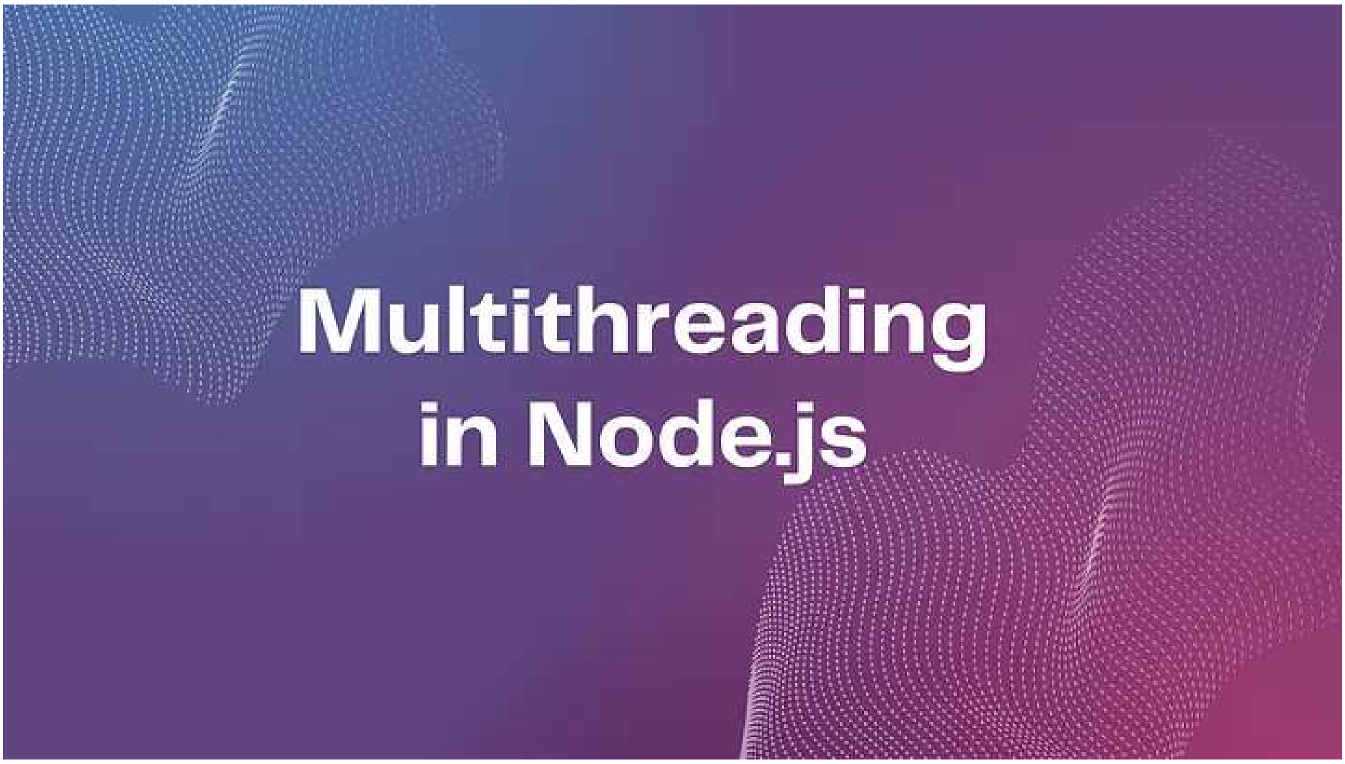
How to: Multithreadi

Matt Stephens · Follow
5 min read · Jan 4

Listen Share



> *All code examples from this article can be found in this GitHub repository.*

You are likely aware of the fact that JavaScript is single-threaded. This means that it can technically only do one thing at a time. Sure, we've got Promises and `async...await` syntax that allow us to wait for things asynchronously (ex. the response of an HTTP request). But when it comes to your JavaScript program actually processing stuff, only one operation can happen at a time. That's not good for scalability.

Consider the `work()` function defined below:

```
// A function that will sleep synchronously for a certain
// number of milliseconds, blocking the main thread.
function sleepSync(milliseconds) {
    const start = Date.now();
    const expire = start + milliseconds;
    while (Date.now() < expire) {}
}

// A function that simulates a heavy blocking workload by
// synchronously sleeping for 2 seconds.
function work() {
    sleepSync(2_000);
}
```

Let’s say that we want to run this function 10 times. That can be done with a simple for-loop:

```
// Run the "work" function 10 times.
console.time('Workflow');

// Run the "work" function 10 times.
for (let i = 0; i <= 10; i++) {
    work();
}

console.timeEnd('Workflow');
```

Output:

```
$ yarn start

  Workflow: 22.000s
```

Looks fine, right? **Nope**. There are some bleeding issues with this code.

- We are running these `work()` calls consecutively — meaning they’re being run one after another. This is **slow**. It would be much more optimal to run them in parallel, but that is not possible on a single thread.

- Our *precious* thread is entirely blocked while this loop run. All other operations will be stuck waiting until the loop is finished, which takes (according to the output above) 22 seconds.

The blocking nature of the logic above can be seen by running this piece of code:

```
function doWork() {
    for (let i = 0; i <= 10; i++) {
        work();
    }
    console.log('Workflow finished.');
}

doWork();
console.log('Hello world!');
```

Because `doWork()` blocks the main thread, the “Hello world!” log (and any other operations) are forced to wait the 20 seconds until it’s finished before they can actually run. The application is slowed to a total crawl.

```
$ yarn start

  Workflow finished.
  Hello world!
```

Multithreading saves the day

To show just how terrible that JavaScript code above actually is, let’s see how Golang handles the same workload:

```
package main

import (
 "fmt"
 "sync"
 "time"
)

// A function that simulates a heavy blocking workload by
```

```
// synchronously sleeping for 2 seconds.
func work() {
 time.Sleep(time.Second * 2)
}

func main() {
 // Create a group to keep track of and wait for our
 // operations.
 var waitGroup sync.WaitGroup
 start := time.Now()

 // Run the "work" function 10 times.
 for i := 1; i <= 10; i++ {
  waitGroup.Add(1)

  // Run the task on a different thread.
  go func() {
   defer waitGroup.Done()
   work()
  }()
 }

 // Wait for all the operations to complete.
 waitGroup.Wait()

 // Comparable to "console.timeEnd" in Javascript
 fmt.Printf("Workflow: %vs\n", time.Since(start).Round(time.Millisecond).Second
}
```

Output:

```
$ go run main.go

  Workflow: 2.007s
```

Despite doing almost the same thing as its JavaScript counterpart, the Golang version only takes around 2 seconds to complete. Why? **Multithreading**. Because of multithreading, both of the bleeding issues the JavaScript code had are eliminated. Not only is the program much faster due to parallelization, but also (and most importantly), other unrelated operations aren't being blocked from running.

You're probably thinking, *"Ok whatever man, Golang is fast. But how does that relate to JavaScript?"*. What the example above demonstrates is how multithreading can make a piece of software extremely scalable, including your Node.js applications.

Multithreading in Node.js

We're going to use the Nanolith NPM package and a few extra lines of code to turn our slow program created at the beginning of this article into a scalable application. The steps are as follows:

Install Nanolith

Using our package manager, we'll first install Nanolith as a dependency:

```
yarn add nanolith
```

Use ESModules (if you're not already)

In order to use Nanolith, you need to use ESModules instead of CommonJS. Luckily, for us, this just means adding the following key-value pair to the top-level of our `package.json`.

```
{
  // ...
  "type": "module",
  // ...
}
```

Create a new file

Now, we need to create a new file dedicated to defining the functions we'd like to multithread. This file can be named anything — let's go with `worker.ts`.

Define your tasks

Instead of defining the `work()` function inside of our main file like before, we need to use the `define()` function (imported from `nanolith`) within the new file we just created:

```
/* File: worker.js */
import { define } from 'nanolith';
```

```
// Create and export a variable with a name of your choice
// that points to the awaited result of defining your tasks.
export const api = await define({
    // Define the work function inside of the object.
    work() {
        sleepSync(2_000);
    },
});

function sleepSync(milliseconds) {
    const start = Date.now();
    const expire = start + milliseconds;
    while (Date.now() < expire) {}
}
```

> *If you try to define and call your tasks in the same file, Nanolith will yell at you with an error. So stick to the multi-file approach.*

That’s it

No more boilerplate is needed. We can now import this `api` variable and use it to call our `work()` function to run as a task on a separate thread (not the main thread). Let’s modify our JavaScript logic to somewhat match that of the Golang example shown before:

```
/* File: index.js */
// Import the definitions we created before.
import { api } from './worker.js';

console.time('Workflow');

// Create an array to store promises generated
// by each multithreaded operation.
const promises = [];

for (let i = 0; i <= 10; i++) {
    // Use the "api" as a function to call the
    // task as a multithreaded operation.
    const promise = api({ name: 'work' });
    // Add the task's promise to the array.
    promises.push(promise);
}

// Wait for all of the operations to complete.
await Promise.all(promises);
```

```
console.timeEnd('Workflow');
```

Output:

```
$ yarn start

  Workflow: 2.409s
```

Based on the output alone, it's clear that we've:

1. Improved the performance of our application almost ten fold. This is because all of the `work()` calls are being executed in parallel.

2. Kept our main thread free of any blocking.

To observe the non-blocking nature of our workflow using Nanolith, we can run this piece of code:

```
import { api } from './worker.js';

async function doWork() {
    const promises = [];

    for (let i = 0; i <= 10; i++) {
        const promise = api({ name: 'work' });
        promises.push(promise);
    }

    await Promise.all(promises);
    console.log('Workflow finished.');
}

doWork();
console.log('Hello world!');
```

Output:

```
$ yarn test-blocking

  Hello world!
  Workflow finished.
```

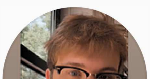
Even though the `doWork()` function takes around 2.5 seconds to run, other operations are still able to occur while it runs.

Wrap up

The purpose of this article isn't to hate on JavaScript's lacking in performance, and it definitely isn't to compare JS to Golang. Rather, the key takeaway after reading this article should be the following:

> Multithreading is a fantastic way to scale any application. Whenever you want to use multithreading in Node.js, look no further than the Nanolith NPM package.

Nodejs Typescript JavaScript Multithreading Performance



50 Followers

Full-Stack Developer | https://github.com/mstephen19 | https://www.linkedin.com/in/mstephen19/

More from Matt Stephens

Matt Stephens

Why is ES6+ so awesome?!

Javascript is a fantastic language, and is only becoming more awesome and advanced as time goes on. There's no denying that ES6 brought...

7 min read · Nov 6, 2021

8

Matt Stephens

Flickity, Parsing queryLinks, and Teamwork

Recently, I completed a project called myCharitySearch with two other developers. This was an absolutely amazing experience, and I can...

3 min read · Oct 8, 2021

2

See all from Matt Stephens

Recommended from Medium

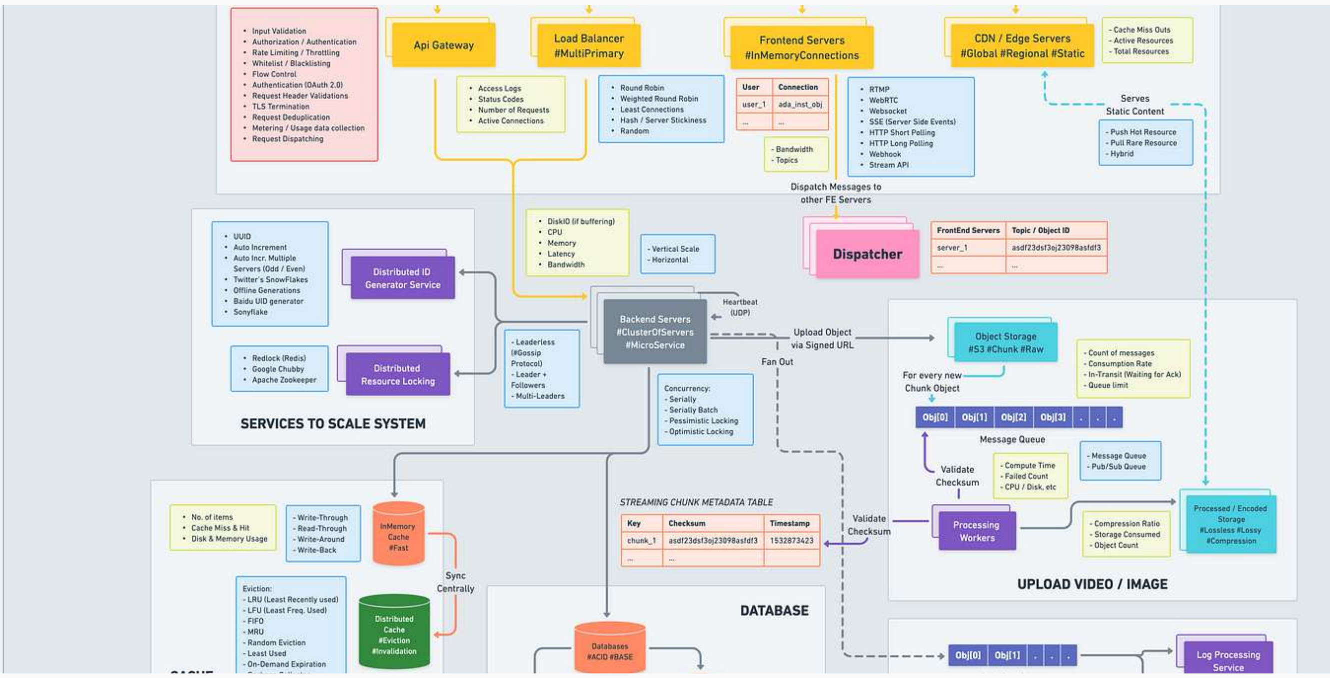
Ibrahim Ahmed in Bootcamp

How I Optimized An API Endpoint To Make It 10x Faster

When it comes to building web applications, performance is one of the most important factors to consider. If your application is slow...

· 3 min read · Jan 11

289 7



Love Sharma in Dev Genius

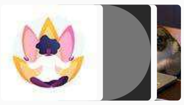
System Design Blueprint: The Ultimate Guide

Developing a robust, scalable, and efficient system can be daunting. However, understanding the key concepts and components can make the...

· 9 min read · Apr 20

1K 10

Lists



Stories to Help You Grow as a Software Developer

19 stories · 48 saves

A Guide to Choosing, Planning, and Achieving Personal Goals

13 stories · 75 saves

Staff Picks

311 stories · 79 saves

Vitalii Shevchuk in ITNEXT

🔥 Mastering TypeScript: 20 Best Practices for Improved Code Quality

Achieve Typescript mastery with a 20-steps guide, that takes you from Padawan to Obi-Wan.

· 14 min read · Jan 20

984 21

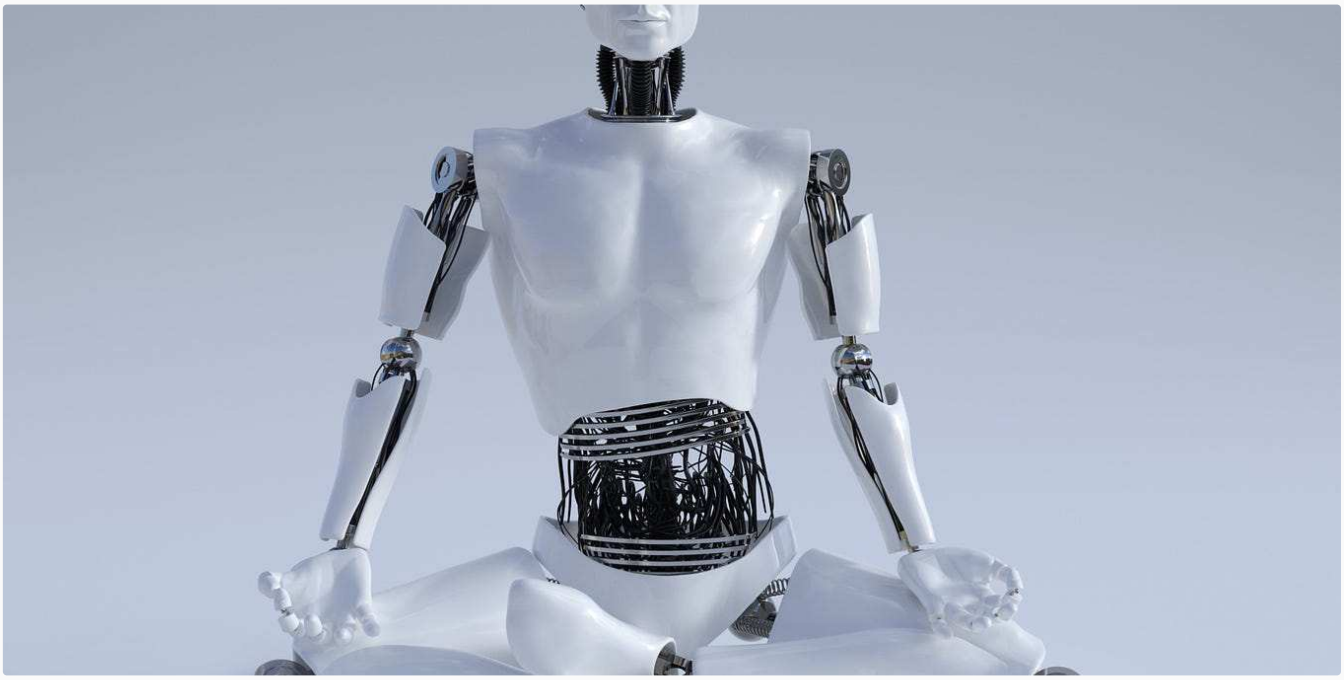
Melih Yumak in JavaScript in Plain English

Nodejs Developer Roadmap 2023

Explore nodejs developer roadmap for 2023. A step-by-step guide to how to become nodejs developer, increase knowledge as nodejs developer

· 7 min read · Jan 29

663 13



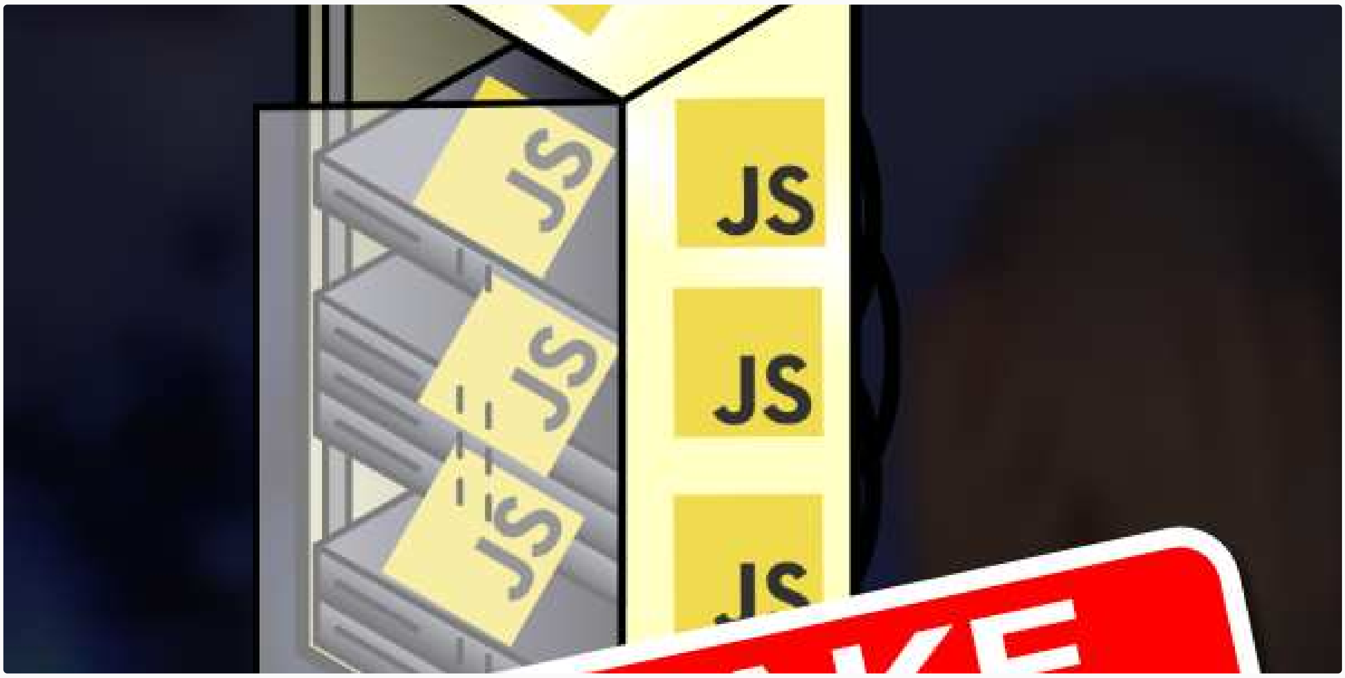
The PyCoach in Artificial Corner

You're Using ChatGPT Wrong! Here's How to Be Ahead of 99% of ChatGPT Users

Master ChatGPT by learning prompt engineering.

· 7 min read · Mar 17

20K 345



LindblomDEV

Node.js is not the optimal choice on the server

Why is JavaScript not best suited for server-side development, and how to work around these issues?

· 3 min read · Feb 14

116 6

See more recommendations