

The Real Deal with Node.js Event Loop Misconceptions: Unmasking the Truth



Lakin Mohapatra · Follow

14 min read · Aug 17, 2023



Listen



Share

Node.js is like a busy expressway transporting thousands of requests smoothly to their destinations. The secret to this high-speed throughput is Node's asynchronous event loop architecture.

However, even experienced drivers can misjudge the lanes and junctions of this expressway. Hidden beneath its asynchronous design are some common myths and misconceptions about how the Node.js event loop truly operates under the hood. Understanding how the Node.js expressway really works will prevent you from getting stuck in deadlocks or crash your application's performance at high speeds.



Credit. — <https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick>

In this blog post, we'll shed light on **sixteen** sneaky misconceptions that many Node.js developers believe at some point. We'll use relatable analogies and real-world code examples so you can truly master the transit system that is the Node.js event loop.

Misconception 1: Node.js is 100% Asynchronous

One of the most common assumptions about Node.js is that it is completely asynchronous out of the box. However, while JavaScript execution and the core I/O primitives are async, Node.js actually provides both synchronous and asynchronous APIs.

For example, the `fs` module contains synchronous file system methods like `fs.readFileSync()`:

```
// Blocks the event loop!  
const data = fs.readFileSync('large_file.txt');
```

When these synchronous methods get called, they block the event loop thread completely while the operation executes. This prevents any other events like network I/O from being handled concurrently.

In effect, synchronous methods turn the asynchronous event loop into a blocking single-threaded model.

Similarly, CPU intensive operations can easily block the single event loop thread:

```
// Blocks event loop  
function countPrimes() {  
  let primes = 0;  
  for(let i = 2; i < 1000000; i++) {  
    if(isPrime(i)) primes++;  
  }  
  return primes;  
}
```

So while Node.js provides an asynchronous advantage, developers have to be careful to avoid blocking the event loop by:

- Using asynchronous API alternatives
- Moving synchronous work to worker threads
- Breaking intensive work into smaller asynchronous chunks
- Using async patterns like promises

By properly leveraging Node.js's asynchrony and avoiding blocking calls, you can reap the performance benefits of the event loop model.

Misconception 2: I/O like network and files is non-blocking

A common assumption is that all I/O in Node.js is non-blocking because of its asynchronous nature. However, while most I/O is handled asynchronously, operations that transfer large chunks of data can still block the event loop.

For example, reading a large file from the filesystem into memory or parsing a large JSON payload from the network can overwhelm and block the single-threaded event loop:

```
// Read 1GB file
fs.readFile('huge_file.jpeg', (err, data) => {
  // Callback after file is fully read!
  // But event loop is blocked...
});

// Receive 16MB JSON payload
http.get('http://large-json-payload', (res) => {
  let json = '';

  res.on('data', (chunk) => {
    json += chunk;
  });

  res.on('end', () => {
    const obj = JSON.parse(json); // Blocking parse!
  });
});
```

In both cases, the synchronous in-memory processing after the I/O read ends up blocking the event loop thread even though the initial file/network read was asynchronous.

The best way to prevent this is to:

- Stream process data in smaller chunks rather than reading everything into memory
- Offload CPU-intensive operations like parsing to worker threads
- Limit payload sizes

So while Node.js asynchronous I/O provides great advantages, for very large data it's important to handle I/O asynchronously and also avoid synchronous processing that blocks the loop.

Misconception 3: Timers are asynchronous

Timers like `setTimeout()` and `setInterval()` are commonly used in Node.js for scheduling future work. It's often assumed that callbacks from timers will fire predictably after the specified delay.

However, a timer is not guaranteed to fire exactly on schedule if the event loop is blocked by other synchronous or long-running async operations.

For example :

```
console.log('Start');

setTimeout(() => {
  console.log('Timeout');
}, 1000);

// Block for 1.5 seconds
while(Date.now() < Date.now() + 1500) {}

console.log('End');
```

Even though the timeout was set for 1 second, the blocking loop delays it from firing in time. The output is:

```
Start
End
```

Timeout

The timer callback had to wait until the loop was free again to execute. Blocking operations essentially pause the event loop including timer scheduling.

To prevent this, we should:

- Avoid blocking the event loop with synchronous or long running code
- Use `setImmediate()` for operations that need to run after I/O not on a schedule
- Account for delays in timer schedules under load
- Use Worker threads if accurate timers are needed under heavy load

So while timers provide asynchronous scheduling, they are still executed on the event loop and can be delayed if it is blocked.

Misconception 4: Node scales well automatically

A common assumption is that Node.js applications scale smoothly as load increases since everything runs asynchronously on the event loop. However, a single threaded event loop can still become overwhelmed under heavy load.

For example, a JSON parsing endpoint:

```
app.post('/data', (req, res) => {  
  const json = JSON.parse(req.body); // Parsing blocks!  
  // ...  
});
```

As load increases, parsing all JSON serially on the event loop can overload the single CPU core. Each request has to wait for previous parsing to finish before the callback can execute.

Some other examples where a single Node process may not scale:

- Apps with few highly loaded routes doing extensive processing
- Data processing apps like image transformation

- Apps bottlenecked on database queries or network I/O

So while Node handles I/O bound work very efficiently, CPU bound operations can overwhelm a single process.

Strategies to scale CPU intensive Node apps:

- Clustering — Spread load across multiple processes
- Load balancing — Distribute incoming requests
- Worker threads — Parallelize heavy processing
- Partitioning — Split data pipelines across services

The key is to profile bottlenecks and add concurrency across nodes, not just async I/O within a node.

Misconception 5: Using clusters magically parallelizes Node

A common technique to scale Node.js apps is to create a cluster of Node.js processes using the cluster module. This spins up multiple worker processes that all run your application code.

However, it's a mistake to assume the clustered processes automatically parallelize your application logic. Each worker process still runs on a single thread with a single event loop.

For example, if you have a CPU intensive image processing endpoint:

```
app.post('/image', (req, res) => {  
  intensiveProcess(req.body.image); // CPU bound operation  
  res.sendStatus(200);  
})
```

Clustering this route across 4 workers will simply round-robin requests between the processes. But each request is still processed synchronously on a single event loop.

We would need to parallelize the `intensiveProcess()` logic itself to benefit from multiple cores:

```
// Spread image chunks across workers
app.post('/image', (req, res) => {

  const chunks = splitImage(req.body.image);

  chunks.forEach(chunk => {
    workers[chunk.id].postMessage(chunk);
  });

  res.sendStatus(200);

});

// Worker process that uses all cores
const worker = () => {
  process.on('message', chunk => {
    intensiveProcess(chunk);
  });
}
```

Clustering helps availability and spreads network I/O across nodes, but you still need code logic to create true parallelism across multiple CPU cores.

Misconception 6: Async means parallel

In Node.js, people often assume that because code runs asynchronously, it must also run in parallel. However, this is not automatically the case.

Due to the single-threaded event loop, asynchronous code runs concurrently but not in parallel by default. Each async operation is suspended while it awaits I/O, but resumes sequentially on the event loop thread.

For example:

```
const fetch = require('node-fetch');

async function getResults() {

  const response1 = await fetch(url1); // (1)
  const data1 = response1.json();

  const response2 = await fetch(url2); // (2)
  const data2 = response2.json();
```

```
    return [data1, data2];  
  }  
}
```

While the `await` calls are asynchronous and non-blocking, they run serially on the event loop. Request 2 will not start until request 1 finishes, even though they could run in parallel.

We would need worker threads to achieve true parallelism:

```
const { Worker } = require('worker_threads');  
  
async function getResults() {  
  const worker1 = new Worker(fn1); // (1)  
  const worker2 = new Worker(fn2); // (2)  
  
  const [data1, data2] = await Promise.all([  
    worker1.promise,  
    worker2.promise  
  ]);  
  
  return [data1, data2];  
}
```

Now both heavy operations can run in parallel on separate threads.

So asynchronous \neq parallel. Async enables concurrency, but parallelism requires multiple threads.

Misconception 7: The event loop runs forever

It's commonly assumed that once a Node.js process starts, its event loop runs indefinitely to handle requests until explicitly terminated. However, there are cases where the event loop will exit even if the program is not terminating.

For example, consider this HTTP server:

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



```
http.createServer((req, res) => {  
  // handler...  
}).listen(3000);  
  
console.log('Server running...');
```

This looks like it should keep running forever. However, if there are no pending timers or I/O callbacks in the event loop queue, the loop will exit and terminate the process even though the program is still running!

Some examples of unhandled events that can cause event loop exit:

- Unhandled rejected promises
- Uncaught exceptions
- Unrefed setTimeout/setInterval callbacks
- Open connections, streams, etc.

To prevent unexpected termination:

- Make sure to handle errors and rejections
- Monitor unhandledRejection
- Ref timers to keep loop alive
- Close unused streams and sockets
- Use a process manager like PM2 in production

So while the event loop is designed to run continuously, we need to make sure to avoid ditching events that will cause it to exit unintentionally!

Misconception 8: Microtasks always run before next event loop tick

It's commonly stated that microtasks like promise callbacks are guaranteed to run before the next iteration of the event loop in Node.js. However, the microtask queue is not the only mechanism that can queue tasks at the end of the loop.

For example, `process.nextTick()` also queues tasks to run at the end of the current loop:

```
Promise.resolve().then(() => {
  console.log('Microtask');
});

process.nextTick(() => {
  console.log('NextTick');
});

// Prints:
// NextTick
// Microtask
```

Even though the promise queued a microtask, `nextTick()` took priority at the end of the current loop iteration.

So while microtasks do get priority over timers and I/O, operations like `nextTick()` can interleave with or even run before microtasks at the end of the loop.

The exact ordering depends on the current phase of the event loop. Understanding these subtleties is key to proper concurrency management in Node.js!

Misconception 9: The event loop is only used for I/O

It's a common misconception that the Node.js event loop is solely for handling async I/O operations. The truth is the event loop handles the scheduling and execution of all callbacks in general, not just I/O.

For example, when a timer finishes via `setTimeout()` or `setInterval()`, those timer callbacks are not considered I/O operations. But they are still queued in the event loop and will run when the call stack is empty:

```
setTimeout(() => {
  console.log('Timer callback!');
}, 1000);
```

The timer callback is queued in the event loop and will run on the next tick after 1 second, despite not being I/O related.

Similarly, callbacks for immediate invocation via `setImmediate()` are also executed in the event loop:

```
setImmediate(() => {  
  console.log('Immediate callback!');  
});
```

This callback is queued at the end of the current event loop tick and runs after I/O callbacks.

while async I/O is a major use case for Node's event loop, any JavaScript callback code scheduled for future execution goes through the loop including timers, immediates, and resolved promises. The event loop handles all deferred JavaScript execution regardless of I/O.

Misconception 10: Node.js is single-threaded

It's a common misconception that Node.js executes all JavaScript code on a single thread. Under the hood, Node.js actually leverages both libuv thread pools and worker threads for non-blocking I/O and parallelism.

For example, filesystem operations happen in the libuv thread pool:

```
fs.readFileSync('file.txt'); // Read from libuv thread pool
```

This allows fs I/O to happen non-blockingly without stalling the main event loop.

Node.js also provides the `worker_threads` module to create additional threads:

```
const { Worker } = require('worker_threads');  
  
const worker = new Worker(func); // Run on separate OS thread
```

Now `func()` can execute CPU intensive code in parallel.

So in summary:

1. libuv thread pools allow non-blocking I/O on the main event loop.
2. worker_threads enable parallel threading manually.

Node.js uses an asynchronous, event driven model enhanced with threads for optimal concurrency. Describing it as just “single threaded” is inaccurate and misleading. Properly leveraging its asynchronous and multi-threaded capabilities is key.

Misconception 11: Node.js can't do CPU intensive work

It's often assumed that because Node.js uses an asynchronous event loop, it is not suitable for CPU intensive operations. However, with proper design, Node.js can absolutely handle high CPU workloads using clusters and threads.

For example, a common CPU intensive task is processing and transforming images:

```
app.post('/image', (req, res) => {  
  intensiveTransform(req.body.image); //CPU intensive  
  res.send(transformedImage);  
})
```

This would block the event loop. We can parallelize across processes using clustering:

```
// Cluster module  
  
if(cluster.isMaster) {  
  // Spin up worker processes  
  for(let i = 0; i < numCPUs; i++) {  
    cluster.fork();  
  }  
} else {  
  app.post('/image', (req, res) => {  
    intensiveTransform(req.body.image);  
  })  
}
```

Now the image processing workload is distributed across multiple processes.

We could also parallelize across threads:

```
const { Worker } = require('worker_threads');

app.post('/image', (req, res) => {

  const worker = new Worker(fn, {workerData: {image}});

  worker.on('message', (transformedImage) => {
    res.send(transformedImage);
  });

})
```

So while extra design is required, Node.js can absolutely handle CPU intensive workloads through clustering and threads!

Misconception 12: Non-blocking means no delays

It's easy to assume that since Node.js uses non-blocking I/O all operations will be fast with minimal delays. However, even though asynchronous operations don't block the event loop, callbacks still execute in a sequential order which can impact throughput.

For example, making two asynchronous file reads:

```
fs.readFile('file1.txt', () => {
  // Callback 1
  fs.readFile('file2.txt', () => {
    // Callback 2
  });
});
```

While the fs module uses non-blocking I/O, callback 2 cannot start until callback 1 finishes since they execute sequentially.

If file1 takes 5 seconds to read, file2 will wait 5 seconds before starting, even though it could technically start right away since it's non-blocking.

This can be a bottleneck for throughput. Solutions include:

- Splitting operations into parallel threads using `worker_threads`
- Processing data in chunks rather than all in memory
- Using async patterns like `Promise.all` to kick off parallel async work
- Queueing multiple async DB operations rather than waiting for each callback

Non-blocking I/O does not automatically mean faster or parallel execution. Careful design is required to maximize throughput across asynchronous workflows in Node.js. Async operations don't block the event loop but callbacks still execute in order which can cause throughput delays.

Misconception 13: Top-level await stops the event loop

It's easy to assume that using `await` at the top level of a Node.js module would stop the event loop by making the code synchronous. However, top-level `await` only blocks the containing async function, not the entire event loop.

For example:

```
// app.js
import fetchUser from './fetchUser.js';

const user = await fetchUser(userId); // top-level await
```

```
// fetchUser.js
export default async function fetchUser(id) {
  return await db.query(`SELECT * FROM users WHERE id = ${id}`);
}
```

The top-level `await` in `app.js` blocks only inside the shorthand async function it is contained in. The event loop can still handle other events like incoming requests concurrently.

The `await` inside `fetchUser()` blocks only `fetchUser()`, not the entire app.

Await always blocks only the containing async function scope, not the entire process. Top-level await provides ergonomics of synchronous code without actually blocking the event loop. Other events will still run concurrently.

Misconception 14: Garbage collection doesn't impact performance

It's easy to think garbage collection (GC) is “free” in Node.js and has no performance impact since it runs asynchronously. However, if large memory allocations occur frequently, GC pauses can degrade performance.

For example, this would trigger frequent large GC cycles:

```
let cache = {};  
  
// Cache grows unbounded  
setInterval(() => {  
  cache[getRandomKey()] = hugeObject;  
}, 100);
```

Each iteration can allocate large objects, quickly filling up memory. Node.js would then need to run GC frequently, pausing script execution.

These pauses can become noticeable in production under load:

- Requests during GC pauses queue, increasing latency.
- Can't take full advantage of all CPU cores if one core is paused for GC.
- GC consumes CPU cycles that could otherwise process requests.

Solutions include:

- Allocating fewer large objects
- Reusing buffers instead of new allocations
- Using proper data structures to avoid unbounded cache growth

While GC enables automatic memory management in Node.js, badly written code can lead to GC impacts, unlike other compiled languages. Carefully managing memory usage avoids degradation.

Misconception 15: Event loop lag isn't a concern

It's easy to assume that since Node.js is asynchronous, event loop lag isn't an issue. But a slow event loop can still cause increased delays between events.

For example, CPU intensive work could execute on each request:

```
app.get('/report', (req, res) => {  
  
  // CPU intensive calculation  
  const data = veryComplexCalculation();  
  
  res.send(data);  
  
});
```

As load increases, the event loop takes longer to process each request callback. This introduces lag between the start and end of each tick.

This can manifest as:

- Degraded request throughput and higher response times
- Timeouts or delays between incoming requests
- Timers and I/O events firing later than scheduled

Solutions include:

- Move synchronous work like complex calculations into worker threads
- Batch/window/throttle requests to smooth the load
- Look for and optimize hot code paths executing frequently
- Monitor event loop delay as a key metric

While asynchronous I/O avoids blocking, careful optimization is still required to keep the event loop responsive. Event loop lag indicates contention that should be addressed.

Misconception 16: Worker threads parallelize everything

It's easy to assume that since Node.js provides worker threads, they automatically parallelize all your code. However, worker threads only provide the environment for parallel execution — the application logic still needs to be properly partitioned.

For example, just offloading code to a worker thread won't parallelize it:

```
// Won't help performance!

function processImage(image) {

  return new Worker(function() {
    // Still runs on 1 CPU
    intensiveTransform(image);
  });
}
```

We need to split the problem across multiple threads:

```
function processImage(image) {

  const threads = [];
  const chunks = splitImageIntoChunks(image);

  chunks.forEach(chunk => {
    // Spread chunk across many threads
    threads.push(new Worker(chunk));
  });

  // Aggregate results
  return mergeThreadResults(threads);
}
```

Now the intensive work can utilize multiple CPUs in parallel.

While worker threads provide the ability to run code in parallel, developers must properly divide up the work across threads to achieve full performance gains.

The asynchronous event-driven architecture makes Node.js uniquely fast and efficient. However, as we've explored, there are several nuances and misconceptions that can trip up even seasoned Node.js developers.

To recap, always remember that:

- Synchronous APIs and JavaScript code can block the event loop — use asynchronous alternatives where possible.
- Very large I/O operations may still block or overwhelm the loop despite callbacks. Handle I/O streams asynchronously.
- Timers are not guaranteed to be precise under heavy event loop load. Account for delays or use `setImmediate` for deferring work.
- A single process can still get overloaded on CPU-intensive work. Leverage clustering and worker threads.
- Async operations do not imply parallelism. Use worker threads if parallel performance is needed.
- The event loop will exit without pending activity. Make sure to handle errors/rejections and cleanup unused resources.
- Microtasks don't always run before the next tick due to `nextTick()` and `setImmediate()`.
- “Node.js can't do CPU intensive work” — It can by leveraging clusters and threads, but requires extra design consideration.
- “Non-blocking means no delays” — Async operations don't block the event loop but callbacks still execute in order which can cause throughput delays.
- “Top-level `await` stops the event loop” — `await` only blocks the wrapping async function, not the entire event loop.
- “Garbage collection doesn't impact performance” — Frequent GC pauses can degrade performance if large objects are allocated.
- “Event loop lag isn't a concern” — A slow event loop can cause delays between events like incoming requests.

- “Worker threads parallelize everything” — Threads only run in parallel, the application logic still needs to be properly partitioned.

By deeply understanding how the Node.js event loop works, you can avoid performance pitfalls and build highly concurrent asynchronous applications that leverage the full power of Node.js.

Thanks for reading my article so far.

You can connect me via [Linkedin](#) or [Twitter](#)

Sadly, Medium does not support any creator in India, if this article provided you value in some way, you can show your support to me here by clicking on the below button.



“Buy me a coffee” is a global platform where millions of people support creators and artists, financially.

References :

Event loop in JavaScript

There are only two kinds of languages: the ones people complain about and the ones nobody uses. - Bjarne Stroustrup...

acemood.github.io

How does nodejs event loop keep running without a for/while forever loop?

I read Nodejs Event Loop and "Event Loop Explained" and Don't Block the Event Loop I don't think there is a for/while...

stackoverflow.com

The Node.js Event Loop, Timers, and process.nextTick() | Node.js

Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine.

nodejs.org

Nodejs

JavaScript

React

Developer

Software Development

Open in app ↗

Sign up

Sign in



Search



Follow



Written by Lakin Mohapatra

229 Followers

Software Engineer | Hungry coder | Proud Indian | Cyber Security Researcher | Blogger | Architect (web2 + web 3)

More from Lakin Mohapatra



Lakin Mohapatra

Kill process running on specific ports in Ubuntu and windows | Fix error “Address already in use”

Developers often face errors related to ports that are already in use on their local machines. A common error is “Address already in use...


2 min read · Nov 9, 2022

 80







 Lakin Mohapatra

Web Transport: The Future of Real-Time Communication, Bridging the Gap beyond WebRTC

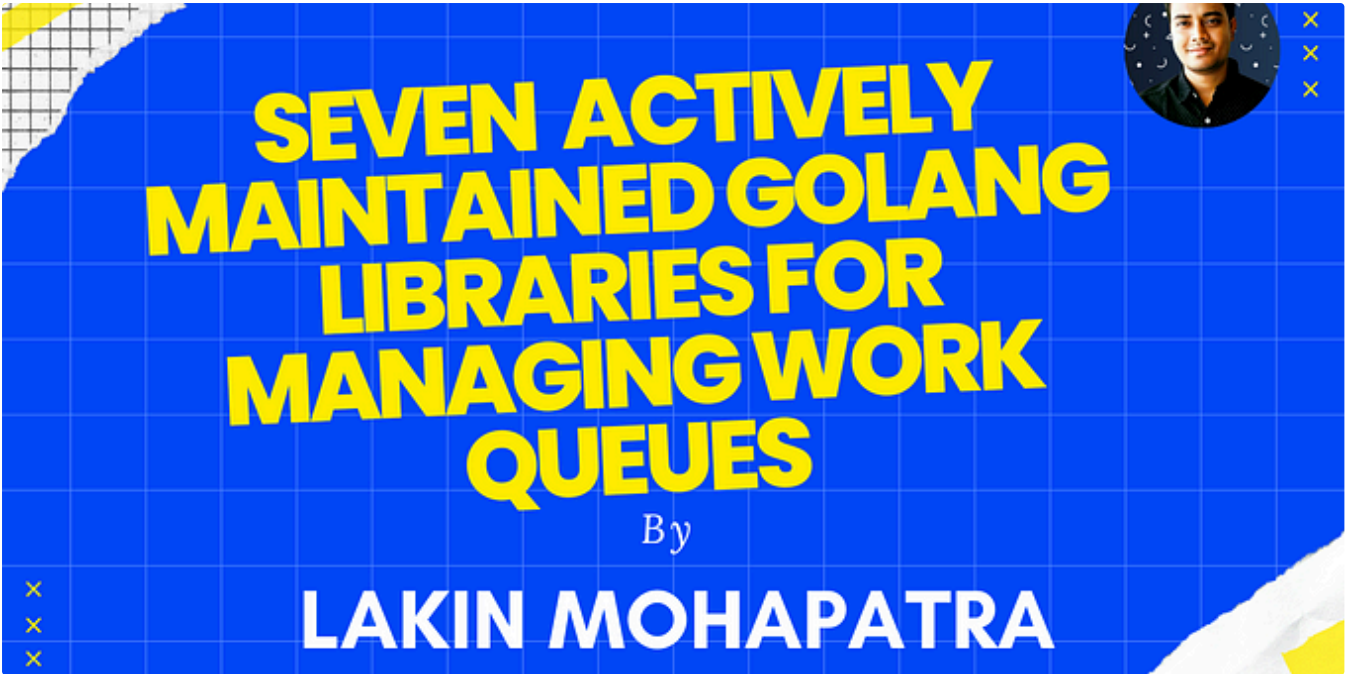
Tired of slow file transfers and bandwidth bottlenecks? Discover the...


10 min read · Jun 9, 2023

 130

 2





 Lakin Mohapatra

7 Actively Maintained Golang Libraries For Managing Work Queues

Dear golang developers, are you looking for actively maintained Golang libraries for managing work queues?

2 min read · Nov 11, 2022

 149





89.248.169.130	/htmlrpc.php
89.248.169.130	/htmlrpc.php
89.248.169.130	/htmlrpc.php
220.181.108.168	/xe4k09v3t93nca.php?tozzwudzf=car-accident-on-dixie-highway-michigan
106.11.156.176	/xe4k09v3t93nca.php?tozzwudzf=socom-bamel-profile
106.11.153.165	/xe4k09v3t93nca.php?tozzwudzf=denso-safety
106.11.159.165	/xe4k09v3t93nca.php?tozzwudzf=denso-safety
66.249.69.171	/xe4k09v3t93nca.php?tozzwudzf=coyote-usa
66.249.69.171	/xe4k09v3t93nca.php?tozzwudzf=coyote-usa
66.249.69.77	/xe4k09v3t93nca.php?tozzwudzf=andromax-b
69.30.198.186	/xe4k09v3t93nca.php?tozzwudzf=%C3%A0%C2%A4%E2%80%94%C3%A0%C2%A4%C2%BE%C3%A0%C2%A4%C2%AF%C3%A0%C2%A4%C2%A4%C3%A0%C2%A5%C2%8D%C3%A0%C2%A4%C2%B0-%C3%A0%C2%A4%
217.182.253.12	/xe4k09v3t93nca.php?tozzwudzf=%C3%A5%C2%9D%E2%80%9A%C3%A8%CB%86%C2%B8-%C3%A9%C2%A8%CB%9C%C3%A8%C2%A0%C2%A1-%C3%A5%C2%AD%C2%A8%C3%A8%C2%B2%C2%B8
217.182.253.12	/xe4k09v3t93nca.php?tozzwudzf=kurent-%C3%83%C2%BCoungsb%C3%83%C2%A4ter
66.249.69.173	/xe4k09v3t93nca.php?tozzwudzf=%E0%A4%B0%E0%A4%BE%E0%A4%B8%E0%A5%8D%E0%A4%A4%E0%A5%87-%E0%A4%AA%E0%A4%B0-%E0%A4%B6%E0%A4%BE%E0%A4%AF%E0%A4%B0%E0%A5%80
66.249.69.75	/xe4k09v3t93nca.php?tozzwudzf=%E0%A4%B0%E0%A4%BE%E0%A4%B8%E0%A5%8D%E0%A4%A4%E0%A5%87-%E0%A4%AA%E0%A4%B0-%E0%A4%B6%E0%A4%BE%E0%A4%AF%E0%A4%B0%E0%A5%80
69.30.198.186	/xe4k09v3t93nca.php?tozzwudzf=%C3%A0%C2%A4%C2%B0%C3%A0%C2%A4%C2%BE%C3%A0%C2%A4%C2%B8%C3%A0%C2%A5%C2%8D%C3%A0%C2%A4%C2%A4%C3%A0%C2%A5%E2%80%A1-%C3%A0%C2%A4%
46.229.168.143	/xe4k09v3t93nca.php?tozzwudzf
93.190.138.201	/wp-main.php

 Lakin Mohapatra

Prevent WordPress malware from infecting the server

It is known that WordPress is used for creating various types of websites, including blogs, static sites, e-commerce, etc. Almost 42% of...

12 min read · Sep 30, 2018



441



7



See all from Lakin Mohapatra

Recommended from Medium



Manik Mudholkar

Worker Threads : Multitasking in NodeJS

Deep Dive into Worker threads

16 min read · Jan 2, 2024




263



1



 Thinh Dang

Diving into the Node.js Event Loop

Explore Node.js event loop: async nature, phases, libuv, queues, JS context, timers, pitfalls, best practices.

31 min read · Feb 18, 2024



59



3



Lists



General Coding Knowledge

20 stories · 1266 saves



Stories to Help You Grow as a Software Developer

19 stories · 1096 saves



Coding & Development

11 stories · 637 saves



Good Product Thinking

11 stories · 586 saves

Multi-threading in



(using **worker threads**)



M /subhanusroy

in /subhanusroy



Subhanu in Engineering at Bajaj Health

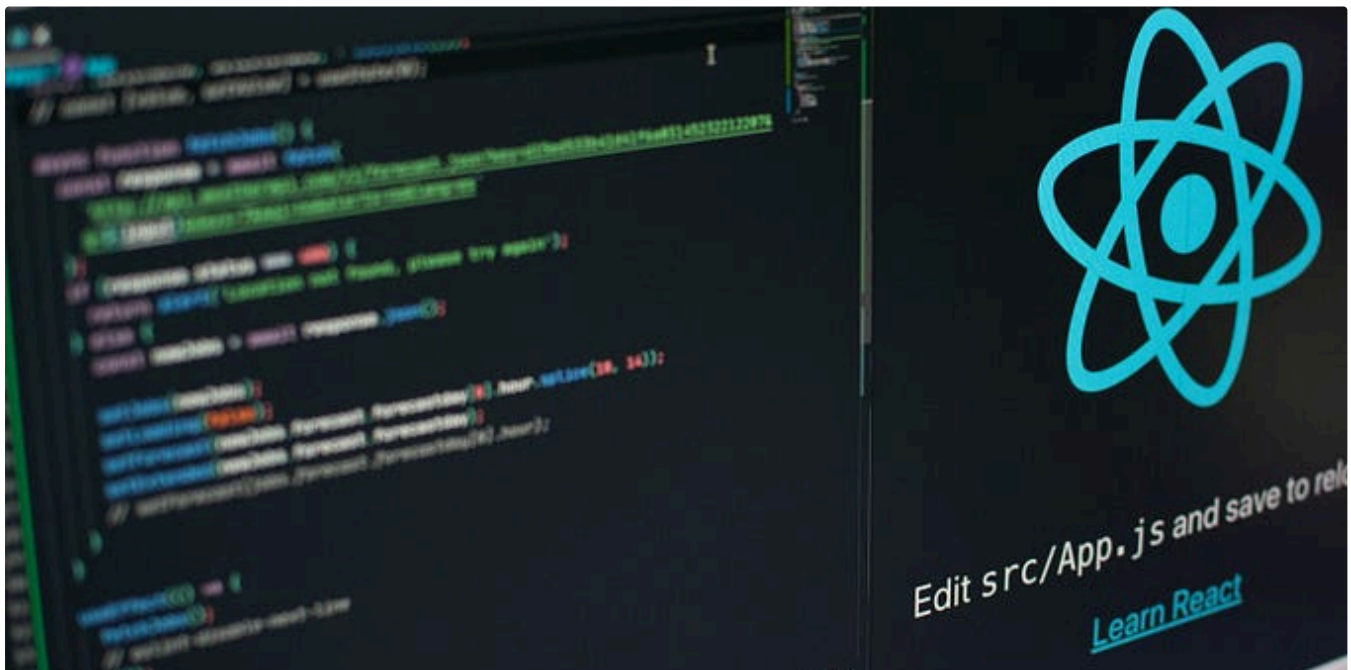
Multithreading in NodeJS—using worker threads

Whether you are

7 min read · Feb 21, 2024



226



Omer Ergun in Picus Security Engineering

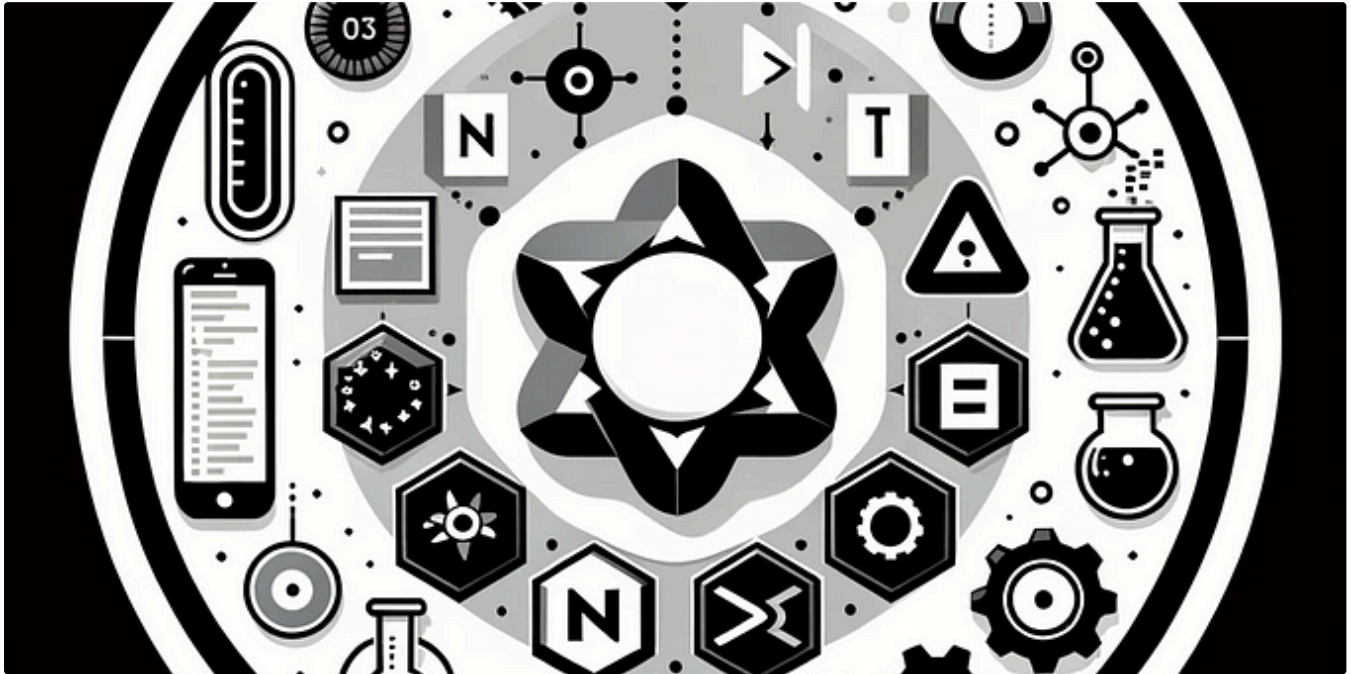
React Performance Optimization and Bundling

Code splitting gives an opportunity for “lazy loading” code blocks on a different files which means the code blocks that is required...

6 min read · Dec 7, 2023



150



Yasir Hamm

How to Set up a Production-Ready Project with Node and TypeScript

Hi, my name is Yasir, and I'm a full-stack developer with a strong focus on backend technologies. Currently, I primarily work with Asp.NET...

16 min read · Jan 15, 2024

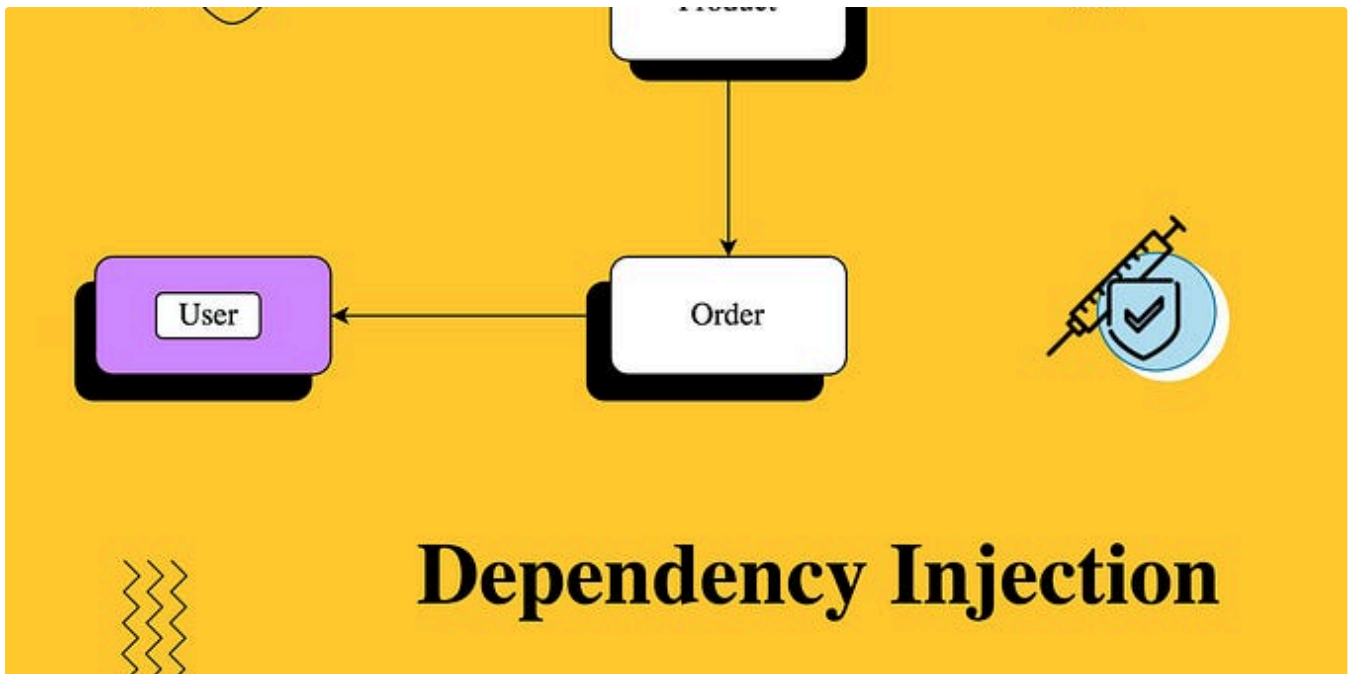


101



4





 Vahid Najafi

Implement a Dependency Injection Container in TypeScript from Scratch

IoC (also known as dependency injection container) from theory to implementation

4 min read · Mar 7, 2024



125



1



See more recommendations