TUTORIAL · JAVASCRIPT · NODE.JS

How To Start Thinking In Server-Side Javascript

If you already know JavaScript, Node.js isn't a new language — it's the same one, cut loose from the browser. This is what that means, why it matters, and how to start thinking in server-side JavaScript.

By Avalynn Circe

For a long time, JavaScript had a boundary. You wrote it in the browser. It ran in the browser. It could manipulate the page, respond to clicks, make network requests, animate things. But when the user closed the tab, JavaScript stopped. There was no JavaScript on the server, no JavaScript reading files from a hard drive, no JavaScript deciding what to put in a database. Those jobs belonged to other languages — PHP, Ruby, Python, Java.

In 2009, a developer named Ryan Dahl made a decision that seemed strange at the time: he took V8, the JavaScript engine Google had built for Chrome, and ran it outside the browser. He connected it to the operating system, gave it access to the file system and the network, and called the result Node.js.

The boundary disappeared. JavaScript could now run anywhere.

If you've been writing JavaScript for the browser and you're wondering what Node actually is and why you should care, this is where we start.

The Same Language, A Different Environment

The most important thing to understand about Node.js is that it is not a new language. There is no "Node syntax" to learn. The JavaScript you already know — functions, objects, arrays, promises, async/await, destructuring, modules — all of it works exactly the same way.

What changes is the environment. In the browser, JavaScript runs inside a sandbox. The browser decides what it can access: the DOM, the window object, localStorage, the fetch API, Web Workers. JavaScript can't reach outside that sandbox. It can't read a file off your hard drive. It can't listen on a network port. The browser won't allow it.

Node removes the sandbox. In its place, it gives JavaScript access to the operating system — through a set of built-in modules called the Node standard library. Instead of a window object, you get a process object. Instead of the DOM, you get modules for reading files, creating servers, making system calls, and managing child processes.

The browser gives you `document` and `window`. Node gives you `fs` and `http`. Different tools for a different job.

> ***Node.js isn't a new language. It's the same JavaScript, with different tools plugged in.***

Your First Five Minutes With Node

If you have Node installed, open a terminal. That's it — there's no browser, no HTML file, no script tag. Type this:

```
node
```

You're now in a Node REPL — a read-eval-print loop, which is a fancy name for an interactive JavaScript console. Anything you'd type in the browser's DevTools console works here too. Try it:

```
> const greet = name => `Hello, ${name}`;
> greet('world');
'Hello, world'
```

Now exit the REPL and create a file called hello.js with one line:

```
console.log('Hello from Node');

// Run it:
// node hello.js
```

That's a Node program. No browser, no HTML, no build step. Just JavaScript running on your machine.

The Thing That Makes Node Different: Non-Blocking I/O

If you've worked with JavaScript in the browser, you already understand async code. You've used setTimeout. You've fetched data and handled it in a .then() callback. You've written async functions with await. You did all of that because JavaScript is single-threaded — it can only do one thing at a time, so operations that take a while (network requests, timers) hand off their work and come back later.

Node applies exactly the same model to server-side operations. Reading a file, querying a database, making an outbound HTTP request — all of these are I/O operations that take time. In languages like Python or Java, a common approach is to block: the program stops and waits for the operation to finish before moving on. Node never blocks. It starts the operation, moves on to whatever is next, and handles the result when it comes back.

Here's what reading a file looks like in Node, and why it's written the way it is:

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

// This does NOT pause the program while the file loads.
// It starts the read and moves on immediately.
fs.readFile('data.txt', 'utf8', (err, contents) => {
  if (err) {
    console.error('Something went wrong:', err.message);
    return;
  }
  console.log(contents);
});

// This runs BEFORE the file is loaded.
console.log('Reading file...');
```

If you've ever been confused by why console.log statements run in a surprising order in browser JavaScript, this is the same mechanism. Node just applies it to everything — including the disk and the network.

The modern way to write this uses promises and async/await, which reads more naturally:

```
const fs = require('fs').promises;

async function readData() {
  try {
    const contents = await fs.readFile('data.txt', 'utf8');
    console.log(contents);
  } catch (err) {
    console.error('Something went wrong:', err.message);
  }
}

readData();
```

Same result, same non-blocking behavior underneath. The await just hides the callback.

> ***Node never blocks. It starts the work, moves on, and picks up the result when it's ready — the same pattern you already use for fetch() in the browser.***

Building a Server in 10 Lines

In the browser, a server is something that exists somewhere else — it's the thing your fetch() calls talk to. In Node, you can be the server. Here's one:

```
const http = require('http');

const server = http.createServer((request, response) => {
  response.writeHead(200, { 'Content-Type': 'text/plain' });
  response.end('Hello from your Node server');
});

server.listen(3000, () => {
  console.log('Server running at http://localhost:3000');
});
```

Save that as server.js and run node server.js. Open a browser and go to http://localhost:3000. You just served a web request with JavaScript.

This is a minimal example — real applications use frameworks like Express or Fastify that add routing, middleware, and a lot of convenience on top of this foundation. But the foundation is worth understanding: an HTTP server in Node is just a function that receives a request and sends back a response. Everything else is built on that.

Modules: require vs. import

One of the first things that trips up browser JavaScript developers moving to Node is the module system. In the browser, you're probably used to ES modules — the import and export syntax that became standard in modern JavaScript:

```
// ES module syntax (browser / modern JS)
import { something } from './module.js';
export const myFunction = () => {};
```

Node was built before ES modules were standardized, so it has its own older module system called CommonJS, which uses require and module.exports:

```
// CommonJS syntax (classic Node)
const something = require('./module');
module.exports = { myFunction };
```

Node now supports both. Which one you'll encounter depends on the project. If a file ends in .mjs or the nearest package.json has "type": "module", it's using ES modules. If it ends in .cjs or there's no type field, it's CommonJS. Most older Node codebases and npm packages use CommonJS. Most newer ones are moving toward ES modules.

You don't need to master both immediately — but when you see require() in Node code and wonder why it isn't import, now you know.

npm: The Part You're Probably Already Using

npm — the Node Package Manager — ships with Node and is almost certainly already part of your workflow even if you've never written a line of server-side code. Every time you've run npm install in a frontend project, you were using Node infrastructure.

On the server side, npm works the same way. You install packages, they land in node_modules, and you require or import them. The difference is that the packages you're reaching for are different — instead of a UI component library or a date formatting utility, you might be installing an HTTP framework, a database client, a JWT library, or a logging tool.

A few packages worth knowing if you're just getting started:

- express — the most widely used HTTP framework for Node. Adds routing, middleware, and request parsing on top of the built-in http module.
- dotenv — loads environment variables from a .env file. The standard way to keep API keys and credentials out of your source code.
- nodemon — automatically restarts your Node process when you save a file. Saves you from manually restarting the server during development.
- jest or vitest — testing frameworks. Writing tests for server-side code works the same way as frontend testing.

What Node Is Good At — And What It Isn't

Node's non-blocking architecture makes it very good at handling lots of simultaneous connections that spend most of their time waiting. A chat application, a real-time notifications service, an API that fans out to multiple other APIs — these are Node's native territory. Because Node never sits around blocking on I/O, a single Node process can handle thousands of concurrent requests without breaking a sweat.

Node is less suited to tasks that are computationally heavy — image processing, video encoding, machine learning inference, number crunching. These are CPU-bound tasks, and because Node is single-threaded, a long-running computation will block the event loop and stop it from handling anything else. Node has tools for dealing with this (worker threads, child processes), but if your application is primarily CPU-heavy, other languages may serve you better.

For the work most JavaScript developers end up doing — building APIs, serving web applications, gluing services together, handling webhooks, running scripts — Node is an excellent fit. And the tooling ecosystem around it, built by over a decade of npm package development, is enormous.

> ***Node is exceptional at handling many things at once. It's less suited to doing one very expensive thing at a time.***

The Mental Shift

The hardest part of moving from browser JavaScript to Node isn't the syntax — that's largely the same. It isn't even the APIs — those you look up as you need them. The

hardest part is the mental shift from thinking about code that responds to user interactions to thinking about code that responds to network requests and manages system resources.

In the browser, your code runs in response to things a person does: they click, they type, they scroll. In Node, your code runs in response to things other programs do: they send an HTTP request, they write to a queue, they trigger a webhook. The event-driven model is identical. The events are just different.

Once that click lands, the rest follows naturally. You already know the language. You already understand async code. You already know how HTTP works from the client side. Node just hands you the other end of the conversation.

Where to Go Next

The best way to learn Node is to build something small that does real I/O — reads a file, makes an API call, serves an HTTP response. Here's a progression that works:

- Write a script that reads a JSON file and prints formatted output to the terminal.
- Build a minimal HTTP server that returns different responses for different URL paths.
- Add Express and build a small REST API with two or three routes.
- Connect it to a database — SQLite is the lowest-friction option for learning.
- Add a .env file with dotenv and move any hardcoded values into environment variables.

At each step, you're adding one new concept to a foundation you're building from scratch. That's more durable than following a tutorial that hands you a completed project.

You already know the language. You're not starting over — you're expanding.

Avalynn Circe is a technical writer and software developer based in St. Paul, MN. She is the author of JavaScript: The Parts, available at leanpub.com/jsparts.