How Node.js Really Works

V8 runs your JavaScript. libuv talks to the OS. Node.js bindings connect them.
Here's how all three fit together — and where to read the official docs.

THE JS ENGINE

V8 — The JavaScript Machine

V8 is Google's open-source JavaScript engine written in C++. It takes your JavaScript code and turns it into machine code the CPU can actually run. Node.js uses V8 as its core engine — the same one that powers Chrome. When Node.js releases a new version, it ships a newer version of V8 inside it, which is why new JS language features (like optional chaining or top-level await) become available in Node.js after they land in V8 first.

WHAT V8 DOES

Your JS Code → Parse & AST → Bytecode (Ignition) → Machine Code ✅

💡 ***JIT (Just-In-Time) Compilation*** *— V8 doesn't compile everything upfront. It starts with bytecode (Ignition interpreter) and compiles hot code (frequently run code) to fast machine code on the fly using the* ***TurboFan*** *optimising compiler. Cold code stays as bytecode to save memory.*

V8 ALSO MANAGES MEMORY

- ✅ Heap — object storage
- ✅ Call Stack — execution
- ✅ Garbage Collection
- ✅ JIT Compilation

WHAT V8 DOES NOT KNOW

- ❌ File system
- ❌ Network requests
- ❌ Timers
- ❌ DNS lookups
- ❌ OS operations

💡 *V8 is a pure JS execution engine — it only runs logic. Everything else (I/O, timers, network) is handled outside of V8 by libuv and the OS layer.*

THE ASYNC POWERHOUSE

libuv — The C Library That Does the Heavy Lifting

libuv is an open-source C library originally built for Node.js. It gives Node.js the ability to talk to the operating system asynchronously — things V8 simply cannot do. This is where the real async magic lives.

WHAT LIBUV PROVIDES

- 🔄 Event Loop
- 🧵 Thread Pool
- 📁 File System I/O
- 🌐 Network I/O
- ⏱ Timers
- 🔍 DNS Lookup
- 🖥️ OS-level Async
- 🔌 TCP / UDP sockets

THREAD POOL EXPLAINED

- → Node.js is single-threaded for JS code
- → But libuv uses a **thread pool of 4 threads** by default for heavy I/O tasks
- → These run in the background without blocking JS
- → When done, they notify the Event Loop
- → Thread pool size can be changed via `UV_THREADPOOL_SIZE`
- → For network I/O, libuv uses the OS's native async APIs (epoll/kqueue/IOCP) — **no thread needed**

SIDE BY SIDE

V8 vs libuv — Two Different Jobs

⚙️ V8 — JS Engine (C++)

- → Written in **C++** by Google
- → Parses and **executes JavaScript**
- → Manages **Heap** (objects) and **Call Stack**
- → Performs **garbage collection**
- → Compiles JS to **machine code via JIT**
- → Knows **nothing about I/O**

🔧 libuv — Async I/O Library (C)

- → Written in **C**, cross-platform
- → Powers the **Event Loop**
- → Manages the **Thread Pool**
- → Handles **file, network, DNS** operations
- → Communicates directly with the **OS**
- → Knows **nothing about JavaScript**

OFFICIAL REFERENCE

Node.js Official Documentation

The Node.js docs at **nodejs.org/docs/latest/api/** document every built-in module that Node exposes to your JS code — these are all implemented via the Bindings + libuv layers under the hood.

📁 fs
File system operations. `readFile`, `writeFile`, streams — all route through libuv's thread pool to the OS.

🌐 http / https
Create servers and make requests. Uses libuv's TCP socket layer and OS-level async network I/O.

⏱ timers
`setTimeout`, `setInterval`, `setImmediate` — all managed by libuv's event loop timer phase.

🔐 crypto
Hashing, encryption, TLS. Heavy crypto ops offloaded to libuv's thread pool to avoid blocking the event loop.

🔌 net / dns
Raw TCP/UDP sockets and DNS resolution. DNS uses libuv's own resolver backed by the thread pool.

🔄 stream
Readable, Writable, Transform streams. Abstract interface over libuv I/O handles for efficient data flow.

📖 `nodejs.org/docs/latest/api/`

How Node.js Really Works

THE BRIDGE

Node.js Bindings — C++ Glue Code

V8 speaks JavaScript. libuv speaks C. They can't talk to each other directly. **Node.js Bindings** are C++ code that act as a translator between the two — they expose libuv's capabilities as JavaScript functions you can call.

What Bindings Do

- → Wrap C/C++ functions so JS can call them
- → Translate JS objects ↔ C++ structs
- → Expose `fs` , `http` , `crypto` modules
- → Bridge V8's heap with libuv's thread pool
- → Built using **Node.js Addon API (N-API)**

When you write `fs.readFile()` in JS, you're actually calling C++ binding code that talks to libuv, which asks the OS to read the file. The JS function is just a thin wrapper over C++ which is itself a wrapper over C.

FULL ARCHITECTURE

The Complete Node.js Stack

Your JavaScript Code
The code you write — `fs.readFile()` , `http.get()` , timers, logic

↕

Node.js Bindings (C++)
Translates JS calls into C++ — bridges V8 and libuv
fs module · http module · crypto · net

↕

V8 Engine (C++)
Executes JS, manages heap, call stack and garbage collection
JIT Compiler · Heap · Call Stack

↕

libuv (C)
Event Loop, Thread Pool, async I/O — talks directly to the OS
Event Loop · Thread Pool · Async I/O

↕

Operating System
The actual file system, network stack, DNS resolver
Linux · macOS · Windows

REAL WORLD FLOW

What Happens When You Call fs.readFile()?

Tracing a single file read from your JS code all the way down to the OS and back.

1. **Your JS calls** `fs.readFile("data.txt", callback)`
   V8 executes this line. The call stack receives it. JS doesn't know how to read files — it hands it off immediately to the binding layer.
2. **Node.js Bindings receive the call**
   The C++ binding layer translates the JS call into a C++ instruction that libuv understands. Your callback reference is registered and stored.
3. **libuv assigns the task to a Thread Pool worker**
   libuv picks a free thread from its pool (default: 4) and sends the file read request to the OS. The JS main thread continues — it is not blocked.
4. **OS reads the file and returns data to libuv**
   The operating system does the actual disk I/O and returns the file contents to the libuv worker thread that requested it.
5. **Event Loop picks up the result and runs your callback**
   libuv signals the Event Loop that the task is done. The loop waits for the Call Stack to be empty, then pushes your `callback(data)` and V8 runs it.

THE ECOSYSTEM

Bun, npm, Deno — How They Relate to Node.js Internals

npm, Bun, and Deno each have a different relationship with V8 and libuv under the hood.

Node.js
JS ENGINE V8 (Google)
ASYNC I/O libuv (C)
The original runtime. All others are inspired by or built on this foundation.

npm
JS ENGINE V8 (via Node.js)
ASYNC I/O libuv (via Node.js)
A JS CLI app that *runs inside* Node.js — no engine of its own.

Bun
JS ENGINE JavaScriptCore (Apple)
ASYNC I/O Zig-based I/O
Full replacement runtime. Node-compatible API with bundler & test runner built in.

Deno
JS ENGINE V8 (Google)
ASYNC I/O Tokio (Rust)
Keeps V8, replaces libuv with Tokio. TypeScript built-in, Web APIs, permissions model.