INDIUM

Unlocking High-Performance Potential: Exploring Concurrency Patterns in Node.js

A Whitepaper

Executive Summary

Node.js is a powerful JavaScript runtime environment widely used for building scalable and high-performance web applications. However, it is a single-threaded platform, which means it can handle only one task at a time, limiting its ability to leverage the power of modern multi-core processors. To overcome this limitation, developers can use various concurrency patterns and techniques to enable the parallel execution of multiple tasks.

Key statistics highlighting the popularity and robustness of node.js technology

- Node.js holds the first place in the most popular web frameworks and technologies in the 2022 Stack Overflow survey.
- Typical response time for identical pages at PayPal decreased by 35% when Java was replaced with Node.js, making it a robust back-end JavaScript framework capable of cutting loading times to 50-60%.
- Node.js is mostly used for creating web apps, according to 85% of developers, and 43% of Node.js developers use it to create enterprise applications.
- Node.js is reportedly used by 1.4% to 2.2% of all websites worldwide, indicating that around 30 million websites use it.
- Node.js has over 93k+ stars on GitHub, 25.2k+ forks, and 3182+ contributors as of January 2023.
- Svelte has recorded 390-400k weekly downloads on NPM as of January 2023.

This whitepaper explores some of Node.js's most useful concurrency patterns, such as the JavaScript Event Loop, Callbacks, Async-Await, and Reactive Programming with RxJS. Developers can fully utilise the capabilities of Node.js and create high-performance applications that can easily scale and handle heavy workloads by understanding these patterns and techniques.

How should concurrency be handled if Node.js is single threaded?

Concurrency is the ability of a programme to perform multiple tasks simultaneously. Node.js is excellent at managing multiple asynchronous I/O operations, as you may have already heard. But wait, how exactly does Node.js handle this? It's single-threaded, isn't it? What about I/O-less operations?

Built on Chrome's V8 Engine, Node.js is an open-source, cross-platform runtime environment. There are two main reasons why/how node.js can handle various concurrent requests easily and why/how it becomes a noticeable choice for the development of highly scalable and server-side applications.

Node.js uses a *single-threaded event loop architecture* and it works *asynchronously.*

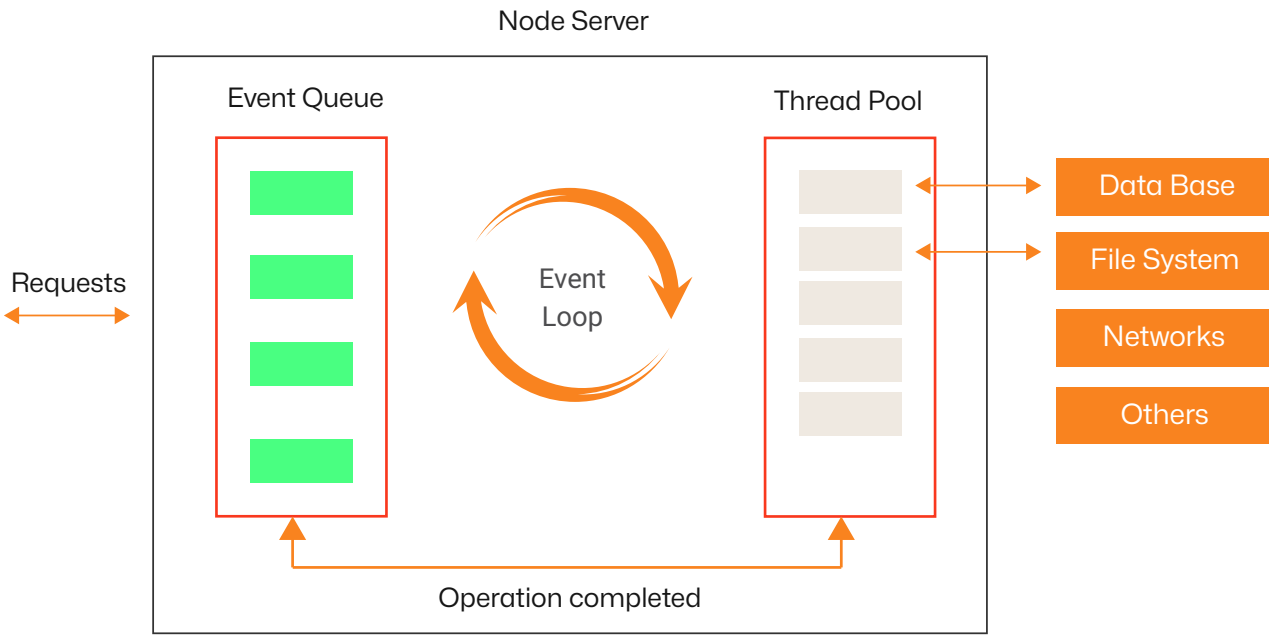
But... how?

As you can see, asynchronous I/O operations are handled by Node well. When one is aware that it only requires one thread, it might not be entirely unexpected. A thread represents one operation at a time, right? Yes and no, I suppose.

Another question that arises from this is, "How does Node know when it's time to handle multi-threaded operations?". You'll probably concur that it's not impressive. So, let's introduce *the Event Loop.,* who will assist Node in managing this mess.

JavaScript Event Loop

Let's briefly explain what the Event Loop is and how it works. Previously, I've hinted that you need a kind of "manager" to be able to handle asynchronous operations. Let's take a deep dive into the Event Loop.



The veneer between requests and the event loop is called **"Event Queue".** The event queue stores the incoming requests in the order that it received them (in queue). The Event loop selects a request from the queue, sends it to the internal C++ threads for processing, makes itself available for subsequent requests, and then begins processing those as well.

The event loop is the skeleton in the closets that allows JavaScript to appear to be multithreaded even though it is only single-threaded. The responses to the tasks that were earlier sent to the internal C++ threads for processing are then sent to the client using the JavaScript **concept of callback** functions.

Example:

```
function sample() {
console.log("sample number 1");
setTimeout(function () {
console.log("sample number 2");
}, 1500);
console.log("sample number 3");
}
console.log("sample number 4");
sample();
```

Output

```
sample number 4
sample number 1
sample number 3
sample number 2
```

Given that it is lined up even before the function call in this case, "sample number 4" will be executed first. The function will then be invoked. The application will then print "sample number 1" and enter timeout for 1.5 seconds (1500 ms).

The application will no longer stop processing new requests after 1.5 seconds; instead, it will handle the subsequent request and print "sample number 3." The lines will now be executed after the timeout expires, and "sample number 2" will be printed to the console.

Answering subsequent requests ensures that the event loop is never occupied or blocked by one. As a result, Node.js can handle multiple user requests concurrently better than traditional web servers. **that is, Concurrency.**

Asynchronous JavaScript allows for the execution of multiple tasks simultaneously, which results in the async **callback.**

Callback Explained

When calling a function that begins running code in the background, callback functions are passed as arguments. The callback function is invoked by the background code when it is finished to notify you that the task has been finished. The purpose of a callback is to run code in response to an event.

You can programme your application to "execute a piece of code every time

Example: To execute this code every time the user needs to click a key on the keyboard

```
const button =
document.getElementById('button');
function callback(){
console.log("I am a Button");
}


button.addEventListener('click', callback);
```

Explanation:

In the above code, we could see addEventListener as a function and we are passing callback as an argument. And when the button is clicked (an event is triggered) the addEventListener registers as the callback function.

Functions that use callbacks take some time to produce a result rather than returning something right away like most functions do.

The term "asynchronous," also known as "async," simply means "takes some time" or "be it in the future, not now." Callbacks are typically only used for I/O tasks, such as downloading, reading files, interacting with databases, etc.

Here, the word mentioned appears only once, just like in the example above. But in real-time application, we fail to anticipate this. Instead, a callback hell situation arises when several asynchronous functions are chained together.

How can we fix a callback hell situation?

Finite and nested callbacks can be handled in a variety of ways. It may involve the async await framework, the promise-based approach from the past, the division of the code into separate functions, the use of generators, or the RxJS library.

Using Promise

For each callback, we create a new promise, converting them to promises. If the callback is successful, we could fulfil the promise; if it is unsuccessful, we would reject the promise.

Example: To create a promise on receiving user data

```
function getPromise {
  const newUser = getUser(user);
  return new Promise((resolve, reject) => {
    if (user)
      resolve(user)
    } else {
      reject(new Error('We don't have a new user!'))
    }
  })
}
```

You can now create a new function and use it as a callback for the function.

Using Async-Await

The use of await allows us to write asynchronous functions as though they were synchronous and executed sequentially because it halts execution until the promise is fulfilled, or until the execution of the function is successful.

Example: To fetch and update the user profile

```
const userProfile = async () => {

// argument indicated number of users to fetch
  const user = await fetchUsers(1)
  const updatedAddress = await updateAddress(user);
  const phonenumber = await getPhonenumber ();
  const updateUser = await updateUser(user, updatedAdress,
phonenumber);
  return user;
}s

// fetch and update user profile
userProfile()
```

Using Generators

Let's start with the word "sequential" then. You should feel most comfortable with this section. It's yet another way of describing single-threaded behaviour and the code that ES6 generators produce that appears to be in sync.

Example: To yield an output sequentially.

```
function *main() {
    var x = yield 100;
    var y = yield 3;
    var z = yield (y * 5);
}

OUTPUT:

10
3
15
```

It is done sequentially, one at a time, to implement each of those statements. The yield keyword, which annotates where a blocking pause (blocking only in the sense of the generator code itself, not the surrounding programme!), may occur, has no effect on the top-down handling of the code inside *main (). It must be easy to comprehend, I suppose.

Using RxJS/ Reactive Programming

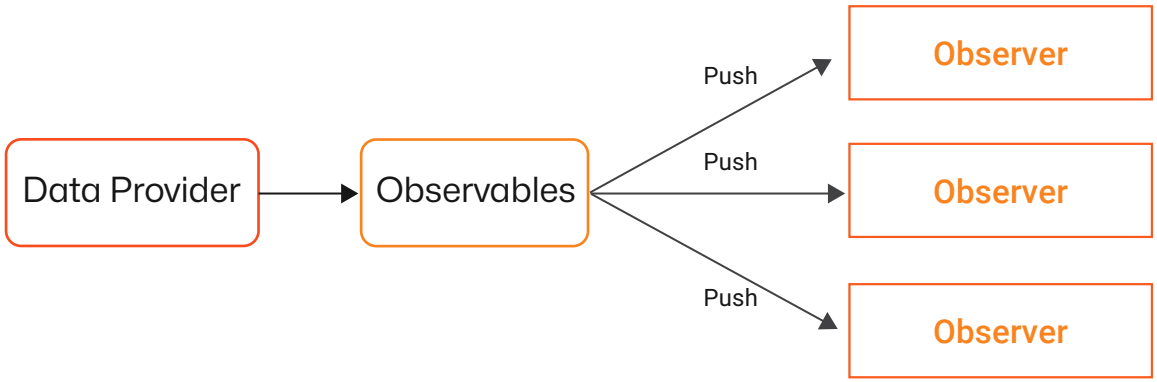
The goal of reactive programming is to develop applications that are responsive and event-driven, and which push an observable event stream to subscribers so that they can watch for and respond to the events. RxJS is a library that allows users to create event-based and asynchronous programmes using observables and operators.

Async data streams can be staged using Observables, queried using Operators, and their concurrency can be adjusted using Schedulers. Simply put, Rx = Observables + Operators + Schedulers.

Getting familiar with RxJS terminology

We need to observe data, so there must be a data producer. This producer could be a server sending data over HTTP or a user-entry field. An observable is a client-side function or object that receives data from the producer and pushes it to the subscriber (s).

An observer is a thing or a function that manages the information that the observable pushes. I think the illustration below would explain much better.



Players of RxJS

- Observable – data stream pushes data over time
- Observer – consumer of a stream of observable data
- Subscriber – connect between observer and observable
- Operator – function for the proceeding data transformation

Let's create an observable that will emit 10, 20, and 30 and subscribe to this observable:

Example: To subscribe to an observable Event.

```
Rx.Observable.of(10,20,30)
    . subscribe(
        value => console.log(value),
        err => console.error(err),
        () => console.log ("End of streaming")
);

OUTPUT:
10
20
30
End of streaming.
```

Conclusion

As Node.js continues gaining popularity as a platform for building high-performance applications, developers must understand and utilize various concurrency patterns and techniques to unlock their full potential.

In this whitepaper, we have explored some of the most effective concurrency patterns in Node.js, including the JavaScript Event Loop, Callbacks, Async-Await, and Reactive Programming with RxJS.

Looking forward, the future of Node.js development lies in utilizing these concurrency patterns and techniques to enable efficient parallel execution of multiple tasks, leveraging the power of modern hardware capabilities.

Additionally, with the increasing adoption of microservices architecture and serverless computing, Node.js is poised to become even more critical in building scalable and resilient applications.

In conclusion, developers must continue to learn and evolve with the advancements in Node.js and utilize the best practices to build high-performance applications that can handle the ever-increasing demands of modern software development.

By following the roadmap of utilizing concurrency patterns and techniques, Node.js can continue to be a reliable and robust platform for building the next generation of web applications.

Author

Divya Devi Mohan

Divya is a senior developer with more than 5+ years of extensive experience in both front-end and back-end technologies. She has managed complex projects and added value to businesses by offering knowledgeable solutions and delivering outcomes that promote business growth and success. She is a highly skilled MERN stack developer. She enjoys researching and writing about the newest and most cutting-edge technologies in her spare time.

INDIUM

About Indium

Indium is an AI-driven digital engineering company that helps enterprises build, scale, and innovate with cutting-edge technology. We specialize in custom solutions, ensuring every engagement is tailored to business needs with a relentless customer-first approach. Our expertise spans Generative AI, Product Engineering, Intelligent Automation, Data & AI, Quality Engineering, and Gaming, delivering high-impact solutions that drive real business impact.

With 5,000+ associates globally, we partner with Fortune 500, Global 2000, and leading technology firms across Financial Services, Healthcare, Manufacturing, Retail, and Technology–driving impact in North America, India, the UK, Singapore, Australia, and Japan to keep businesses ahead in an AI-first world.

USA	INDIA	UK	SINGAPORE
Cupertino | Princeton Toll-free: +1-888-207-5969	Chennai | Bengaluru | Mumbai Hyderabad | Pune Toll-free: 1800-123-1191	London Ph: +44 1420 300014	Singapore Ph: +65 6812 7888

www.indium.tech

For Sales Inquiries
sales@indium.tech

For General Inquiries
info@indium.tech