antes de ejecutar el código para comprobar, porque adivinar después de ver el resultado no te enseña nada. Si te atorras en una pregunta de código, respira y divídela en partes pequeñas: identifica qué hace cada línea por separado y luego une las piezas. La mayoría de los errores en exámenes no son de conocimiento, sino de prisa: leer mal el enunciado o confundir dos métodos parecidos. La calma también se entrena.

PREGUNTA 1: ¿Qué es Node.js? a) Un framework de JavaScript para crear interfaces de usuario en el navegador b) Un entorno de ejecución que permite correr JavaScript fuera del navegador, del lado del servidor c) Un sistema manejador de bases de datos relacionales d) Un lenguaje de programación nuevo creado desde cero por Google

PREGUNTA 2: ¿Qué motor de JavaScript utiliza Node.js internamente para ejecutar el código? a) SpiderMonkey, el motor de Firefox b) V8, el motor creado por Google para Chrome c) JavaScriptCore, el motor de Safari d) Chakra, el motor clásico de Internet Explorer

PREGUNTA 3: (V/F) Node.js solo puede ejecutar un programa si antes se compila a código binario nativo con un compilador como gcc.

PREGUNTA 4: (V/F) Para verificar que Node.js quedó bien instalado en la computadora basta con abrir la terminal y ejecutar el comando "node -v", el cual debe mostrar el número de versión.

PREGUNTA 5: Escribe en orden los comandos de terminal necesarios para realizar estas cuatro tareas: 1) crear una carpeta llamada "mi-proyecto", 2) entrar a esa carpeta, 3) crear dentro un archivo app.js que imprima "Hola Node" usando console.log, 4) ejecutar ese archivo con Node.js. Puedes usar comandos de Linux, macOS o Windows según tu sistema, pero indica claramente cada paso.

PREGUNTA 6: ¿Cuál es la forma correcta de importar el módulo "fs" usando el sistema de módulos CommonJS? a) import fs from "fs" b) const fs = require("fs") c) include("fs") d) using fs = "fs"

PREGUNTA 7: ¿Qué objeto global de Node.js permite acceder a los argumentos que se pasan en la línea de comandos al ejecutar un programa? a) window.args b) process.argv c) global.args d) console.argv

PREGUNTA 8: (V/F) En Node.js, la variable `__dirname` contiene la ruta absoluta de la carpeta donde se encuentra el archivo que se está ejecutando.

PREGUNTA 9: Encuentra el error en el siguiente fragmento de código, explica por qué falla y escribe la versión corregida.

```
const os = require("os");
console.log("Plataforma: " + os.plataforma());
```

PREGUNTA 10: Completa el código para que funcione el sistema de exportación e importación entre dos archivos. Escribe la línea que falta en cada archivo.

```
// archivo saludo.js
function saludar(nombre) {
  return "Hola " + nombre;
}
// FALTA LÍNEA 1: exportar la función saludar

// archivo app.js
// FALTA LÍNEA 2: importar la función saludar desde "../saludo"
console.log(saludar("Esteban"));
```

PREGUNTA 11: ¿Qué método del módulo `fs` lee el contenido de un archivo de forma asíncrona, sin bloquear el programa mientras se lee? a) `fs.readFileSync` b) `fs.readFile` c) `fs.openFile` d) `fs.getFile`

PREGUNTA 12: ¿Qué hace la llamada `path.join("carpeta", "sub", "archivo.txt")`? a) Crea físicamente esas carpetas en el disco duro b) Une los segmentos en una ruta válida usando el separador correcto según el sistema operativo c) Verifica si el archivo existe en esa ruta d) Copia el archivo a una carpeta temporal

PREGUNTA 13: (V/F) El método `fs.writeFile` crea el archivo si no existe, pero si el archivo ya existe lanza un error y no modifica su contenido.

PREGUNTA 14: Encuentra el error en el siguiente fragmento de código, explica por qué falla y escribe la versión corregida.

```
const fs = require("fs");
const datos = fs.readFile("datos.txt", "utf8");
console.log(datos);
```

PREGUNTA 15: Escribe un programa completo en Node.js que lea el archivo "notas.txt" usando `fs.readFileSync` dentro de un bloque `try/catch`. Si la lectura tiene éxito, debe imprimir el contenido del archivo en la consola. Si el archivo no existe o ocurre cualquier error, debe mostrar un mensaje amable como "No se pudo leer el archivo" junto con una breve descripción del problema.

PREGUNTA 16: ¿Qué método del módulo `os` devuelve la cantidad de memoria RAM libre disponible en el sistema? a) `os.freeMemory()` b) `os.freemem()` c) `os.memory()` d) `os.ram()`

PREGUNTA 17: Cuando se crea un servidor HTTP con el módulo `http` de Node.js puro (sin Express), ¿qué método del objeto de respuesta se usa para terminar la respuesta y enviarla al cliente? a) `res.send()` b) `res.end()` c) `res.write()` usado solo, sin nada más d) `res.close()`

PREGUNTA 18: (V/F) La instrucción `server.listen(3000)` hace que el servidor HTTP comience a escuchar peticiones en el puerto 3000 de la computadora.

PREGUNTA 19: Analiza el siguiente servidor y responde: si un navegador visita la dirección `http://localhost:3000/saludo`, ¿qué código de estado HTTP recibe y qué texto se muestra en pantalla? ¿Y qué recibe si visita `http://localhost:3000/` ?

```
const http = require("http");

const server = http.createServer((req, res) => {
  if (req.url === "/saludo") {
    res.writeHead(200, { "Content-Type": "text/plain" });
    res.end("Hola desde Node");
  } else {
    res.writeHead(404, { "Content-Type": "text/plain" });
    res.end("No encontrado");
  }
});

server.listen(3000);
```

PREGUNTA 20: Completa el siguiente servidor para que responda con el objeto `{ curso: "Node.js" }` en formato JSON. Escribe las líneas que faltan, incluyendo el encabezado `Content-Type` correcto.

```
const http = require("http");

const server = http.createServer((req, res) => {
  const datos = { curso: "Node.js" };
  // FALTA: configurar el encabezado como application/json
  // FALTA: enviar el objeto convertido a texto JSON y terminar la respuesta
});

server.listen(3000);
```

PREGUNTA 21: ¿Para qué sirve ejecutar el comando "npm init -y" dentro de la carpeta de un proyecto? a) Instala todas las dependencias del proyecto b) Crea un archivo package.json con valores por defecto sin hacer preguntas c) Actualiza Node.js a la última versión d) Borra la carpeta node_modules y la vuelve a crear

PREGUNTA 22: ¿Qué hace la herramienta nodemon cuando se usa para ejecutar una aplicación de Node.js? a) Convierte el código JavaScript a código máquina para que corra más rápido b) Reinicia automáticamente la aplicación cada vez que detecta cambios en los archivos del proyecto c) Crea una copia de seguridad del proyecto en la nube d) Optimiza el uso de memoria del servidor HTTP

PREGUNTA 23: (V/F) La carpeta node_modules debe subirse al repositorio de Git junto con el código del proyecto, porque contiene las dependencias sin las cuales la aplicación no funciona.

PREGUNTA 24: (V/F) Al ejecutar el comando "npm install" sin ningún argumento dentro de un proyecto, npm lee el archivo package.json e instala todas las dependencias listadas en él.

PREGUNTA 25: Escribe el contenido completo de un archivo package.json mínimo pero válido para un proyecto con estas características: el nombre del proyecto es "mi-api", la versión es "1.0.0", tiene un script llamado "start" que ejecuta el comando "node server.js", y declara a express como dependencia en la versión "^4.18.2". Incluye también una breve descripción del proyecto.

PREGUNTA 26: ¿Qué método de la clase EventEmitter se utiliza para registrar una función que se ejecutará cada vez que se emita un evento con cierto nombre? a) emitter.listen("evento", función) b) emitter.on("evento", función) c) emitter.add("evento", función) d) emitter.subscribe("evento", función)

PREGUNTA 27: ¿Qué es un Buffer en Node.js? a) Una memoria caché del navegador para guardar páginas web b) Una estructura de datos diseñada para manejar datos

binarios directamente, como el contenido de archivos o datos de red c) Un tipo de stream que solo funciona con texto plano d) Una función que retrasa la ejecución del código unos milisegundos

PREGUNTA 28: (V/F) Los streams permiten procesar datos por fragmentos (chunks) conforme van llegando, sin necesidad de cargar todo el contenido en memoria de una sola vez.

PREGUNTA 29: Predice con exactitud el orden de las líneas que se imprimirán en la consola al ejecutar el siguiente programa. Explica brevemente por qué sale en ese orden.

```
const { EventEmitter } = require("events");

const emisor = new EventEmitter();

emisor.on("listo", (mensaje) => {
  console.log("Recibido: " + mensaje);
});

emisor.emit("listo", "Hola mundo");
console.log("Fin");
```

PREGUNTA 30: Completa el siguiente programa para copiar el contenido de "entrada.txt" a "salida.txt" usando streams, de modo que funcione incluso con archivos muy grandes. Escribe las líneas que faltan.

```
const fs = require("fs");

const lectura = fs.createReadStream("entrada.txt");
// FALTA LÍNEA 1: crear el stream de escritura hacia "salida.txt"
// FALTA LÍNEA 2: conectar el stream de lectura con el de escritura
```

PREGUNTA 31: ¿En qué orden se imprimen los números al ejecutar el siguiente código?
a) 1, 2, 3, 4 b) 1, 4, 3, 2 c) 1, 4, 2, 3 d) 4, 1, 3, 2

```
console.log("1");
setTimeout(() => console.log("2"), 0);
Promise.resolve().then(() => console.log("3"));
console.log("4");
```

PREGUNTA 32: Dentro de una función declarada con `async`, ¿qué palabra clave se usa para pausar la ejecución hasta que una promesa se resuelva y obtener su valor? a) `wait` b) `await` c) `async` d) `then`

PREGUNTA 33: (V/F) Una función declarada con la palabra `async` siempre devuelve una promesa, incluso si dentro de ella se retorna un valor simple como un número o un texto.

PREGUNTA 34: (V/F) El callback que se pasa a `fs.readFile` se ejecuta de forma inmediata, antes de que el sistema operativo termine de leer el archivo del disco.

PREGUNTA 35: Reescribe el siguiente programa que usa callbacks con la sintaxis moderna de `async/await`. Usa el módulo `"fs/promises"` y maneja el posible error con `try/catch`.

```
const fs = require("fs");

fs.readFile("datos.txt", "utf8", (err, data) => {
  if (err) {
    console.error("Error al leer");
    return;
  }
  console.log(data);
});
```

PREGUNTA 36: ¿Cuál es la forma correcta de crear una ruta que responda a peticiones GET en `"/ruta"` usando Express? a) `app.get("/ruta", manejador)` b) `app.route("GET", "/ruta", manejador)` c) `express.get("/ruta", manejador)` d) `app.fetch("/ruta", manejador)`

PREGUNTA 37: ¿Qué hace la instrucción `res.json({ ok: true })` dentro de un manejador de Express? a) Guarda el objeto en un archivo JSON del servidor b) Envía una respuesta con el objeto convertido a JSON y el encabezado `Content-Type` correcto c) Imprime el objeto en la consola del servidor d) Valida que el objeto tenga la propiedad `ok`

PREGUNTA 38: (V/F) En Express, el objeto `req.body` contiene los datos enviados en el cuerpo de una petición POST, siempre y cuando se haya configurado el middleware que interpreta ese cuerpo, como `express.json()`.

PREGUNTA 39: (V/F) La instrucción `app.listen(3000)` en Express es equivalente a crear un servidor HTTP con el módulo `http` y ponerlo a escuchar peticiones en el puerto 3000.

PREGUNTA 40: Encuentra el error en el siguiente programa de Express, explica por qué el servidor no funciona cómo se espera y escribe la versión corregida.

```
const express = require("express");
const app = express();

app.get("/usuarios", (req, res) => {
  res.send("Lista de usuarios");
});
```

PREGUNTA 41: Si se define la ruta `"/usuarios/:id"` en Express y un cliente visita `"/usuarios/42"`, ¿cómo se obtiene el valor `"42"` dentro del manejador? a) `req.params.id` b) `req.query.id` c) `req.body.id` d) `req.id`

PREGUNTA 42: Si un cliente visita la URL `"/buscar?q=node&orden=asc"`, ¿cómo se obtiene el valor `"node"` dentro del manejador de Express? a) `req.params.q` b) `req.query.q` c) `req.body.q` d) `req.search.q`

PREGUNTA 43: (V/F) Un middleware en Express es una función que recibe los parámetros (`req`, `res`, `next`) y que puede pasar el control al siguiente middleware o ruta llamando a `next()`.

PREGUNTA 44: (V/F) La instrucción `app.use(express.static("public"))` configura Express para servir archivos estáticos como imágenes, hojas de estilo CSS y scripts desde la carpeta `"public"`.

PREGUNTA 45: Escribe un middleware de Express que, por cada petición que llegue al servidor, imprima en la consola la fecha y hora actual, el método HTTP y la URL solicitada, y que después permita que la petición continúe hacia la ruta correspondiente usando `next()`. Incluye también la línea para registrarlo en la aplicación.

PREGUNTA 46: En una API REST, ¿qué método HTTP se utiliza convencionalmente para crear un recurso nuevo en el servidor? a) GET b) POST c) PUT d) DELETE

PREGUNTA 47: ¿Qué código de estado HTTP indica que una petición de creación de recurso fue exitosa y que el recurso nuevo ya existe en el servidor? a) 200 OK b) 201 Created c) 404 Not Found d) 500 Internal Server Error

PREGUNTA 48: (V/F) La función `fetch()`, disponible en los navegadores modernos y en Node.js desde la versión 18, devuelve una promesa que se resuelve con el objeto de respuesta HTTP.

PREGUNTA 49: Completa el siguiente endpoint de Express para que cree un usuario nuevo. Debe tomar los datos del cuerpo de la petición, agregarlos al arreglo "usuarios" y responder con el código de estado 201 junto con el usuario creado en formato JSON. Escribe las líneas que faltan.

```
const express = require("express");
const app = express();

app.use(express.json());

const usuarios = [];

app.post("/usuarios", (req, res) => {
  // FALTA LÍNEA 1: obtener los datos del usuario desde req.body
  // FALTA LÍNEA 2: agregar el usuario al arreglo usuarios
  // FALTA LÍNEA 3: responder con estado 201 y el usuario en JSON
});

app.listen(3000);
```

PREGUNTA 50: Explica con tus propias palabras los pasos necesarios para desplegar (publicar en internet) una aplicación de Node.js con Express en un servicio en la nube como Render o Railway. Menciona al menos cuatro pasos concretos, por ejemplo lo relacionado con el repositorio de código, el archivo package.json, las variables de entorno y el puerto de escucha.

Respuestas y soluciones

Aquí están las 50 respuestas correctas, cada una con su explicación didáctica. No te limites a ver si acertaste o fallaste: lee cada explicación completa, incluso las de las preguntas que respondiste bien, porque contienen detalles, consejos y advertencias sobre errores comunes que refuerzan lo aprendido. Si fallaste una pregunta, vuelve a la unidad correspondiente del libro, relea el tema y trata de escribir el ejemplo de código por tu cuenta antes de seguir. Recuerda que cada respuesta incluye al final un consejo práctico marcado como Tip, Dato, Recuerda o Cuidado: esos consejos resumen la lección más valiosa de cada pregunta. Si quieres subir de nivel, elige las tres preguntas donde fallaste más y vuelve a escribir su código desde cero sin mirar la solución, hasta que te salga de forma natural. Esa es la diferencia entre haber leído sobre Node.js y saber Node.js de verdad. Repite este ciclo de examen, repaso y práctica hasta que el puntaje deje de importarte porque la confianza ya la tienes.

RESPUESTA 1: b). Explicación: Node.js es un entorno de ejecución que permite correr JavaScript fuera del navegador, típicamente del lado del servidor, y por eso la opción b) es la correcta. La opción a) es incorrecta porque Node.js no es un framework de interfaces como React o Vue, sino la plataforma base sobre la que corren esos programas. La opción c) es incorrecta porque Node.js no es un manejador de bases de datos, aunque puede conectarse a ellas. La opción d) es incorrecta porque Node.js no es un lenguaje nuevo: el lenguaje sigue siendo JavaScript. Consejo: recuerda que Node.js usa el mismo lenguaje JavaScript del navegador, así que todo lo que ya sabes de JavaScript te sirve tal cual aquí.

RESPUESTA 2: b). Explicación: Node.js utiliza el motor V8, desarrollado por Google para el navegador Chrome, y por eso la opción b) es la correcta. V8 es el encargado de interpretar y ejecutar el código JavaScript con alto rendimiento. La opción a) SpiderMonkey es el motor de Firefox, la opción c) JavaScriptCore es el motor de Safari y la opción d) Chakra fue el motor clásico de Internet Explorer y Edge antiguo. Ninguno de esos tres tiene relación con Node.js. Dato útil: V8 también es el motor de Chrome, por eso el JavaScript corre igual de rápido en el servidor que en el navegador.

RESPUESTA 3: Falso. Explicación: La afirmación es falsa porque Node.js ejecuta el código JavaScript directamente, sin necesidad de compilarlo antes a binario nativo con gcc ni con ningún otro compilador. El motor V8 se encarga de interpretar el código y, además, lo compila de forma interna y transparente a código máquina mediante compilación justo a tiempo (JIT). Por eso basta con escribir un archivo .js y ejecutarlo con el comando node. Recuerda: la compilación justo a tiempo la hace V8 por dentro; tú solo escribes JavaScript normal y lo ejecutas con node.

RESPUESTA 4: Verdadero. Explicación: La afirmación es verdadera porque el comando "node -v" le pide al intérprete de Node.js que reporte su número de versión, por ejemplo "v20.11.0". Si el comando muestra la versión, significa que Node.js está instalado y disponible en la ruta del sistema. Si en cambio la terminal responde que el comando no existe, la instalación no se completó o no quedó registrada en el PATH. Tip: también puedes usar node --versión, que es lo mismo que node -v pero escrito con la palabra completa.

RESPUESTA 5: Los comandos son: 1) mkdir mi-proyecto, 2) cd mi-proyecto, 3) crear app.js con el contenido console.log("Hola Node"); (se puede crear con un editor o con echo), 4) node app.js. Explicación: El comando mkdir crea la carpeta y cd entra en ella, lo cual organiza el proyecto en su propio espacio. Después se crea el archivo app.js con una sola línea que usa console.log, la función más básica para imprimir en Node.js. Finalmente, node app.js le ordena al intérprete ejecutar ese archivo, y en la consola

debe aparecer el texto "Hola Node". Práctica estos cuatro comandos hasta que te salgan de memoria, porque los usaras al iniciar cada proyecto nuevo.

RESPUESTA 6: b). Explicación: La forma correcta en CommonJS, el sistema de módulos clásico de Node.js, es `const fs = require("fs")`, por eso la opción b) es correcta. La opción a) usa la sintaxis `import`, que pertenece a los módulos ES (ESM) y no a CommonJS, así que es incorrecta en este contexto. Las opciones c) y d) usan palabras como `include` y `using` que no existen en JavaScript ni en Node.js para importar módulos. Nota: en proyectos modernos también verás `import` de ESM, pero `require` sigue siendo el sistema clásico de Node.js.

RESPUESTA 7: b). Explicación: El objeto global `process` representa el proceso actual de Node.js, y su propiedad `argv` es un arreglo con los argumentos de la línea de comandos, por eso la opción b) es correcta. La opción a) es incorrecta porque `window` no existe en Node.js, solo en los navegadores. Las opciones c) y d) son incorrectas porque ni `global.args` ni `console.argv` existen como API para leer argumentos. Recuerda que `process.argv[0]` es la ruta de node y `process.argv[1]` la de tu archivo; tus argumentos empiezan en el índice 2.

RESPUESTA 8: Verdadero. Explicación: La afirmación es verdadera porque `__dirname` es una variable disponible en cada módulo de CommonJS y siempre contiene la ruta absoluta del directorio donde está guardado el archivo que se ejecuta. Esto es muy útil para construir rutas a archivos del proyecto sin depender de desde dónde se invocó el comando node. No debe confundirse con `process.cwd()`, que devuelve la carpeta desde donde se lanzó el proceso, la cual puede ser diferente. Tip: combina `__dirname` con `path.join` para construir rutas que funcionen en cualquier sistema operativo.

RESPUESTA 9: El error es que el método se llama `os.platform()` y no `os.plataforma()`. Explicación: Los métodos del módulo `os` están en inglés, así que `os.plataforma()` no existe y al ejecutarlo Node.js lanza un `TypeError` indicando que no es una función. La versión corregida es `console.log("Plataforma: " + os.platform());`, que devuelve textos como "linux", "darwin" o "win32". La lección es consultar siempre la documentación oficial en inglés cuando un método "no existe". Consejo: cuando veas el error de que algo no es una función, casi siempre es un nombre mal escrito o un módulo mal importado.

RESPUESTA 10: Línea 1 en `saludo.js`: `module.exports = saludar`; (también es válido `module.exports.saludar = saludar`);. Línea 2 en `app.js`: `const saludar = require("./saludo");`. Explicación: En CommonJS se expone lo que otro archivo puede usar asignándolo a `module.exports`, y se importa con `require` indicando la ruta relativa

que empieza con `./` . Si se olvida el `./` , Node.js buscará un módulo instalado en `node_modules` en lugar del archivo local. Con esas dos líneas, `saludar("Esteban")` imprime "Hola Esteban". Práctica este patrón de exportar e importar, porque es la base para organizar cualquier proyecto en varios archivos.

RESPUESTA 11: b). Explicación: El método correcto es `fs.readFile`, que lee el archivo de forma asíncrona y avisa mediante un callback cuando termina, sin bloquear el resto del programa, por eso la opción b) es correcta. La opción a) `fs.readFileSync` es la versión síncrona y sí bloquea la ejecución hasta terminar la lectura. Las opciones c) y d) son incorrectas porque esos métodos no existen en el módulo `fs`. Recuerda: la terminación Sync siempre indica la versión bloqueante; usala solo en scripts cortos o al arrancar el programa.

RESPUESTA 12: b). Explicación: La función `path.join` une los segmentos usando el separador correcto del sistema operativo (diagonal invertida en Windows, diagonal normal en Linux y macOS), por eso la opción b) es correcta. La opción a) es incorrecta porque `path.join` solo construye el texto de la ruta, no crea carpetas en el disco. Las opciones c) y d) son incorrectas porque no verifica existencia ni copia archivos; para eso existen `fs.existsSync` y `fs.copyFile`. Tip: `path.join` también normaliza las diagonales de más, así evitas rutas rotas por errores de escritura.

RESPUESTA 13: Falso. Explicación: La afirmación es falsa porque `fs.writeFile` sí crea el archivo cuando no existe, pero cuando el archivo ya existe no lanza ningún error: simplemente sobrescribe su contenido anterior con el nuevo. Esta es una característica importante que hay que manejar con cuidado, porque se pueden perder datos si se escribe sin querer sobre un archivo existente. Si se quiere evitar la sobrescritura, se puede usar la bandera "wx" al escribir. Cuidado: este comportamiento de sobrescribir hace que `writeFile` sea peligroso si no verificas bien la ruta antes.

RESPUESTA 14: El error es usar `fs.readFile` (asíncrono) como si devolviera el contenido directamente. Explicación: `fs.readFile` no retorna los datos, sino que trabaja con un callback que recibe el resultado cuando la lectura termina, así que la variable `datos` queda con valor `undefined` y se imprime `undefined`. Hay dos correcciones válidas: usar la versión síncrona con `const datos = fs.readFileSync("datos.txt", "utf8");`, o usar el callback con `fs.readFile("datos.txt", "utf8", (err, datos) => console.log(datos));`. La lección es nunca tratar una función asíncrona como si fuera síncrona. Regla de oro: si una función recibe un callback, su valor de retorno casi nunca trae el resultado que buscas.

RESPUESTA 15: Una solución válida es: Tip: en código real también conviene verificar que el contenido no venga vacío antes de procesarlo.

```
const fs = require("fs");
try {
  const contenido = fs.readFileSync("notas.txt", "utf8");
  console.log(contenido);
} catch (error) {
  console.log("No se pudo leer el archivo.");
  console.log("Detalle: " + error.message);
}
```

Explicación: Se usa `fs.readFileSync` dentro de `try` porque la versión síncrona lanza una excepción cuando algo falla, por ejemplo si el archivo no existe. El bloque `catch` atrapa ese error y muestra un mensaje amable en lugar de dejar que el programa se rompa con un error técnico. Mostrar `error.message` ayuda a diagnosticar sin asustar al usuario. Este patrón de `try/catch` es el estándar para operaciones síncronas que pueden fallar.

RESPUESTA 16: b). Explicación: El método correcto es `os.freemem()`, que devuelve la memoria libre en bytes, por eso la opción b) es correcta. La opción a) es incorrecta porque `freeMemory` no existe; el nombre real abrevia "memory" como "mem". Las opciones c) y d) son incorrectas porque `os.memory()` y `os.ram()` no forman parte de la API del módulo `os`. Para la memoria total existe `os.totalmem()`, que conviene conocer junto a esta. Dato: los valores vienen en bytes, así que divídelos entre 1024 dos veces para verlos en megabytes.

RESPUESTA 17: b). Explicación: En un servidor creado con el módulo `http`, el método `res.end()` es el que finaliza la respuesta y la envía al cliente, por eso la opción b) es correcta. La opción a) `res.send()` pertenece a Express, no al `http` puro de Node.js, así que es incorrecta aquí. La opción c) es incorrecta porque `res.write()` solo escribe fragmentos pero no termina la respuesta; si no se llama a `res.end()`, el navegador se queda esperando. La opción d) es incorrecta porque `res.close()` no es el método estándar para este propósito. Recuerda: en Express usarias `res.send`, pero con `http` puro siempre debes cerrar la respuesta con `res.end`.

RESPUESTA 18: Verdadero. Explicación: La afirmación es verdadera porque el método `listen` le ordena al servidor que se ponga a escuchar conexiones entrantes en el puerto indicado, en este caso el 3000. Mientras no se llame a `listen`, el servidor existe pero no atiende a nadie. Es común usar puertos como el 3000 o el 8080 para desarrollo, y en producción se suele usar el puerto que indique la variable de entorno `PORT`. Tip: si el puerto está ocupado verás el error `EADDRINUSE`; cambia el número o cierra el otro programa.

RESPUESTA 19: Al visitar `/saludo` el navegador recibe el código 200 y el texto "Hola desde Node"; al visitar `/` recibe el código 404 y el texto "No encontrado". Explicación: El manejador revisa `req.url` y cuando coincide con `/saludo` ejecuta `res.writeHead(200, ...)` para anunciar éxito y envía el saludo. Para cualquier otra ruta, incluida `/`, entra al `else`, anuncia el código 404 que significa recurso no encontrado y envía ese texto. Esto muestra el patrón básico de enrutamiento manual: comparar la URL y responder distinto según el caso. Práctica este enrutamiento manual una vez, y luego valora lo mucho que Express te simplifica la vida.

RESPUESTA 20: Las líneas faltantes son: Recuerda: sin `JSON.stringify` enviarías el texto `[object Object]`, el error clásico de quien empieza con JSON.

```
res.writeHead(200, { "Content-Type": "application/json" });  
res.end(JSON.stringify(datos));
```

Explicación: Primero se debe anunciar con `writeHead` que el contenido es JSON usando el encabezado `Content-Type: application/json`, para que el cliente sepa cómo interpretar la respuesta. Después se convierte el objeto a texto con `JSON.stringify`, porque `res.end` solo puede enviar texto o buffers, no objetos de JavaScript. Sin la conversión se enviaría el texto `"[object Object]"`, que no sirve para nada.

RESPUESTA 21: b). Explicación: El comando `"npm init -y"` crea el archivo `package.json` aceptando todos los valores por defecto sin hacer preguntas interactivas, por eso la opción b) es correcta. La opción a) es incorrecta porque instalar dependencias es trabajo de `"npm install"`. La opción c) es incorrecta porque npm no actualiza Node.js, solo gestiona paquetes. La opción d) es incorrecta porque ese comando no toca la carpeta `node_modules`. Tip: la bandera `-y` acepta todo por defecto; si quieres personalizar, usa `npm init` sin la `y` y responde las preguntas.

RESPUESTA 22: b). Explicación: `nodemon` vigila los archivos del proyecto y reinicia automáticamente la aplicación cuando detecta un cambio guardado, por eso la opción b) es correcta y es la razón por la que se usa tanto en desarrollo. La opción a) es

incorrecta porque nodemon no compila nada a código máquina. La opción c) es incorrecta porque no hace respaldos en la nube. La opción d) es incorrecta porque no optimiza memoria; su único trabajo es ahorrar el reinicio manual. Nota: nodemon es solo para desarrollo; en producción se usa node directamente o un gestor de procesos.

RESPUESTA 23: Falso. Explicación: La afirmación es falsa porque la carpeta `node_modules` no debe subirse a Git: suele ser muy pesada y su contenido se puede regenerar en cualquier momento con `"npm install"` a partir del `package.json`. La práctica correcta es agregar `node_modules` al archivo `.gitignore` para que Git la ignore. Subirla al repositorio solo hace lento el proyecto y provoca conflictos innecesarios entre computadoras. Recuerda agregar `node_modules` al archivo `.gitignore` desde el primer commit de tu proyecto.

RESPUESTA 24: Verdadero. Explicación: La afirmación es verdadera porque `"npm install"` sin argumentos lee las secciones `dependencies` y `devDependencies` del `package.json` y descarga cada paquete listado, reconstruyendo la carpeta `node_modules`. Esto es lo que permite que un proyecto funcione en una computadora nueva con solo clonar el repositorio y ejecutar ese comando. El archivo `package-lock.json` ayuda a que las versiones instaladas sean las mismas en todos lados. Tip: si borras `node_modules` por accidente, este comando reconstruye todo sin que pierdas nada.

RESPUESTA 25: Un `package.json` válido es: Práctica escribir este archivo a mano una vez, y después deja que `npm init` te ayude con la estructura.

```
{
  "name": "mi-api",
  "version": "1.0.0",
  "description": "API de ejemplo del curso de Node.js",
  "main": "server.js",
  "scripts": {
    "start": "node server.js"
  },
  "dependencies": {
    "express": "^4.18.2"
  }
}
```

Explicación: El campo `name` identifica el proyecto y `versión` indica su versión inicial. La sección `scripts` define atajos de comandos, y `"start"` con `"node server.js"` permite arrancar con `"npm start"`. En `dependencies` se declara `express` con el símbolo `^`, que permite actualizaciones

menores compatibles sin romper el código. Con este archivo, cualquier persona puede instalar todo con "npm install".

RESPUESTA 26: b). Explicación: El método correcto es `emitter.on("evento", función)`, que registra la función como escucha de ese nombre de evento, por eso la opción b) es correcta. La opción a) es incorrecta porque `listen` se usa en servidores HTTP, no en emisores de eventos. Las opciones c) y d) son incorrectas porque `add` y `subscribe` no existen en la clase `EventEmitter`. Para emitir el evento después se usa `emitter.emit("evento")`. Recuerda: `on` registra el listener y `emit` dispara el evento; sin `emit`, los listeners nunca se ejecutan.

RESPUESTA 27: b). Explicación: Un Buffer es una estructura para trabajar con datos binarios de forma directa, como el contenido de un archivo o los datos que viajan por la red, por eso la opción b) es correcta. La opción a) es incorrecta porque no tiene nada que ver con la caché del navegador. La opción c) es incorrecta porque un Buffer no es un stream, aunque los streams suelen entregar los datos en forma de buffers. La opción d) es incorrecta porque no es una función de retardo; para eso existe `setTimeout`. Dato: con `Buffer.from` puedes crear un buffer desde texto, útil para convertir entre texto y binario.

RESPUESTA 28: Verdadero. Explicación: La afirmación es verdadera porque esa es precisamente la ventaja de los streams: van entregando los datos en pedazos pequeños (chunks) a medida que están disponibles, en lugar de esperar a tener todo el contenido. Esto permite procesar archivos gigantes, como videos o respaldos, sin agotar la memoria RAM. Ejemplos típicos son `fs.createReadStream` y la respuesta de un servidor que envía datos por partes. Tip: `pipe` es tu mejor amigo con streams, porque conecta lectura y escritura con manejo de flujo automatico.

RESPUESTA 29: El orden de impresión es: "Recibido: Hola mundo" y después "Fin". Explicación: Primero se registra el listener con `on`, luego `emit` dispara el evento de forma síncrona, así que la función del evento se ejecuta inmediatamente e imprime "Recibido: Hola mundo". Después continúa la ejecución normal e imprime "Fin". Es importante notar que `emit` es síncrono: los listeners corren en ese mismo momento, no se ponen en cola como los callbacks de temporizadores o promesas. Recuerda: `emit` es síncrono, a diferencia de casi todo lo demás en Node.js que es asíncrono.

RESPUESTA 30: Las líneas faltantes son: Práctica este ejemplo con un archivo grande y compara la memoria usada contra la versión con `readFileSync`.


```
const escritura = fs.createWriteStream("salida.txt");
lectura.pipe(escritura);
```

Explicación: Se crea el stream de escritura con `createWriteStream` apuntando al archivo destino. Después el método `pipe` conecta la salida del stream de lectura con la entrada del de escritura, de modo que cada fragmento leído se escribe automáticamente. Este patrón es eficiente porque nunca carga el archivo completo en memoria y maneja solo el flujo de datos. También se podrían escuchar los eventos "error" en ambos streams para un programa más robusto.

RESPUESTA 31: b). Explicación: El orden correcto es 1, 4, 3, 2, por eso la opción b) es correcta. Primero se imprimen "1" y "4" porque son código síncrono que corre de inmediato. Después, entre las tareas pendientes, las promesas (microtareas) tienen prioridad sobre los temporizadores (macrotareas), así que el `then` de la promesa imprime "3" antes de que el `setTimeout`, aunque tenga 0 milisegundos, imprima "2". Este es el comportamiento clásico del event loop de Node.js. Memoriza este orden: código síncrono primero, luego microtareas de promesas, luego macrotareas de temporizadores.

RESPUESTA 32: b). Explicación: La palabra clave correcta es `await`, que pausa la función `async` hasta que la promesa se resuelve y entrega su valor, por eso la opción b) es correcta. La opción a) `wait` no existe en JavaScript. La opción c) `async` se usa para declarar la función, no para esperar dentro de ella. La opción d) `then` es el método clásico de las promesas, pero dentro de `async/await` se reemplaza por `await` para un código más legible. Recuerda: `await` solo funciona dentro de funciones `async` o en el nivel superior de módulos ES.

RESPUESTA 33: Verdadero. Explicación: La afirmación es verdadera porque toda función `async` envuelve su valor de retorno en una promesa de forma automática. Si se retorna un número o un texto, la promesa se resuelve con ese valor; si se lanza un error, la promesa se rechaza. Por eso para obtener el valor real hay que usar `await` o `.then()` sobre el resultado. Este detalle explica por qué no se puede simplemente asignar el retorno de una función `async` a una variable y usarlo directamente. Tip: por eso siempre debes usar `await` o `.then` al llamar una función `async` si necesitas su valor real.

RESPUESTA 34: Falso. Explicación: La afirmación es falsa porque el callback de `fs.readFile` se ejecuta después de que la lectura del archivo termina, nunca antes. Node.js delega la lectura al sistema operativo y sigue con el resto del programa; cuando los datos están listos, el event loop coloca el callback en la cola y lo ejecuta. Si el callback corriera antes, los datos aún no existirían y no tendría nada que entregar.

Recuerda: este modelo no bloqueante es lo que permite a Node.js atender miles de conexiones a la vez.

RESPUESTA 35: La versión con `async/await` es: Práctica convertir callbacks a `async/await`, porque el código moderno de Node.js usa este estilo en todos lados.

```
const fs = require("fs/promises");

async function leerArchivo() {
  try {
    const data = await fs.readFile("datos.txt", "utf8");
    console.log(data);
  } catch (err) {
    console.error("Error al leer");
  }
}

leerArchivo();
```

Explicación: Se importa `"fs/promises"`, que ofrece las mismas funciones pero devolviendo promesas en lugar de pedir callbacks. Dentro de una función `async` se usa `await` para esperar la lectura y el resultado queda en la variable `data` como si fuera código síncrono. El `try/catch` reemplaza la revisión manual del parámetro `err` del callback y atrapa cualquier falló de lectura. El resultado es un código más lineal y fácil de leer.

RESPUESTA 36: a). Explicación: La forma correcta en Express es `app.get("/ruta", manejador)`, donde el primer argumento es la ruta y el segundo la función que responde, por eso la opción a) es correcta. La opción b) es incorrecta porque `app.route` se usa de otra manera y no recibe el método como primer argumento así. La opción c) es incorrecta porque express es el módulo importado, no la aplicación creada con `express()`. La opción d) es incorrecta porque `app.fetch` no existe en Express. Tip: Express también tiene `app.post`, `app.put` y `app.delete` con la misma firma para los otros métodos HTTP.

RESPUESTA 37: b). Explicación: El método `res.json` convierte el objeto a texto JSON, pone automáticamente el encabezado `Content-Type: application/json` y envía la respuesta, por eso la opción b) es correcta. La opción a) es incorrecta porque no guarda nada en disco. La opción c) es incorrecta porque no imprime en consola; para eso está `console.log`. La opción d) es incorrecta porque no valida el objeto, solo lo serializa y lo envía tal cual. Recuerda: `res.json` termina la respuesta por sí solo, no necesitas llamar a `res.end` después.

RESPUESTA 38: Verdadero. Explicación: La afirmación es verdadera porque Express no interpreta el cuerpo de las peticiones por sí solo; necesita el middleware `express.json()` (o `express.urlencoded()` para formularios) registrado con `app.use`. Una vez configurado, los datos enviados en el POST quedan disponibles en `req.body` como un objeto de JavaScript. Si se olvida registrar el middleware, `req.body` queda como `undefined` y los datos se pierden. Cuidado: sin el middleware de interpretación, `req.body` queda `undefined` y tu endpoint recibirá datos vacíos.

RESPUESTA 39: Verdadero. Explicación: La afirmación es verdadera porque Express está construido sobre el módulo `http` de Node.js, y `app.listen` es simplemente un atajo que crea el servidor HTTP y lo pone a escuchar en el puerto indicado. Por debajo hace lo mismo que `http.createServer(app).listen(3000)`. La ventaja de Express no está en cómo escucha, sino en el sistema de rutas, middlewares y utilidades que agrega encima. Dato: por eso puedes mezclar código `http` puro con Express si alguna vez lo necesitas en un proyecto.

RESPUESTA 40: El error es que falta la línea `app.listen(3000)`; al final. Explicación: El programa define la ruta `/usuarios` correctamente, pero nunca pone el servidor a escuchar en ningún puerto, así que al ejecutarlo el proceso termina de inmediato sin atender peticiones. La versión corregida agrega `app.listen(3000, () => console.log("Servidor en puerto 3000"))`; después de las rutas. Es un error muy común en quienes empiezan: definir rutas sin levantar el servidor equivale a escribir un menú sin abrir el restaurante. Consejo: pon siempre el `app.listen` al final del archivo, después de definir todas tus rutas y middlewares.

RESPUESTA 41: a). Explicación: Los segmentos dinámicos de la ruta definidos con dos puntos, como `:id`, quedan disponibles en el objeto `req.params`, por eso la opción a) es correcta y `req.params.id` vale `"42"`. La opción b) es incorrecta porque `req.query` sirve para los parámetros después del signo `?` en la URL. La opción c) es incorrecta porque `req.body` contiene datos del cuerpo de la petición, no de la ruta. La opción d) es incorrecta porque `req.id` no existe. Tip: `req.params` siempre entrega textos; si necesitas un número, conviértelo con `Number()`.

RESPUESTA 42: b). Explicación: Los parámetros que vienen después del signo `?` en la URL se llaman query strings y Express los entrega ya interpretados en el objeto `req.query`, por eso la opción b) es correcta y `req.query.q` vale `"node"`. La opción a) es incorrecta porque `req.params` es para segmentos de la ruta como `:q`. La opción c) es incorrecta porque `req.body` es para el cuerpo de la petición. La opción d) es incorrecta porque `req.search` no existe en Express. Recuerda: Express ya interpreta los query strings por ti, no necesitas hacerlo a mano.

RESPUESTA 43: Verdadero. Explicación: La afirmación es verdadera porque esa es exactamente la firma de un middleware en Express: una función con (req, res, next) que se ejecuta en el orden en que fue registrada. Si el middleware no termina la respuesta con res.send o similar, debe llamar a next() para pasar el control al siguiente middleware o a la ruta final. Si olvida llamar a next() y tampoco responde, la petición se queda colgada esperando. Cuidado: olvidar next() es la causa más común de peticiones que se quedan colgadas sin respuesta.

RESPUESTA 44: Verdadero. Explicación: La afirmación es verdadera porque express.static es el middleware incluido en Express para servir archivos estáticos, y al registrarlo con app.use le indica desde qué carpeta tomarlos. Con esa línea, un archivo como public/estilos.css queda accesible en la URL /estilos.css. Es la forma estándar de publicar imágenes, CSS y JavaScript del lado del cliente sin escribir una ruta para cada archivo. Tip: por seguridad, nunca pongas archivos privados dentro de la carpeta de archivos publicos.

RESPUESTA 45: Un middleware válido es: Práctica este middleware de registro, porque casi todo proyecto real lleva uno parecido.

```
function registrarPeticiones(req, res, next) {  
  const ahora = new Date().toISOString();  
  console.log(ahora + " " + req.method + " " + req.url);  
  next();  
}  
  
app.use(registrarPeticiones);
```

Explicación: La función recibe req, res y next, extrae el método con req.method y la URL con req.url, y los imprime junto con la fecha actual. Después llama a next() para no detener la petición, lo cual es obligatorio en un middleware que solo observa. Finalmente se registra con app.use para que se ejecute en todas las peticiones antes de las rutas. Este es el middleware más típico para llevar un registro de actividad del servidor.

RESPUESTA 46: b). Explicación: Por convención REST, el método POST se usa para crear recursos nuevos enviando los datos en el cuerpo de la petición, por eso la opción b) es correcta. La opción a) GET es incorrecta porque solo lee datos y no debe modificar nada. La opción c) PUT se usa para reemplazar un recurso existente, no para crearlo desde cero. La opción d) DELETE se usa para eliminar recursos. Recuerda el mapa mental REST: GET lee, POST crea, PUT reemplaza y DELETE elimina.

RESPUESTA 47: b). Explicación: El código 201 Created significa que la petición fue exitosa y que se creó un recurso nuevo como resultado, por eso la opción b) es correcta y es el estándar para responder a un POST de creación. La opción a) 200 indica éxito genérico pero no comunica que algo fue creado. La opción c) 404 indica que no se encontró el recurso. La opción d) 500 indica un error interno del servidor. Tip: acompaña el 201 devolviendo el recurso creado, como hacen las buenas APIs.

RESPUESTA 48: Verdadero. Explicación: La afirmación es verdadera porque fetch está diseñado con promesas: devuelve una promesa que se resuelve con el objeto Response cuando llegan los encabezados de la respuesta HTTP. Después, métodos como response.json() o response.text(), que también devuelven promesas, permiten leer el cuerpo. Por eso el uso típico es await fetch(url) dentro de una función async, o encadenar .then(). Recuerda: fetch solo rechaza su promesa por errores de red, no por códigos como 404 o 500.

RESPUESTA 49: Las líneas faltantes son: Práctica este endpoint y pruebalo con curl o Postman enviando un JSON de prueba.

```
const nuevoUsuario = req.body;
usuarios.push(nuevoUsuario);
res.status(201).json(nuevoUsuario);
```

Explicación: Primero se extraen los datos enviados por el cliente desde req.body, lo cual requiere que el middleware express.json() esté registrado. Después se agregan al arreglo con push, simulando el guardado en una base de datos. Finalmente se responde con res.status(201) para indicar creación exitosa y .json() para devolver el usuario creado. Este es el esqueleto de cualquier endpoint POST de una API REST.

RESPUESTA 50: Respuesta modelo: 1) Subir el código a un repositorio de Git, por ejemplo en GitHub, sin incluir `node_modules` gracias al `.gitignore`. 2) Verificar que el `package.json` tenga el script "start" y las dependencias correctas, porque el servicio ejecutará "npm install" y "npm start" automáticamente. 3) Configurar las variables de entorno en el panel del servicio (como claves de API o la cadena de conexión a la base de datos) en lugar de escribirlas en el código. 4) Hacer que el servidor escuche en el puerto de la variable de entorno PORT con `const puerto = process.env.PORT || 3000;`, porque la nube asigna el puerto y no siempre es el 3000. Explicación: Estos cuatro pasos cubren lo esencial del despliegue moderno: el código viaja por Git, las dependencias se reinstalan solas, los secretos quedan fuera del código y el puerto es dinámico. Si alguno falla, el despliegue típicamente muestra el error en los registros del servicio para poder corregirlo. Tip: Render y Railway detectan un proyecto Node.js automáticamente si el `package.json` está bien hecho.

Felicidades por terminar el examen final. Independientemente de tu puntaje, el simple hecho de haber recorrido las 10 unidades del libro y enfrentarte a estas 50 preguntas ya te pone por delante de la mayoría de las personas que dicen querer aprender a programar. Node.js es una herramienta poderosa y tú ya conoces sus fundamentos, su modelo asíncrono, sus módulos principales y cómo construir servidores y APIs REST con Express.

Si este examen te dejó con ganas de más, el siguiente paso natural es construir un proyecto completo que integre todo: por ejemplo, una API REST con operaciones CRUD conectada a una base de datos, con autenticación de usuarios y desplegada en un servicio en la nube. Ese proyecto será tu carta de presentación y la prueba de que dominas lo aprendido. También puedes explorar temas avanzados como WebSockets para aplicaciones en tiempo real, pruebas automatizadas con frameworks de testing, o el trabajo con bases de datos como MongoDB y PostgreSQL.

Guarda este examen y vuelve a presentarlo dentro de unos meses. Te sorprenderá lo fácil que te parecerán las preguntas que hoy te costaron trabajo. Así es cómo se mide el progreso real en la programación: no con un número, sino con la soltura con la que resuelves lo que antes te parecía imposible. Y si puedes, comenta tus respuestas con alguien más que también esté aprendiendo: explicar en voz alta por qué una opción es correcta y las demás no es una de las formas más efectivas de fijar el conocimiento. Enseñar lo que sabes, aunque sea a un compañero que va empezando, te obliga a ordenar tus ideas y revela los huecos que todavía tienes. Sigue construyendo, sigue practicando y nos vemos en el siguiente curso de Lucas EGA Academy, con más proyectos reales.

Colibrí Editorial EGA

Curso Completo de Node.js

Curso completo de Node.js: módulos, archivos, npm, asincronía, Express, middleware y APIs REST, con proyecto final de backend y examen final.

lucasegaacademy.com

© 2026 Esteban Gutiérrez · Colibrí Editorial EGA · Hecho en México