# Node.js

It’s a runtime environment for executing JavaScript code built on chrome’s V8 JavaScript engine. It allows developers to execute JavaScript code on the server side, enabling full-stack JavaScript development. Node.js extends JavaScript's capabilities to the server, meaning you can run JavaScript code outside the browser, on the server.

## Features

- Asynchronous and event-driven.
- Single threaded to handle all requests.
- Non-blocking I/O
- Well suited for building real time applications.
- Support for WebSocket.
- Rich set of libraries, tools and framework

## Advantages

- Highly scalable and data intensive.
- Great for prototyping and agile environment
- Fast and efficient
- Single programming language for full stack development.
- Large ecosystem for open source libraries.

## Disadvantages

- Not suited for CPU intensive tasks
- Nested callbacks lead to callback hell.
- No built-in support for multithreading.

## Where to use

- Real time web applications
- Network applications
- Distributed systems
- SPA
- Microservices

## Npm (Node package manager)

It’s the default package manager for node.js. It helps developers manage dependencies (libraries or modules) in their Node.js projects.

## How it works

- Single-Threaded event loop
- Non blocking I/O
- Event driven Architecture

***Single-Threaded event loop:***

Node.js uses a single-threaded event loop to handle multiple requests concurrently.Node.js uses one thread to handle all requests.

***Non blocking I/O:***

It doesn’t wait for I/O operations to complete before moving on to the next task.

***Event driven architecture:***

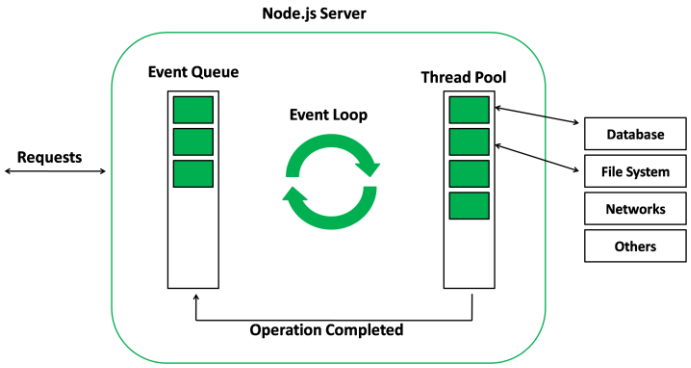
The event loop ensures that Node.js processes events asynchronously without blocking the main thread.

## Call stack :

It keeps track of function calls in a program. It operates as a Last In, First Out (LIFO) data structure, meaning that the last function called is the first one to be resolved.

## Event loop

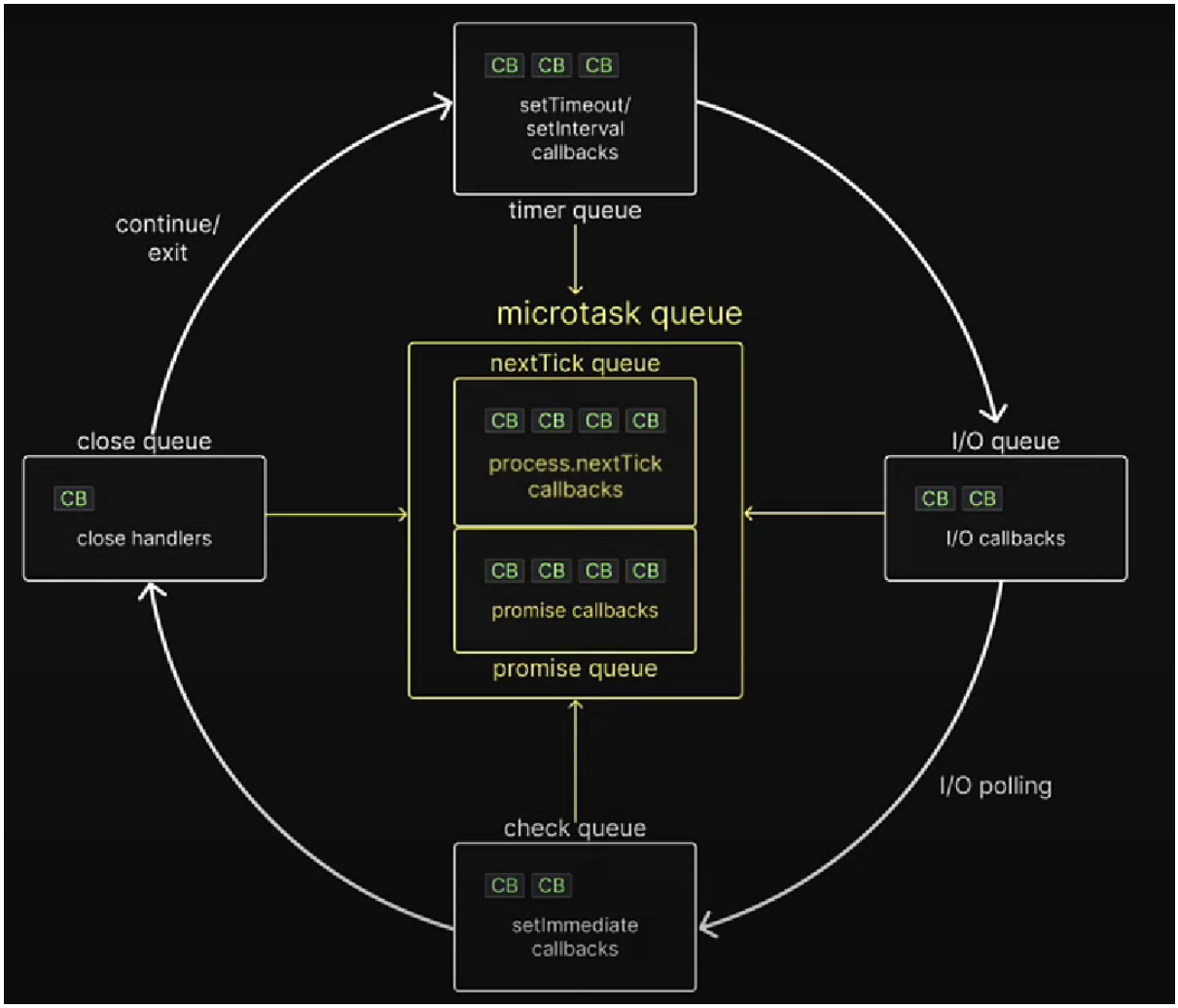
The event loop allows node to perform non-blocking I/O operations despite the fact that JavaScript is single-threaded.




When using Node.js, a special library module called libuv is used to perform async operations. This library is also used, together with the back logic of Node, to manage a special thread pool called the libuv thread pool. This thread pool is composed of four threads used to delegate operations that are too heavy for the event loop. I/O operations, Opening and closing connections, setTimeouts are examples of such operations.

When the thread pool completes a task, a callback function is called which handles the error(if any) or does some other operation. This callback function is sent to the event queue. When the call stack is empty, the event goes through the event queue and sends the callback to the call stack.

## Execution of event loop:




## *Microtask queues:*

- *nextTick queues:*
  process.nextTick(()=>{})
- *promise queue:*
  Promise.resolve().then(()=>{})

All CB IN nextTick queue are executed before CB in promise queue. process.nextTick() is a powerful tool in Node.js for prioritizing certain operations before others in the event loop.

**It gets executed before any other queues.**

## *Timer queue :*

CB in the microtask queues are executed before & in between CB in timer queue. Its getting executed in FIFO order.

- setTimeout()
- setInterval()

### *I/O Queue:*

Handling callbacks associated with I/O operations, such as reading from a file, receiving data from a network, or interacting with a database.

- fs.readFile()

**I/O Polling : I/O events are polled and CB functions are added to the I/O queue only after the I/O is complete (checks for completed I/O operations: If there are any completed I/O tasks, their associated callbacks are executed.)**

### *Check Queue :*

To queue a CB into this queue, we use:

- setImmediate()

### *Close queue :*

Its executes after all other queues CB in a given iteration of the event loop.

Attach close event listeners to queue into the close queue.

Eg, server.on('close', () => (register the close event listener)

```
 { console.log('Server has been closed.');
});
 server.close(() =>
{ console.log('Server shutdown complete.');
 });
```

nextTick and promise queues are executed in between each queue and also in between each call back execution in the timer and check queues.

## Routing :

Routing in Node.js refers to the process of defining how an application responds to client requests to different URLs (or paths) and HTTP methods (GET, POST, etc.). It involves setting up specific handlers for different endpoints

so that when a request hits a particular URL, the corresponding function is triggered to provide the appropriate response.

## Global Objects :

They are objects that are available in all modules without having to use require(). They can be accessed from anywhere in your application.

Eg, process,__filename,console,setTimeout,setImmediate,setInterval

## Events :

The events module allows us to work with events in nodeJs.

An event is an action or an occurrence that happened in our application that we can respond to.

Events module returns a class called **Event Emitter** which encapsulates functionality to emit events and respond to events.

Eg,
```
const Eventemitter=require('events')
const emitter=new Eventemitter()
emitter.on("Hello",()=>{
   console.log("Hi");
})
emitter.emit('Hello')
```

## fs Module :

The file system module allows you to work with the file system on your computer.

Reading a file;

Eg,
```
const fs=require('fs')
const fileContents=fs.readFileSync("./file.txt","utf-8")
console.log(fileContents);
```

readFileSync tells that method has a synchronous way of reading a file(js engine will wait till the file contents are read before moving onto next line).

To read asynchronously;

```
fs.readFile("./file.txt","utf-8",(error,data)=>{
   if(error){
      console.log(error);
   }
   else{
      console.log(data);
   }
})
```

Writing a file;

Eg, `fs.writeFileSync("./hello.txt","hello world")`

```
fs.writeFile("./hello.txt","Hello Navya",(err)=>{
   if(err){
      console.log(err);
   }
   else{
      console.log("File written");
   }
})
```

## Errors and Error handling

*Error object :*

Can manually create an error by using **throw** keyword.

Eg; `throw new Error("I am an error object")`

*Custom error:*

To create a custom error, extend the base Error class and add any additional behaviour or properties you need.

Eg;
```
class CustomError extends Error {
         constructor(message) {
         super(message);
        }
        }
        throw new CustomError("Something went wrong!");
```

*try and catch :*

```
try {
// Code that may throw an error
 } catch (error) {
 // Code that handles the error
 }
```

*Uncaught Exceptions:*

Refers to an error that occurs during the execution of a program but is not properly handled within a try...catch block. The process.on('uncaughtException', ...) event handler allows you to catch and handle uncaught exceptions.

Eg;

```
process.on('uncaughtException', (err) => {
console.error('There was an uncaught error:', err.message);
process.exit(1);
});
```

*Exceptions with promises:*

```
const myPromise = new Promise((resolve, reject) => {
  let success = true;
  if (success) {
    resolve("Operation succeeded");
  } else {
    reject("Operation failed");
  }
});
myPromise
  .then(result => {
    console.log(result);
  })
  .catch(error => {
    console.error(error);
  });
```

*With async/await:*

```
const readfile=async()=>{
try{
        await …
}
catch(err){
….
}}
```

## Modules:

A module is an encapsulated and reusable chunk of code that has its own context. In Node.Js each file is treated as a separate modules. Each module in Node.js has its own scope, meaning that variables and functions defined within a module are not available outside that module unless explicitly exported.

### *Local Modules:*

Modules that we create in our application.

To load a module into another file, we use the require function. We can export a module by using module.exports.

### *Built-in modules:*

Also referred to as core modules. Need to import the module before we use it.

path,events,fs,stream,http

- Path module:
  path.join() = Joins all given path segments together
  path.resolve() = path segments into an absolute path.
  path.parse() = that allows to break down a file path into its components
  path.extname() = Returns the extension of the path
  path.basename() = Returns the last portion of a path
  path.dirname() = Returns the directory name of a path

- Streams:
  It's a sequence of data that is being moved from one point to another overtime. Work with data in chunks(64kb) instead of waiting for the entire data to be available at once.
  Stream is a built-in module that inherits from the event Emitter class.

Eg;
```
const fs=require('fs')
const readableStream=fs.createReadStream('./file.txt','utf-8')
const writeableStream=fs.createWriteStream('./file2.txt')

readableStream.on('data',(chunk)=>{
        console.log(chunk);
        writeableStream.write(chunk)
})
```

Types of streams:

1. Readable streams : from which data can be read
2. Writable stream : to which we can write data
3. Duplex streams : that are both readable and writable
4. Transform streams : that can modify or transform data as it is written and read.

## Asynchronous Programming:

Web browsers and NodeJS define functions and APIs that allow us to register functions that should not be executed synchronously and should instead be invoked asynchronously when some kind of event occurs.

*async/await :*

Declaring a function as async ensures that it returns a promise. await keyword can only be used inside async function. It pauses the execution of the function until a Promise is resolved or rejected.

It's commonly used to wait for asynchronous operations like database queries, file reads, or HTTP requests. Also to avoid callback hell.

Eg;
```
const fs = require('fs').promises;
async function readFile() {
   try {
      const data = await fs.readFile('file.txt', 'utf-8');
      console.log(data);
} catch (error) {
```

```
        console.error('Error reading file:', error);
    }
    }
    readFile();
```

*Promises:*

A Promise represents the eventual completion (or failure) of an asynchronous operation and its resulting value. It allows asynchronous code to run like synchronous code, avoiding "callback hell"

It has three states;

**Pending, Fulfilled, Rejected**

Can handle the output of a promise by using **.then** for success and **.catch** for handling errors.

Eg;

```
const myPromise = new Promise((resolve, reject) => {
let success = true;
 if (success) {
resolve("Operation succeeded");
} else {
 reject("Operation failed");
}
});
myPromise
.then(result => {
console.log(result);
})
.catch(error => {
 console.error(error);
});
```

## Websocket :

It allows a real-time, full-duplex communication channel between the client and server. It provides bidirectional communication between the client and the server, meaning both can send and receive data at any time.