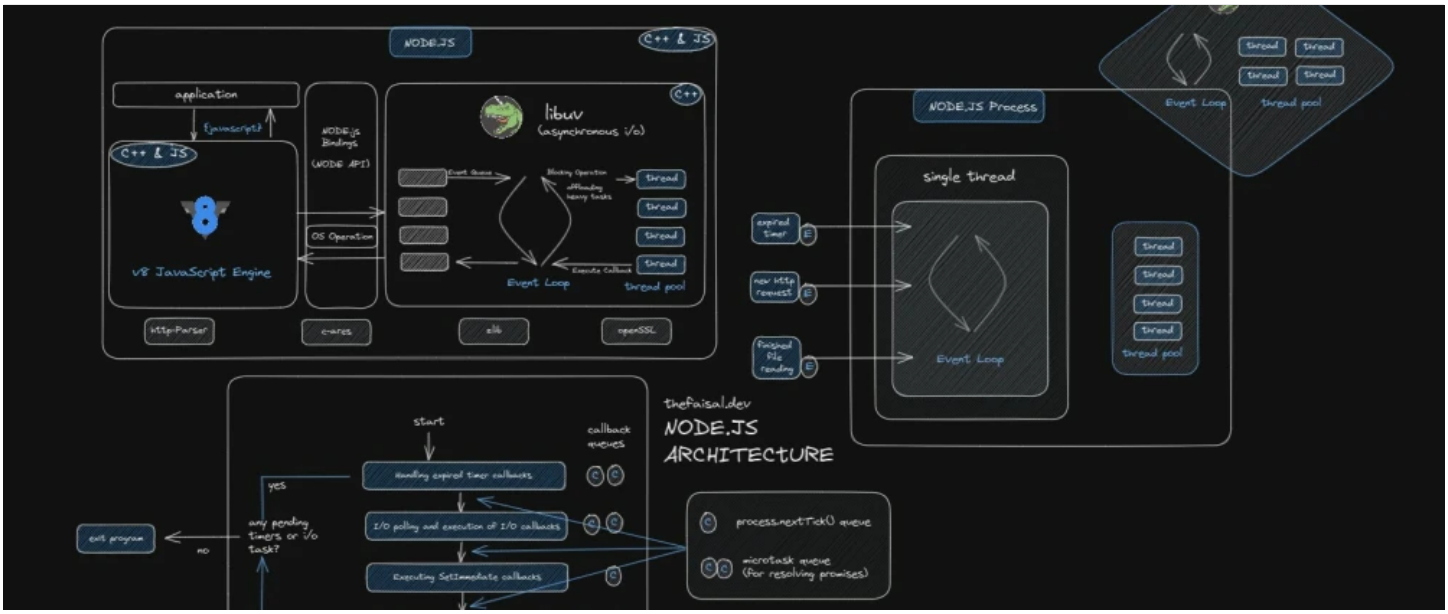


Faisal
Posted on Feb 18 • Originally published at thefaisal.dev

34 1

A Deep Dive into Node.js: Understanding its history, threading, and event-driven architecture

#webdev #javascript #node #eventdriven

What is Node.js?

According to Node's official documentation, Node.js is a JavaScript runtime built on Google's open-source V8 JavaScript engine. Node.js can execute JavaScript code on the server side, making it possible to create fast, scalable, high-performance network applications.

History of Node.js

Node.js was introduced in 2009 by **Ryan Dahl**. Despite initial skepticism, the concept gained popularity in 2011 when Dahl presented Node.js at a PHP meetup in San Fransisco on 22 Feb 2011. The video went viral, drawing attention to using JavaScript on the server side. In that video, Ryan Dahl explained Node.js in front of a bunch of programmers. He explained how he developed a new JavaScript runtime with the help of the v8 engine of Google Chrome and how we can use JavaScript on the server side. Before that, JavaScript was used only on the client and browser sides. After listening to Ryan's explanation, most programmers made fun of him because they couldn't accept

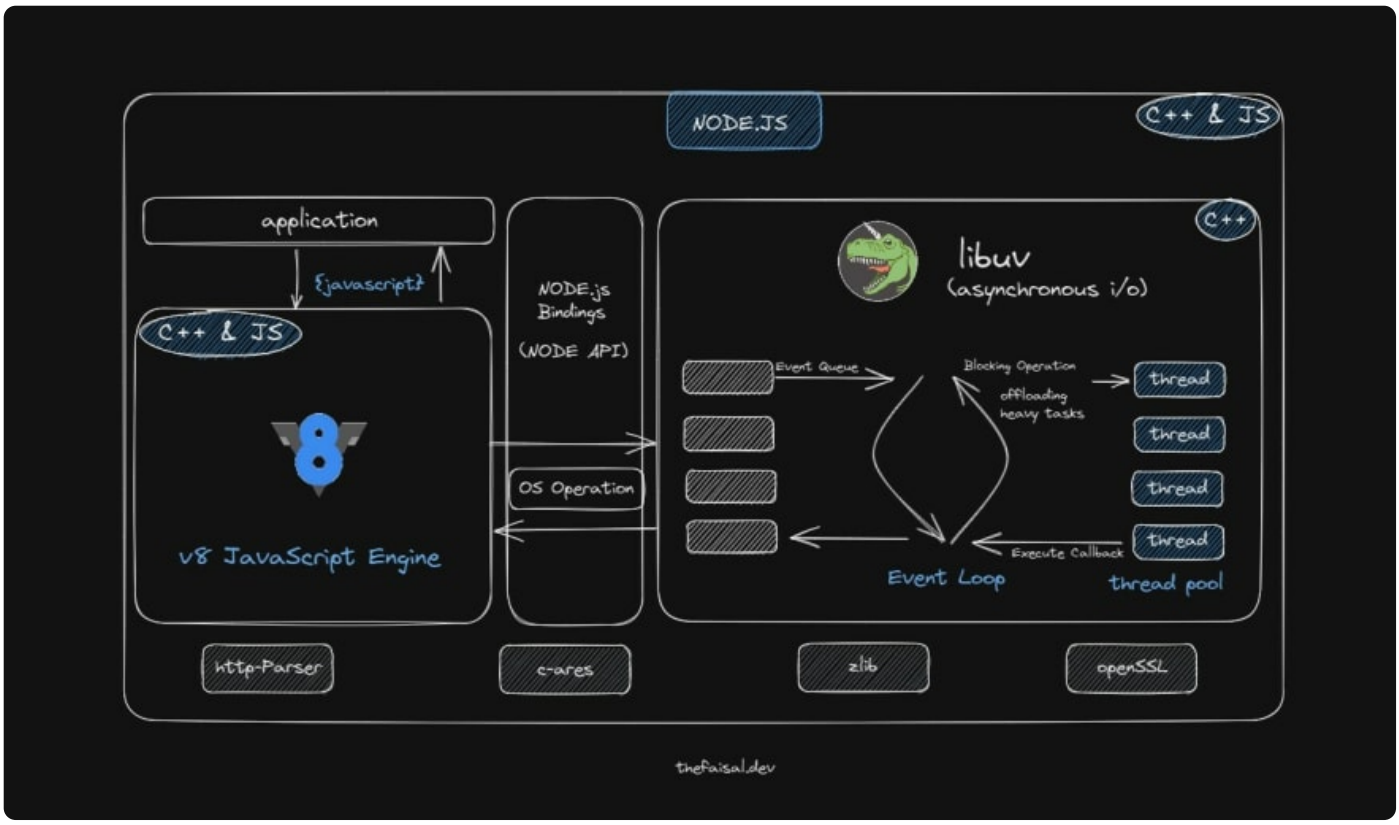
JavaScript as a server-side language. At that time, some hot server-side languages like Ruby on Rails existed. But Ryan Dahl didn't stop. Dahl saw the potential for using JavaScript on the server side and developing and improving Node.js as a solution.

Reasons Behind the Popularity of Node.js

There was some critical incident that helped the Node gain its popularity.

- In **2010**, Isaac Z. Schlueter created NPM, short for Node Package Manager. At the time, Node.js was a relatively new platform and lacked a standard package manager like those used in other programming languages. It made it easier for developers to share their code with other developers and reuse existing packages in their projects.
- In **2007**, Dwight Merriman and Eliot Horowitz created MongoDB. It uses a JSON-style binary data structure to store data which is easy to use with JavaScript. As expected, developer communities did not accept a NoSQL database like MongoDB because the SQL relational database was popular then. As social media companies like Twitter and Facebook started to publish APIs in JSON format, JSON-style data structures increased as developers found it more challenging to work with relational databases and JSON-format data. As a result, solutions like MongoDB and Node.js became more widely adopted.
- Node.js has made it easier for developers to use JavaScript in both the front-end and back-end for their web applications. This capability has been a significant factor in developers' popularity of Node.js. By using the same language on both the client and server sides, Node.js eliminates the need to switch between multiple programming languages, leading to more streamlined and efficient development. Furthermore, since JavaScript is a widely-used language, many developers already have experience with it, making the transition to using Node.js for server-side development a natural choice. These factors have contributed to Node.js becoming a go-to solution for building fast, scalable, and highly performant network applications.

The Architecture of Node.js and Understanding the V8 Engine



Let's learn about the node architectures regarding the Node's dependencies.

Node.js has several dependencies to work/ function properly. The most important ones are the v8 engine and the libuv. We saw earlier that Node.js was built on Chrome's v8 engine. First, we must know the role of the v8 engine here.

Many of us write JavaScript for the browser. When we write code, machines don't understand our code directly. The machine must convert it into machine code to understand what we command. There are engines in the browser to understand our code as machine code. The different browsers have different engines to understand/convert JavaScript code into machine code. Here are some examples:

- Google Chrome: V8
- Mozilla Firefox: Spider Monkey
- Apple Safari: JavaScriptCore
- Microsoft Edge: Chakra
- Opera: Blink (which is based on the same engine as Google Chrome, V8)

The most powerful one is Chrome's v8 engine.

On the server side, there has to be something that converts JavaScript code to machine code. For this, Ryan Dahl used a super-speed V8 engine in Node.js. Now, by this, Node.js can convert JavaScript code into machine code. That's the use case of the V8 engine in Node.

libuv Library

V8 engine is not alone enough to run a server-side framework like Node.js. Here comes another critical library called “libuv.” It’s an open-source library that strongly focuses on asynchronous IO(Input/output).

By libuv, Node.js can access the underlying computer operating system, file system, stuff related to networking, and more.

“libuv” implements **two** critical features of Node.js—

1. **Event Loop**, which handles easy tasks like a callback or network io
2. **Thread Pool**, to handle heavy work like accessing files or file compression.

I’ll discuss these two featured in detail later.

Is Node.js Entirely Written in JavaScript?

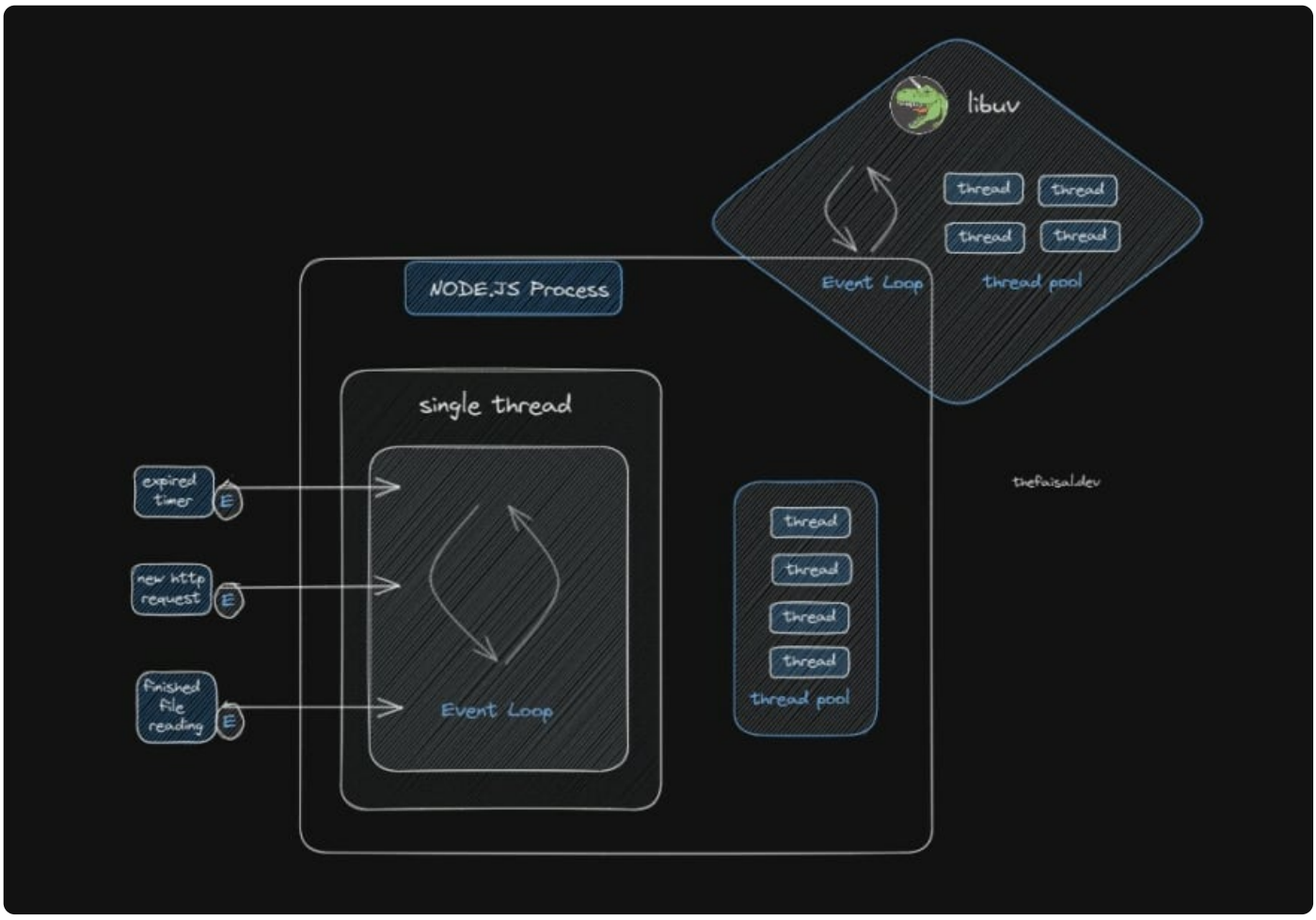
We intuitively think there is only JavaScript code in Node.js behind the scenes, as Node.js is a JavaScript runtime. But that’s not the case. As I said, libuv is an essential part of Node, and libuv is entirely written in C++. On the other hand, V8 is also written in JS & C++. So, Node is not only written in JavaScript but also C++. But there are some abstraction layers so we can access all functions and everything in the Node by pure JavaScript functions. We don’t have to mess up with C++ or any underlying code. For example- for file reading, we write a pure js function to read from the file system, which functionality is written in libuv in C++. Pretty interesting.

An important thing to note is that Node.js depends not only on the V8 engine and libuv. There are other libraries as well, like—HTTP parser (for parsing HTTP), c-ares (for handling DNS request stuff), OpenSSL (for cryptography), zlib (for file compression), etc.

Now let’s see what thread and thread pool are.

Thread & Thread Pool

A node process is initiated on our computer whenever we use Node.js. This process represents a program that is currently in execution.



Within that process, **the Node runs in a single thread**. A thread is like a mini-program within the bigger program, a sequence of instructions that can perform multiple tasks simultaneously. It's like a box where our code is executed.

It's essential to understand the concepts of **multithreading** and **single threading** clearly. Multithreading enables the execution of multiple operations simultaneously, where several tasks can run in the background at the same time. On the other hand, single threading only allows one operation to be performed at a time. In other words, it operates in a linear sequence and cannot execute multiple operations simultaneously.

It's important to understand that Node runs on a single thread, no matter we have how many users. This means that **if the single thread gets blocked, the entire application can be affected**. So, we need to be extra mindful of actions that could potentially block the thread.

Let's see what happens when the Node runs in a single thread.

When a Node application is run, the program goes through several stages. Node.js **initializes** the program, **executes all the top-level code**, **requires necessary modules**, and then **event callbacks are registered.**

Once all these steps are completed, the **event loop** starts running. Event loop where the majority of the work of the application gets done. It's the heart of the application. But, some tasks can be resource-intensive and heavy. They can potentially block the single thread if executed within the event loop. Handling these heavy tasks outside the event loop is important to avoid this.

Here, the **thread pool** comes to the rescue, which the libuv library provides. The event loop offloads those heavy tasks to the thread pool. It happens behind the scene. We don't have to worry about which tasks will go to the thread pool and which will not.

The thread pool has four additional threads separate from the single main thread. We can configure a thread pool of up to 128 threads. But, usually, we don't need to. Four threads are enough.

Here are some task examples that are offloaded to the thread pool from Event Loop -

1. **Tasks that use file system API**: When we perform operations like reading or writing files, these tasks can take a significant amount of time. They can block a single thread and cause performance issues.
2. **Cryptography-related task:** Cryptography involves complex mathematical calculations. And performing these operations in the Event Loop can slow down our application.
3. **Compression tasks:** When we need to compress large data like images or videos, these tasks can take a lot of time and can also slow down the Event Loop. Therefore, these tasks are offloaded to the thread pool to prevent this.
4. **DNS lookup:** When we need to resolve a domain name into an IP address, it's done through a DNS lookup process. This process can also take time and block the Event Loop, which may cause performance issues. Therefore, to prevent this, these tasks are offloaded to the thread pool, where they can run concurrently without blocking the main thread.

Event Loop and Its Implementation in Node.js

Now, we are in the Node.js process. This process contains a thread where the event loop runs. The Event Loop is an essential component of the Node.js process. It is responsible for executing all of the code within callback functions that make up a Node.js application.

As we learned earlier, some tasks may also be offloaded to the thread pool for efficient processing. But, the event loop is the central mechanism of the Node.js runtime. Node.js is based on a callback-driven architecture. It means that functions are called

as soon as a certain event has finished or been emitted. This happens because of the event-driven architecture of Node.js.

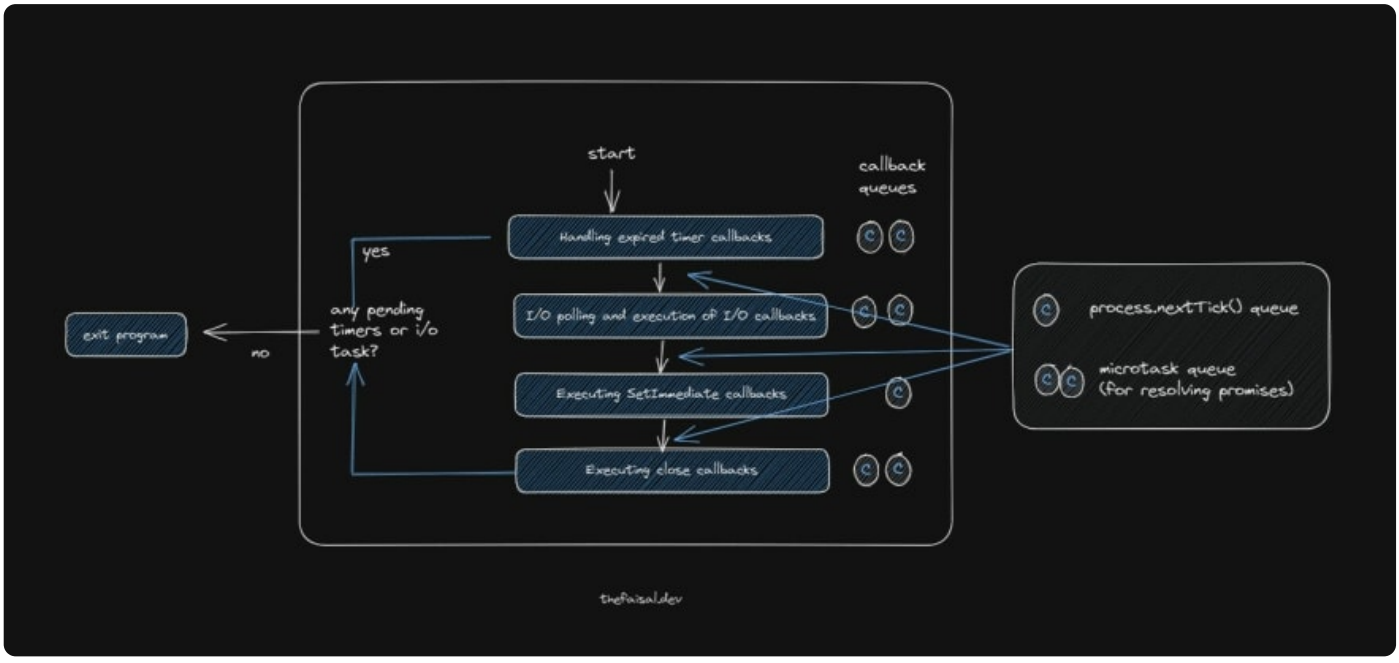
Event-Driven Architecture

Event-driven architecture is a system where events are emitted, picked up, and processed by the Event Loop. Then the Event Loop executes the associated callback functions to that event.

For example, events such as a new HTTP request, expiration of a timer, or a finished file reading or writing into a file operation will emit events, which the Event Loop will then pick up. It then processes those events by executing the associated callback functions. The Event Loop observes events and performs the necessary orchestration behind the scenes.

Let's take a closer look at how the event loop actually operates behind the scene:

The event loop starts running as soon as the Node.js application starts. The Event Loop has several phases. Each phase has its own callback queue. Some might say the Event Loop has only one callback or event queue. But it has multiple phases, each with its own callback queue.



The four most important phases of the Event Loop are as follows:

1. **Handling expired timer callbacks (setTimeout()):** The first phase involves executing callbacks from expired timers. If a timer expires during any of the other phases, its associated callback function will be executed as soon as the current phase finishes, and the event loop returns to its first phase.

2. **I/O polling and execution of I/O callbacks:** Node IO means networking or file system stuff like file reading, writing into new files, etc., and Polling means—looking for new IO events that are ready to be processed and place them in the callback queue. This phase is where 99% of the application code is executed.

3. **Executing SetImmediate callbacks:** This is a special type of timer that allows code to be executed immediately after the execution of the I/O callbacks.

4. **Executing close callbacks:** The Event Loop processes all close events in this phase. This includes events such as closing a web server or a web socket.

In addition to these four phases, two other queues are important to be aware of:

The process.nextTick() queue: This queue executes its callbacks immediately after the current phase finishes. It is similar to the SetImmediate callbacks. The only difference is that SetImmediate callbacks only execute immediately right after the execution of the I/O callbacks, but process.nextTick() executes right after any of the phases.

The microtask queue for resolving promises: This queue executes its callbacks immediately after the current phase finishes, just like the process.nextTick() queue. If there are any callbacks in either of these two queues, they will be executed immediately after the current phase is completed rather than waiting for the entire event loop to finish its four phases. For example, if a promise resolves and returns data from an API call while the callback of an expired timer is running, its callback will be executed immediately after the timer's callback has finished.

After each of the first four phases, the event loop checks if there are any callbacks in these two queues. If there are, they will be executed right away. This completes one tick of the event loop. A tick is defined as one cycle of the loop. After one cycle, the Node.js runtime checks if there are any pending timers or I/O tasks, or any callback of any phases. If there are, the loop runs again. Otherwise, the application exits.

Asynchronous programming is made possible in Node.js due to the Event Loop, which makes the Event Loop the most important feature of Node.js. It allows Node.js to run in a single thread, making it lightweight and fast but also presenting some potential risks. Since Node.js is a single-threaded program, it's possible to block our application accidentally. So, it's important to take extra care while writing code to avoid accidentally blocking the application.

Steps to Avoid Blocking the Event Loop

Here are some tips to avoid blocking the code in Node.js:

- Avoid using synchronous versions of functions; write any necessary synchronous code outside of any callback functions. This way, the code will be executed before the event loop starts.
- Avoid complex calculations such as nested loops in Node.js, as they can cause the application to become blocked.
- Be cautious when dealing with large JSON objects, as parsing or stringifying them can become time-consuming.
- Avoid using complex regular expressions, as they require a lot of processing power that can be resource-intensive and slow down the event loop. If we use a complex regular expression, it can take significant time to execute. That's why it is also advisable to break down large regular expressions into smaller, simpler ones that can be executed more efficiently. This will ensure that the event loop continues to run smoothly and that the application remains responsive.
- Do not perform any CPU-intensive tasks within the event loop. It can cause the application to become blocked. Here are some CPU-intensive tasks example-
- Heavy computation, such as mathematical calculations and scientific simulations
- Image and video processing, such as resizing, cropping, and filtering images and videos
- Cryptographic operations, such as encryption and decryption of data
- Data compression and decompression
- It's important to be careful if we want to do these tasks in Node.js. We will have to take extra steps to minimize their impact on the event loop. This can be done by manually offloading these tasks to a separate process or thread or using an external library optimized for performing these types of operations.

In addition, it's important to structure the application's code in a way that doesn't put pressure on the event loop, such as by avoiding complex algorithms, minimizing blocking operations, and using asynchronous programming techniques wherever possible. This will help to ensure that our Node.js application remains fast, efficient, and responsive even under heavy load.

That's it for today!

Congratulations on finishing this article! You now have a deeper understanding of Node.js. Whether a beginner or an experienced developer, this information is valuable in helping you build efficient, scalable, and high-performance applications.

If you enjoyed reading this, you can connect with me on Twitter or check out my other articles. Additionally, you can subscribe to my non-tech newsletter, where I share my thoughts on life, productivity, and more, as well as the best content I've come across each month.

Happy Learning!

Top comments (5)

Fadi Nouh • Feb 21

Thank you , it's interesting article, can I know what is the name of the tool where you make those diagrams?

Faisal • Feb 21

sure @fadinouh1, I use excalidraw. Let me know if you have any better suggestions to draw diagram!

Faisal • Feb 21

Thanks for the great feedback, @fadinouh1!

Zalka Ernő • Feb 21

Does Node.JS have similar like *workers* in client-side JS? (I just can't believe, that in the age of 64-core processors, kinda' single-thread is a valid design.)

Faisal • Feb 22

Yes, @ern0 , node.js has a similar worker-threads concept. It allows us to run js code in separate threads within a Node.js process. It was introduced in Node.js version 10.

Btw, web workers and worker threads are not exactly similar. They have similarities in concepts like separate threads, how they pass messages, concurrency, and asynchronous behavior. But, there are some other cases where they differ in design goals, limitations, or performance characteristics.

I'll write about these two in one of my upcoming articles. I'll explain how we can practically implement them, similarities and differences, etc.

Also, thank you so much @ern0 ,for mentioning this topic.

Code of Conduct • Report abuse

Faisal

I code, read, learn, share.

WORK
Software Engineer

JOINED
Feb 7, 2023

More from Faisal

How the Web Works: Understanding the Architecture of the Web
#webdev #programming #internet #beginners