**🧵 Node.js — Single Threaded but Asynchronous**

JavaScript Core Nature (Applies to Node.js too). Because **Node.js is not a new language** — it's a **runtime environment** that runs **JavaScript** outside the browser.

**🔸 1. Is JavaScript Single Threaded?**

✅ **Yes.**

* JavaScript (in Node.js too) runs on **one thread only**.
* It can only do **one thing at a time** — no multitasking by default.

**🔸 2. What Does Synchronous Mean?**

✅ It means code runs **line by line**, **in order**.

console.log("A");

console.log("B");

console.log("C");

🧾 Output:

css

CopyEdit

A

B

C

👉 **Line 2 won't run** until Line 1 finishes.

**3. How Does Node.js Execute JS Code?**

* Node.js uses **V8 engine** (same as Chrome).
* The V8 engine **runs JS code immediately** when it receives it.
* We **don’t need multiple threads** — just one is enough to run JavaScript.

**4. Limitation: Only One Task at a Time**

❗ JavaScript can run only **one piece of code** at a time (due to the single thread).

If Line 3 is running, Line 4 has to wait.

**🔸 5. What Happens in Synchronous Programming?**

❌ **Blocking occurs**.

* If one task is slow (e.g. reading a file), it **stops everything else**.

const data = fs.readFileSync("bigFile.txt");

console.log("Done!"); // This line waits until file is fully read

⛔ **Problem**: Slow tasks like file read, DB call, API call will block the whole thread.

**🔸 6. Why This is a Problem in Node.js?**

Node.js is used to build **servers** (like APIs).  
If one task blocks, other users have to wait. That’s bad!

Imagine Swiggy stops taking new orders while it's cooking your noodles 🍜  
Not good, right?

**🔸 7. How Does Node.js Solve This?**

✅ With **Asynchronous & Non-blocking** features:

* Uses **libuv** under the hood (handles I/O with thread pool).
* Uses **Event Loop** to keep JS running smoothly.
* Offloads slow work (file read, network calls) to background.

**🔁 Difference Between Synchronous and Asynchronous Execution (Node.js Style)**

**🍕 Analogy: Restaurant Order Example**

You’re in a restaurant. There are 3 food items:

* 🥤 Coke → Ready instantly (0 min)
* 🍜 Noodles → Takes 5 mins
* 🍕 Pizza → Takes 10 mins

**🧾 Customer Order Table**

| **Customer** | **Ordered Item** | **🕒 Synchronous waitingTime** | **⚡ Asynchronous Time** |
| --- | --- | --- | --- |
| A | Coke | 0 min | 0 min |
| B | Noodles | 5 min | 5 min |
| C | Pizza | 15 min | 10 min |
| D | Coke | 15 min | 0 min |
| E | Noodles | 20 min | 5 min |

**📌 What Happens in Synchronous Execution?**

* Orders are **processed one by one**.
* Next order will start **only when** the previous one finishes.
* This creates a **waiting queue**, even for instant items like Coke.
* ❗ **Blocking behavior** — bad for performance in real-time apps.

🧠 Think of it as:

"Line 2 must wait for Line 1 to finish."

**📌 What Happens in Asynchronous Execution?**

* Orders are **processed independently**.
* Coke is served immediately.
* Noodles and Pizza are cooked **in parallel** (in the background).
* No one has to wait unnecessarily.

✅ This is called a **non-blocking operation**, and Node.js is built around this model.

🧠 Think of it as:

"Let slow tasks run in the kitchen. Don’t stop the cashier!"

**🧠 Why This Matters in Node.js?**

**🔸 Synchronous Model (❌ Not good for Node)**

js

CopyEdit

const data = fs.readFileSync('file.txt'); // ⛔ blocks the thread

console.log('This line waits');

**🔸 Asynchronous Model (✅ Ideal for Node)**

js

CopyEdit

fs.readFile('file.txt', (err, data) => {

console.log('File loaded');

});

console.log('I will not wait'); // ✅ runs immediately

**✅ Interview One-Liners**

| **💡 Question** | **🧠 Answer** |
| --- | --- |
| What is synchronous execution? | Tasks run one after another. Blocking. |
| What is asynchronous execution? | Tasks run independently. Non-blocking. |
| Which is better for Node.js? | Asynchronous, because Node is single-threaded and needs non-blocking behavior. |

**🔍 How Synchronous Code is Executed in Node.js (JS Engine View)**

**🧠 Code Example**

var a = 407429;

var b = 172410;

function multiplyFn(x, y) {

const result = x \* y;

return result;

}

var c = multiplyFn(a, b);

**⚙️ Behind the Scenes in Node.js**

Node.js uses the **V8 JavaScript Engine** (same as Chrome).  
This engine follows the **same execution model** for synchronous code as in any JS environment.

**🔄 Execution Flow in Synchronous Code**

1. **Global Execution Context (GEC)** is created  
   – This happens when the script first runs  
   – It's **pushed to the Call Stack**
2. a and b are declared and initialized  
   – Stored in the **Memory Heap**
3. multiplyFn(a, b) is called  
   – A **new Execution Context** for the function is created  
   – It’s **pushed onto the Call Stack**
4. Inside multiplyFn, result = a \* b is computed  
   – Stored temporarily in memory
5. Once return result runs:  
   – Function Execution Context is **popped off the Call Stack**  
   – Control goes back to the Global Execution Context  
   – Result is assigned to variable c
6. Once everything finishes, the **Call Stack becomes empty**

**🔁 Visual Summary (based on your diagram):**

📦 V8 Engine (in Node.js)

- Memory Heap (stores a, b, result)

- Garbage Collector (cleans unused memory)

- Call Stack:

1. GEC pushed

2. multiplyFn() pushed

3. multiplyFn() popped after return

4. GEC popped when done

**✅ One-Liner Summary for Interview**

"In Node.js, synchronous code is executed using the Call Stack. The V8 engine creates an Execution Context for each function, and memory is managed in the heap. The Call Stack runs top to bottom and clears once execution is complete."

How Node.js Handles Async Code Internally

### ✅ 2. Node.js Enables Non-Blocking Async via:

### 

### 🧠 1. ****Call Stack****

**Role**: Executes all your **synchronous JavaScript code** line by line.

✅ Think of it like a stack of plates –  
• Functions go on top  
• They're removed only when done

💡 Runs things like:

js

CopyEdit

console.log("A"); // Executes immediately

📌 **Limitation**: It cannot handle async tasks (like setTimeout or I/O) directly.  
So, those are "offloaded"...

### ⚙️ 2. ****libuv (Super Hero)****

### The V8 engine executes the synchronous parts of Node.js code. When it encounters an asynchronous operation, it offloads the responsibility to **libuv** — a C++ library used internally by Node.js.

libuv gives **Node.js superpowers** (timers , filesystem and all )to access the underlying OS-level capabilities for async operatons.

OS-level capabilities -

 **File system access**

 **Timers**

 **Thread pool (for CPU-bound tasks)**

 **Network (TCP/HTTP)**

 **DNS**

 **Child processes**

And many more …

📤 When JS sees an async task → it **offloads it to libuv**, and moves on.

✅ Now Let’s Deep dive into each async operations

### File System (📂 fs.readFile)

fs.readFile(\_\_filename, 'utf-8', (err, data) => {

if (!err) console.log('📂 File reading done');

});

* This is an **I/O operation**, so:
  + libuv offloads it to its **Thread Pool** (default 4 threads).
* Once the thread completes reading the file:
  + The callback is queued in the **Poll phase** of the Event Loop.

Note -

You can change the thread pool size using process.env.UV\_THREADPOOL\_SIZE = 8.

### 2. Timer (⏱ setTimeout)

setTimeout(() => {

console.log('⏱ Timer is done');

}, 500);

🧠 What happens when V8 encounters setTimeout()?

The JavaScript engine (V8) does not execute the callback immediately. Instead:

The setTimeout call is handed over to libuv, which registers it as a timer.

libuv puts it in its Timer System and sets up a minimum delay of 500ms.

After 500ms (not guaranteed to be exact), the callback is moved to the Timer Queue.

When the Event Loop reaches the Timers phase, it checks the Timer Queue.

If the delay has passed and the call stack is empty, the callback is executed.

⏱ Important Note on setTimeout(fn, 0)

setTimeout(() => {...}, 0) doesn’t run instantly.

Why?

Even though 0ms is passed, the callback will:

Wait until current synchronous code finishes.

Wait until all microtasks (like Promise.then or process.nextTick) are done.

Then run in the Timers phase of the next Event Loop tick.

**🌐 HTTP Request (using http.get) — How libuv Handles Network**

const http = require('http');

http.get('http://worldtimeapi.org/api/timezone/Asia/Kolkata', (res) => {

res.on('data', () => {}); // just consuming

res.on('end', () => {

console.log('🌐 HTTP request done');

});

});

Code explanation -

This code makes an HTTP GET request to a public API.

 The response comes in **chunks** (streamed), not all at once.

 Two event listeners are attached:

* res.on('data') → called whenever a piece of data arrives
* res.on('end') → called when the entire response is received

 Then the message 🌐 HTTP request done is printed.

## 🔁 Step-by-Step Execution of http.get(...) in Node.js

We'll now go deep into each of your 5 phases. 💡

### 🔹 ****Phase 1: Setup & Request Triggered****

JS code starts. Network request is initiated. But V8 does not handle the actual I/O.

#### ✅ What Happens:

* 🧠 **V8 executes the JS line**:

js

CopyEdit

http.get('http://worldtimeapi.org/api/timezone/Asia/Kolkata', ...)

* + http.get() is part of Node.js built-in HTTP module.
* 🔄 Node.js sees it’s a **non-blocking network operation**.

#### ✅ libuv Steps In:

* libuv (C++ engine) **sets up a TCP connection**.
* It uses **OS-level system calls**:
  + socket() → Create socket
  + connect() → Connect to server IP
  + send() → Send HTTP GET request

📦 **Why libuv?**

**JavaScript can’t directly interact with the operating system.**  
**libuv acts as an interface between JavaScript and the OS**, handling low-level tasks like file I/O, networking, and timers behind the scenes.

⏳ **While libuv manages these operations asynchronously, the JavaScript thread continues execution without waiting** — enabling non-blocking, efficient performance in Node.js.

### 🔹 ****Phase 2: Server Starts Sending Data****

Server responds to the GET request. Data starts flowing in chunks.

#### ✅ What Happens:

* The remote API server receives your request.
* It replies — **but not all at once!** Data is streamed in chunks like pieces of a large file.

#### ✅ Node Response:

* http.get() internally gives you a res object.
* This res is a **Readable Stream**.

Streams are Node.js interfaces for handling continuous data — like video buffering or reading large files.

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

res.on('end', () => { ... });

Think: You're downloading a movie — each piece comes in, you don’t wait for the whole file to start watching.

### 🔹 ****Phase 3: Response is Streamed (on**** 'data'****)****

Each chunk that arrives triggers a 'data' event.

#### ✅ What Happens:

* As each chunk of response data arrives:
  + The 'data' event is fired.
  + Your res.on('data', callback) gets called.

#### 🧠 Behind the Scenes:

* libuv **watches the socket** for incoming data using epoll/kqueue/IOCP.
* When data is detected:
  + It queues the callback into the **Poll Phase** of the Event Loop.
  + JS engine then picks it up and calls your data handler.

You can use these chunks to build the full response manually:

let result = '';

res.on('data', chunk => result += chunk);

### 🔹 ****Phase 4: Response Ends (on**** 'end'****)****

All chunks have been received — now it's done!

#### ✅ What Happens:

* Once the last chunk of data is received:
  + Node triggers the 'end' event.
  + Your callback:

res.on('end', () => {

console.log('🌐 HTTP request done');

});

is now called.

#### 🧠 Behind the Scenes:

* libuv informs Node that **no more data is coming**.
* Event Loop (Poll Phase again) picks up 'end' event.
* Callback is executed — this signals the end of response.

At this point, you've fully consumed the HTTP response.

### 🔹 ****Phase 5: Event Loop Handling****

Event Loop manages all your callback executions.

#### ✅ What Happens:

* All your callbacks (res.on('data'), res.on('end')) are treated as **I/O callbacks**.
* These are not microtasks or timers — they belong to the **Poll Phase** of the Event Loop.

#### 🧠 In Detail:

• When an I/O operation (like receiving data chunks) completes:  
    🔸 **libuv queues the associated callback inside the Poll Phase’s macro task queue**.

• During the Event Loop cycle:

1. ✅ JavaScript finishes executing all current synchronous code.
2. 🌀 The **Microtask Queue** (e.g., Promises, process.nextTick) is **fully drained**.
3. 🔁 The Event Loop enters the **Poll Phase**.
4. 📦 It executes the queued I/O callbacks (like 'data', 'end', etc.) in **FIFO order** from the Poll Phase's macro task queue.

🧠 Callbacks are executed in **FIFO (First-In, First-Out)** order.

In Node.js, do all Event Loop phases have their own microtask queues and macrotask queues?

**No.**

* ✅ Each **Event Loop phase** (like Timers, Poll, Check, etc.) has **its own Macro Task Queue**.
* ❌ But there is **only one global Microtask Queue** that is **shared across all phases**.

🌀 This Microtask Queue includes:

* Promise.then()
* queueMicrotask()
* process.nextTick() (Node-specific)

#### Q1. Where are callbacks from setTimeout, fs.readFile, and setImmediate queued?

**A:**

* setTimeout → Timers Phase macro task queue
* fs.readFile → Poll Phase macro task queue
* setImmediate → Check Phase macro task queue

#### Q3. Is there a separate microtask queue for each phase?

**A:**  
No. There's only **one microtask queue**, and it's global. It runs after each macro task, no matter which phase it came from.

## ❓ Does the ****Microtask Queue**** run before or after macro tasks?

### 🧠 ✅ Here's the Real Execution Rule:

**Microtasks run after the currently executing macro task finishes, but before the Event Loop moves on to the next macro task (or next phase).**

### 🔥 Rule Breakdown:

1. 🧾 A **macro task** (from any phase) starts running
2. 🌀 During this macro task, you might schedule **microtasks** (e.g., Promise.then, nextTick)
3. ✅ When the macro task finishes, Node will:
   * Pause the Event Loop from moving to the next phase
   * **Run ALL queued microtasks (in order)**
4. Only then does the Event Loop continue to the next macro task or phase

## 🔁 Real-World Example:

js

CopyEdit

setTimeout(() => {

console.log('⏱ Timer macro task');

Promise.resolve().then(() => {

console.log('🌀 Microtask inside timer');

});

}, 0);

console.log('🧠 Sync code');

**🔁 Real-World Example:**

js

CopyEdit

setTimeout(() => {

console.log('⏱ Timer macro task');

Promise.resolve().then(() => {

console.log('🌀 Microtask inside timer');

});

}, 0);

console.log('🧠 Sync code');

**✅ Output:**

rust

CopyEdit

🧠 Sync code

⏱ Timer macro task

🌀 Microtask inside timer

**❓ Why does the microtask run after timer, but before next task?**

Because:

* setTimeout callback is a macro task
* While executing that callback, we queue a microtask
* **Microtask runs right after this macro task ends**
* Then, Event Loop moves forward

**✅ Interview-Ready One-Liner:**

Microtasks don’t preempt macro tasks — they wait until the **current macro task completes**, and then **run before the next one starts**, giving them a **higher priority in execution order**.

**✅ 1. Real Code Example — Show Chunked Data + Full Response**

js

CopyEdit

const http = require('http');

http.get('http://worldtimeapi.org/api/timezone/Asia/Kolkata', (res) => {

let fullData = '';

// Step 1: Handle each chunk

res.on('data', (chunk) => {

console.log('📦 Received chunk:', chunk.toString()); // log chunk-by-chunk

fullData += chunk; // build full response

});

// Step 2: Handle end of response

res.on('end', () => {

console.log('✅ All data received.');

console.log('🌐 Full response:\n', fullData); // final complete response

});

// Optional: Handle error

res.on('error', (err) => {

console.error('❌ Error:', err);

});

});

**🧠 Output Example:**

When you run this, you’ll see something like:

css

CopyEdit

📦 Received chunk: {

📦 Received chunk: "abbreviation":"IST",

📦 Received chunk: "datetime":"2025-07-31T19:43:12+05:30",

...

✅ All data received.

🌐 Full response:

{"abbreviation":"IST","datetime":"2025-07-31T19:43:12+05:30",...}

Each 📦 Received chunk: line is a **piece of the response body**.  
When everything is received, 'end' fires, and you get the full response.

**✅ Why Does This Happen?**

* http.get() uses a **Readable Stream** (res)
* Streams are **chunked by nature** → You don’t get the full data at once
* You must:
  + Collect each chunk in 'data'
  + Wait for 'end' to get the full thing

**🧠 Real-World Use Case:**

This is **exactly** how Node handles:

* File streaming (fs.createReadStream)
* Large HTTP downloads
* API responses that return big JSON or file data

**4️⃣ DNS Lookup (🌍 dns.lookup)**

### 📌 DNS = ****Domain Name System****

It’s like the **phonebook of the internet**.

### Real-world Example:

When you type this in your browser:

https://www.google.com

👉 Your computer doesn’t understand "google.com" directly.

🔍 It needs the **IP address** of Google’s server (like 142.250.193.206) to connect.

So your computer asks:

“Hey DNS, what is the IP address of google.com?”

And DNS replies:

“Here you go: 142.250.193.206”

That process is called **DNS Lookup** ✅

## 🧠 What is dns.lookup() in Node.js?

const dns = require('dns');

dns.lookup('google.com', (err, address) => {

if (!err) console.log('🌍 DNS resolved to:', address);

});

This line of code tells Node.js:

“Please find out the IP address of google.com, and once you do, run this callback function.”

**Use case –**

Once dns.lookup() resolves the domain, the callback receives the IP address.  
Inside that callback, we use the IP to send a request to the HTTPS server.  
To ensure the server knows which domain we intended

## 🔁 What Happens Behind the Scenes?

1. You call dns.lookup('google.com')
2. Node uses a system function called getaddrinfo() to perform the lookup
3. This is a **slow, blocking operation** — so:
   * It is **offloaded to libuv's thread pool** (background worker)
4. When it's done:
   * The callback is added to the **Poll Phase queue**
5. The Event Loop later runs that callback and prints the IP

## 📦 In Simple Words:

dns.lookup() asks your operating system:

🗣️ “What is the IP address of this website name?”

It gets the answer, and then gives it back to you inside a callback.

## 🧪 Output Example:

🌍 DNS resolved to: 142.250.193.206

## ✅ Why Use This in Node.js?

Imagine you’re building a program that:

* Checks if a domain exists
* Connects to a server
* Verifies internet access

All of these start by **resolving a domain name to an IP**, and that’s exactly what dns.lookup() does.

const dns = require('dns');

const http = require('http');

// Step 1: Lookup the IP address of the domain

dns.lookup('worldtimeapi.org', (err, address) => {

if (err) return console.error('❌ DNS error:', err);

console.log('🌍 DNS resolved to:', address);

// Step 2: Make HTTP request using the IP address

const options = {

host: address, // Use resolved IP instead of domain

path: '/api/timezone/Asia/Kolkata',

headers: {

Host: 'worldtimeapi.org' // Required so server knows which domain you meant

}

};

http.get(options, (res) => {

let data = '';

res.on('data', (chunk) => {

data += chunk;

});

res.on('end', () => {

console.log('✅ Full response received:\n', data);

});

res.on('error', (e) => {

console.error('❌ HTTP Error:', e);

});

});

});

**Explanation:**

**🔹 Step 1: DNS Lookup**

js

CopyEdit

dns.lookup('worldtimeapi.org', ...)

* This asks **your OS** to resolve the domain name to its IP address.
* The result (e.g., 34.160.111.145) is returned asynchronously in a callback.

**🔹 Step 2: Make HTTP Request to the IP**

js

CopyEdit

http.get({

host: address,

path: '/api/timezone/Asia/Kolkata',

headers: {

Host: 'worldtimeapi.org'

}

}, ...)

* Now you use the **IP address** instead of the domain to connect.
* But you **must include the Host header** so the server knows you're referring to worldtimeapi.org.
* This is critical for **virtual hosting**, where multiple domains live on the same IP.

**🔹 Event Flow:**

1. JS runs the code, V8 executes dns.lookup
2. Node offloads it to **libuv thread pool** (since DNS is blocking)
3. When IP is resolved, callback is queued into the **Poll Phase**
4. You make an HTTP request
5. Response is streamed chunk-by-chunk (data)
6. At the end (end), you get the full response

**✅ Output Example:**

bash

CopyEdit

🌍 DNS resolved to: 34.160.111.145

✅ Full response received:

{

"abbreviation": "IST",

"datetime": "2025-07-31T20:45:12+05:30",

...

}

**🔥 Interview Insight:**

dns.lookup() resolves a domain to an IP using the OS resolver.  
After resolution, you can make a low-level HTTP request using the IP, but you must include the Host header to preserve the original domain's context.

# 🔐 crypto.pbkdf2() — Step-by-Step from Scratch

# const crypto = require('crypto');

# crypto.pbkdf2('secret', 'salt', 100000, 64, 'sha512', () => { console.log('🔐 Password hashed'); });

**🔐 Code Breakdown: Line-by-Line**

js

CopyEdit

const crypto = require('crypto');

**✅ What it does:**

* This line **imports Node.js’s built-in crypto module**.
* This module contains tools for:
  + Hashing
  + Encryption/decryption
  + Secure key generation
  + Other cryptography operations(batman arkham city lol)

📦 It's like saying:

“I want to use Node’s security tools.”

crypto.pbkdf2('secret', 'salt', 100000, 64, 'sha512', () => {

console.log('🔐 Password hashed');

});

Let’s break this big line into each part 👇

## 🧠 What’s this code doing?

* It **hashes a password** using a strong algorithm (sha512)
* Makes it hard for hackers to guess the password
* Used in login/signup systems to protect user credentials

**Steo by step code explnatioin -**

**🔹 crypto.pbkdf2(...)**

This is the **function that performs the secure hashing**.  
It uses a hashing algorithm called **PBKDF2** (Password-Based Key Derivation Function 2)

**🔹 'secret'**

This is the **input password or string** you want to hash.  
Think of this as the user’s password.

🧠 In real apps, this would come from user input like:

js

const password = req.body.password;

**🔹 'salt'**

A **random string added to the password** to make it unique before hashing.  
Even if two users have the same password, their hash will look different with different salts.

🧠 Why?

* Prevents **rainbow table attacks**
* Makes brute-force harder

**🔹 100000**

This is the **number of iterations** (rounds of hashing).  
The higher the number:

* ✅ More secure
* ❌ Slower

🧠 100,000 rounds means:

100000 tells Node.js to repeat the hashing process 100,000 times to make password cracking harder and safer for real-world use.

**🔹 64**

This is the **length of the output hash** (in bytes).

So the result will be a **64-byte** hash (or 128-character string in hexadecimal).

**🔹 'sha512'**

This is the **hashing algorithm used** under the hood.

* 'sha512' = Strong, secure, and long output
* Other options: 'sha256', 'sha1', etc.

🛑 Never use sha1 today — it's weak and outdated.

**🔹 () => { console.log('🔐 Password hashed'); }**

This is the **callback function** that runs **after hashing is complete**.

🧠 Node uses libuv’s **thread pool** to run the hashing in the background.  
When it finishes:

* Your callback is queued in the **Poll Phase**
* Then executed by the Event Loop

## 🔁 Detailled Step-by-Step Execution Flow

### ✅ 1. Node v8 engine runs the code

* Your main code hits:

js

CopyEdit

crypto.pbkdf2(...)

* This is a **built-in Node.js function**
* It’s a **CPU-heavy task** (not a timer or network request)

### ✅ 2. Node passes the task to ****libuv****

* Node knows hashing is heavy work 🧠💪
* So it sends this task to **libuv**, a helper engine inside Node
* libuv doesn’t run the task directly either

### ✅ 3. libuv puts the work in the ****thread pool****

* libuv has a small team of background workers (default 4 threads)
* These workers run **heavy stuff in the background**
* One worker picks up the pbkdf2 task and starts hashing

### ✅ 4. Meanwhile... JavaScript keeps running 😎

* The main thread doesn’t wait
* Other code continues running without blocking

### ✅ 5. Hashing completes ✅

* After maybe 300ms or more (based on workload)
* The background worker finishes the task
* It sends the result (hashed output) back to libuv

### ✅ 6. libuv queues the callback

* libuv adds the callback (() => { console.log(...) }) to the **Poll Phase** of the **Event Loop**

### ✅ 7. Event Loop reaches Poll Phase

* Node’s event loop checks the **Poll Phase**
* It sees your pbkdf2 callback is waiting
* It runs the callback

console.log('🔐 Password hashed');

✅ Done!

## 🧠 Easy One-Liner Summary:

crypto.pbkdf2() does password hashing in the background using libuv’s thread pool, and when done, its callback runs in the Event Loop’s Poll Phase — keeping your app fast and non-blocking.

**🔧 How to Increase Thread Pool Size in Node.js**

**✅ Default Thread Pool Size**

* Node.js (via libuv) uses **4 threads by default**
* This is fine for small tasks
* But when you run **more than 4 parallel CPU-heavy tasks**, the extras are put in a **waiting queue**

**🔥 How to Increase It?**

**📍 Option 1: Set an environment variable before running your script**

UV\_THREADPOOL\_SIZE=8 node app.js

✅ This sets the pool to **8 threads**

Steps -

### ✅ 1. Open ****Command Prompt (CMD)****

### ✅ 2. Navigate to your project folder:

cd C:\Users\Likan\Projects\NodeApp

### ✅ 3. Set the thread pool size and run the file:

#### 👉 On Windows CMD:

cmd

CopyEdit

set UV\_THREADPOOL\_SIZE=8

node app.js

✅ Done! Now Node.js will run your app with **8 background threads**.

**we have to go to the folder where your app.js file is**,  
then run above commands to increase the worker thread size.

**📍 Option 2: Set it in code (must be done before any async tasks)**

process.env.UV\_THREADPOOL\_SIZE = 8;

const crypto = require('crypto');

for (let i = 0; i < 8; i++) {

crypto.pbkdf2('secret', 'salt', 100000, 64, 'sha512', () => {

console.log(`🔐 Task ${i + 1} done`);

});

}

**⚠️ Max Limit?**

* You can increase the pool up to **128 threads**, but be careful:
  + Too many threads = too much memory and CPU pressure
  + Ideal range: **4 to 16**, depending on your server

**🧠 Final Summary (Interview-Ready)**

Yes — Node.js allows increasing the thread pool size using the UV\_THREADPOOL\_SIZE variable.  
This helps when you have **many parallel CPU-bound tasks**, so more threads can work **at the same time** instead of queuing up.

**6️⃣ Child Process (👶 exec)**

const { exec } = require('child\_process');

exec('echo Hello from terminal', (err, stdout) => {

if (!err) console.log('👶 Child says:', stdout.trim());

});

🔍 Step-by-Step Explanation:

✅ 1. require('child\_process')

const { exec } = require('child\_process');

This line imports the built-in Node.js module called child\_process.

From that module, we extract just the exec() function using destructuring.

exec() is used to run OS-level shell commands from Node.js.

✅ 2. Calling exec()

exec('echo Hello from terminal', ...);

exec() takes a string command like you’d run in CMD/bash/terminal.

Here, 'echo Hello from terminal' is a basic shell command that prints text to output.

📦 This is just like opening your terminal and typing:

echo Hello from terminal

✅ 3. What Happens Behind the Scenes

🔧 Node gives the command to libuv

⚙️ libuv uses OS system calls (like fork() or CreateProcess)

🧠 The OS spins up(launches) a child process

🖥️ That child process runs inside the default shell:

Windows → cmd.exe

Linux/macOS → /bin/sh

🧾 The command executes → and its stdout (output) is captured

✅ 4. Callback Function

(err, stdout) => {

if (!err) console.log('👶 Child says:', stdout.trim());

}

🔁 After the command finishes, Node runs this callback

err → If there was an error running the command (e.g., command not found)

stdout → Whatever was printed by the command

stderr → (Not used here) — any error output from the command

🧼 .trim() is used to remove any extra newline or space at the end.

✅ Output:

👶 Child says: Hello from terminal

**Example 2 -**

const { exec } = require('child\_process');

exec('echo Hello', (err, stdout) => {

console.log('👶 Child says:', stdout.trim());

});

console.log('✅ Code moved on...');

**🔁 Step-by-Step Execution Flow of exec() in Node.js**

**✅ 1️⃣ V8 Starts Executing Your Code**

* Node.js uses the V8 engine to run JavaScript **line by line**.
* First, it sees:

js

CopyEdit

const { exec } = require('child\_process');

* ✅ This just loads the exec function from Node’s internal library.
* Nothing asynchronous yet.

**✅ 2️⃣ V8 Calls exec()**

js

CopyEdit

exec('echo Hello', (err, stdout) => {

console.log('👶 Child says:', stdout.trim());

});

* V8 **encounters this function call**
* V8 doesn’t know how to run shell commands itself
* So it **passes the task** to Node’s **C++ internals via bindings**

**✅ 3️⃣ libuv Offloads the Task to the OS**

* exec() is handled by **libuv**, Node’s internal async engine
* libuv says:

“Hey OS, please run this command for me.”

* It does **not** run the command — it just asks the OS to create a process and handle the rest.

**✅ 4️⃣ OS Spawns a Child Process and Runs the Command**

* The **Operating System** launches a new **child shell process** (like bash, sh, or cmd)
* Inside that shell, the command runs:

bash

CopyEdit

echo Hello

* The shell prints Hello\n and exits
* The OS captures:
  + stdout: "Hello\n"
  + stderr: if any error happened

**✅ 5️⃣ Meanwhile, JS Continues**

* Right after the exec() call, V8 **keeps moving forward**
* This line runs immediately:

console.log("✅ Code moved on...");

* So the output order will be:

csharp

CopyEdit

✅ Code moved on...

🧠 Proof: exec() is **non-blocking** — JS continues execution.

**✅ 6️⃣ OS Sends the Output Back to libuv**

* Once the shell command finishes:
  + OS hands back the result (stdout + stderr) to **libuv**
  + Now the task is complete

**✅ 7️⃣ libuv Queues Your Callback in the Poll Phase**

* The callback:

(err, stdout) => { ... }

is placed in the **Poll Phase macrotask queue** of the **Node.js Event Loop**

* This means it's scheduled to run later — after all sync code + microtasks are done

**✅ 8️⃣ Event Loop Reaches Poll Phase and Executes the Callback**

* Once current tasks and microtasks are complete…
* The Event Loop enters the **Poll Phase**
* It finds the callback waiting, and executes it:

js

CopyEdit

console.log('👶 Child says:', stdout.trim());

* The final output is:

👶 Child says: Hello

**✅ Final Recap (For Interviews):**

exec() is an async function in Node.js that runs terminal commands in the background by spawning a child process via the OS. V8 initiates the call, libuv hands it to the OS, and the result is returned through the Poll Phase of the Event Loop.

# 👶 What is a ****Child Process**** in Node.js?

A **Child Process** is a separate program (or shell command) that Node.js can run **outside the main thread**.

It’s like telling your operating system:

“Hey, run this command/script in the background while I do other stuff.”

🧠 Real Use Case:

Running Git commands, Docker, or scripts via Node.

**🌀 process.nextTick() in Node.js — Explained in Detail**

**✅ What Is process.nextTick()?**

It's a special microtask queue in Node.js — more powerful than Promises.

📌 **process.nextTick() schedules a callback to run immediately after the current synchronous code finishes, but before the Node.js Event Loop proceeds to the next phase.**

**🔧 Syntax:**

js

CopyEdit

process.nextTick(() => {

console.log("🌀 nextTick runs first");

});

**🔄 Execution Flow**

Let’s say you have this:

js

CopyEdit

console.log("1");

process.nextTick(() => console.log("🌀 nextTick"));

Promise.resolve().then(() => console.log("🔁 Promise"));

console.log("2");

**🧠 Output:**

javascript

CopyEdit

1

2

🌀 nextTick

🔁 Promise

**👇 Step-by-Step:**

1. **"1"** is printed — sync code ✅
2. **process.nextTick()** is registered — 🌀 added to the nextTick queue
3. **Promise.then(...)** is registered — 🔁 added to the microtask queue
4. **"2"** is printed — sync code ✅
5. Now, **nextTick runs before Promises**
6. Then **Promise runs**

Usecase –

process.nextTick() is perfect when you want to delay a callback **just enough** to let the current sync code finish, without waiting for Promises or timers. It's often used in error handling, state management, and core Node internals like streams."

## 🧠 Where is process coming from in process.nextTick()?

### ✅ It’s a ****global object**** in Node.js — just like window in the browser.

You **don’t need to import** or require anything.

});

You're just using Node’s built-in global API — no import needed ✅

You can even log the entire object to explore what's inside:

js

CopyEdit

console.log(process);

It contains:

* process.env → environment variables
* process.argv → CLI arguments
* process.pid → process ID
* process.cwd() → current working dir
* process.nextTick() → microtask queue scheduler

**8️⃣ setImmediate**

**📘 setImmediate() in Node.js — Full Notes (All Cases Covered)**

**✅ What is setImmediate()?**

setImmediate() is a Node.js function that schedules a callback to be executed **after the current Poll phase**, during the **Check phase** of the Event Loop.

**🔧 Syntax**

js

CopyEdit

setImmediate(() => {

console.log("✅ I ran in the Check phase");

});

**🧠 Core Concept**

| **Feature** | **Value** |
| --- | --- |
| Phase | ✅ Check Phase |
| Execution Order | After I/O callbacks (Poll Phase ends) |
| Priority | Higher than setTimeout(..., 0) *after I/O* |
| Nature | Asynchronous |
| Event Loop Context | Macrotask in the Check Phase |

**🧪 Case 1: 📂 Inside an I/O Operation or inside pool phase (Best Practice)**

js

CopyEdit

const fs = require("fs");

fs.readFile(\_\_filename, () => {

console.log("📂 File Read Done");

setImmediate(() => {

console.log("✅ setImmediate after I/O");

});

setTimeout(() => {

console.log("⏱ setTimeout after I/O");

}, 0);

});

**🔍 Output:**

mathematica

CopyEdit

📂 File Read Done

✅ setImmediate after I/O

⏱ setTimeout after I/O

**🧠 Why?**

* After the I/O callback finishes (fs.readFile), Node enters:
  + 👉 First: Check Phase → setImmediate() runs ✅
  + 👉 Then: Timers Phase → setTimeout() runs ⏱

**🧪 Case 2: ⏱ Outside I/O/poll phase (No Guaranteed Order)**

js

CopyEdit

setImmediate(() => {

console.log("✅ setImmediate");

});

setTimeout(() => {

console.log("⏱ setTimeout 0");

}, 0);

**🔍 Output (Can vary):**

* One run:

arduino

CopyEdit

✅ setImmediate

⏱ setTimeout 0

* Another run:

arduino

CopyEdit

⏱ setTimeout 0

✅ setImmediate

**⚠️ Why No Guarantee?**

Outside of I/O, both are scheduled at the same time. The order depends on:

* OS timing
* CPU load
* Internal implementation

**🔁 How It Works Internally**

text

CopyEdit

┌──────────────────────────┐

│ Node.js Code │

└──────────────────────────┘

│

▼

✔️ Synchronous Code Runs

│

▼

✅ Event Loop starts

│

▼

📤 I/O Callbacks handled in Poll Phase

│

▼

✅ setImmediate() runs in Check Phase

▼

⏱ setTimeout(..., 0) runs in Timers Phase

## 💬 Interview One-Liners

* "setImmediate() is a Node.js macro task scheduled in the Check phase — ideal for running logic right after async I/O callbacks complete."
* "Unlike setTimeout, setImmediate ensures your callback runs \*\*after I/O\*\*, making it predictable in I/O-heavy flows."
* "setImmediate and setTimeout(0) are not the same — their order is not guaranteed outside I/O."

**🧰 Real-World Use Cases of setImmediate()**

**1️⃣ Run Logic After I/O (Best Use Case)**

When you want your code to execute *after* a file is read, a network call is completed, or a database query finishes.

js

CopyEdit

fs.readFile('data.txt', () => {

// File is read ✅

setImmediate(() => {

console.log("✅ Processing after file read");

});

});

✅ Useful when:

* You want to log something or process the file only after I/O is fully done.
* You want to ensure your logic **doesn't block the Poll phase**.

### 4️⃣ ****Safe Post-Callback Actions****

You want to ensure something runs after all current logic and callbacks.

js

CopyEdit

someAsyncTask(() => {

setImmediate(() => {

console.log("✅ Cleanup or next step runs after async");

});

});

✅ This gives async operations time to settle before continuing with final logic.

## 📘 What is the "Close Callbacks Phase"?

The **Close Callbacks Phase** is the **last phase** in the Node.js Event Loop.

🧠 It runs **after**:

* Timers ⏱
* Pending Callbacks
* Idle/Prepare
* Poll (I/O)
* Check (✅ setImmediate)
* **👉 Then: Close Callbacks**

## 🧨 When does this phase run?

This phase handles **cleanup callbacks** for:

* net.Socket or fs streams
* Any object that **emits a 'close' event**
* File or network handles that were **forcefully closed**

**🔌 Real Example with net.Socket**

const net = require('net');

// 1️⃣ Create a connection to some server (you can start a local TCP server or connect to google.com)

const socket = net.createConnection({ host: 'example.com', port: 80 }, () => {

console.log('🌐 Connected to server');

// 2️⃣ Immediately destroy the socket (forcefully close it)

socket.destroy();

});

// 3️⃣ Listen to 'close' event — this runs during the Close Callbacks Phase

socket.on('close', () => {

console.log('🔚 Socket closed — running in Close Callback phase');

});

**✅ Output:**

pgsql

CopyEdit

🌐 Connected to server

🔚 Socket closed — running in Close Callback phase

**🧠 Explanation (Step-by-step)**

1. net.createConnection(...) initiates a TCP connection to the server (like example.com).
2. Once connected, the callback runs and immediately socket.destroy() is called — this forcibly closes the connection.
3. The 'close' event is triggered by Node.js and queued in the **Close Callbacks Phase**.
4. The callback inside socket.on('close', ...) runs **after all other phases**, just before the event loop restarts.

**📌 When is the "close" event triggered?**

* When you **forcefully destroy** a socket (socket.destroy()).
* Or when a stream is **automatically closed** (like file streams, HTTP streams).
* It’s not triggered by default like 'end' or 'finish'. You must **listen explicitly** to it.

**💬 Interview One-Liner:**

"on('close') events in Node.js are handled in the Close Callbacks Phase — perfect for socket or stream cleanup when resources are closed."

### 🔁 3. ****Event Loop****

**Role**: It's the **traffic controller** between the Call Stack and the background tasks.

🔄 Keeps checking:

* Is the call stack empty?
* Are there any tasks ready to be executed?

📌 When a background task (like fs.readFile) is done:

* Event Loop pushes its **callback** onto the call stack if the stack is empty
* JS engine v8 finally executes it

💡 This is how Node.js handles **non-blocking async behavior**

## 🤖 Who Controls the Event Loop in Node.js?

### ✅ libuv ****controls the event loop****

* The **event loop is implemented inside libuv** (a C-based library)
* It's not written in JavaScript
* It's always running in the background — a **C++ "while loop"** that manages task queues

### 📦 4. ****Queues (Task Queues)****

**Role**: Store **tasks/callbacks** waiting to run.

Two main types:

#### a. 🧬 **Microtask Queue**

* Promises
* process.nextTick() (Node.js only)
* Runs **before** the macrotasks

#### b. 🕒 **Macrotask Queue**

* setTimeout, setInterval
* setImmediate
* I/O callbacks

📌 **Order of execution**:

1. All synchronous code (call stack)
2. Microtasks (Promise callbacks)
3. Then Macrotasks (Timers, I/O callbacks)

**🧠 Who Controls Microtask & Macrotask Queues in Node.js?**

| **Queue Type** | **Controlled by** | **Description** |
| --- | --- | --- |
| 🔬 **Microtask Queue** | ✅ **V8 JavaScript Engine** | For promises & process.nextTick() |
| 🕓 **Macrotask Queue** | ✅ **libuv (inside Node.js)** | For timers, I/O, setTimeout, setImmediate |

### 🧠 Node.js Event Loop Phases — Explained One by One

|  |  |  |  |
| --- | --- | --- | --- |
| **1️⃣ 🟢 Sync Phase (Top-Level Code)**  **Explanation:**   * This is **your main script code** — anything outside async callbacks or timers. * It runs immediately when the file is executed. * It’s executed on the **Call Stack**, one line at a time.   **Example:**  js  CopyEdit  console.log("Start"); // runs immediately  **2️⃣ ⚡ Microtasks Phase (process.nextTick, Promises)**  **Explanation:**   * These are **high-priority async tasks** that run **after sync code but before any other phase**. * Includes:   + process.nextTick() → Node.js only   + Promise.then(), .catch(), .finally() → Standard microtasks   **Order:**  process.nextTick() > Promise.then()  **Example:**  CopyEdit  process.nextTick(() => console.log("Tick"));  Promise.resolve().then(() => console.log("Promise"));  **3️⃣ ⏰ Timer Phase (setTimeout, setInterval) – MACRO TASK QUEUE**  **Explanation:**   * Executes callbacks scheduled with:   + setTimeout(fn, 0)   + setInterval(fn, x) * These tasks are handled by libuv and executed **after their timer expires**.   **Example:**  setTimeout(() => console.log("Timeout"), 0);  **4️⃣ 📥 Poll Phase (I/O Callbacks)**  **Explanation:**   * Handles **I/O-related callbacks**:   + fs.readFile, net, http, TCP, DNS * This phase is **DYNAMIC**:   + If an I/O callback is ready → runs immediately   + If not → may **pause** and **wait**   **Example:**  js  CopyEdit  fs.readFile("file.txt", () => console.log("Read complete"));  **5️⃣ ✅ Check Phase (setImmediate)**  The **Check Phase** is a part of the Node.js Event Loop where Node runs the code you gave it using **setImmediate()**.  **Example:**  setImmediate(() => console.log("Immediate"));  **6️⃣ ❌ Close Callbacks Phase**  **Explanation:**   * Runs cleanup functions for:   + .on('close')   + Socket/file handle shutdowns   + Streams or net connections ending   **Example:**  socket.on("close", () => console.log("Socket closed"));  ✅ **Fixed Position:** Yes 🧠 Think: *"Cleanup for closed handles (streams, sockets, etc.)"* |  |  |  |

## 🧪 PRACTICAL: Ultimate Async Code (with All Phases In + Out of fs.readFile)

js

CopyEdit

const fs = require("fs");

console.log("1️⃣ start");

// 🔸 OUTSIDE

process.nextTick(() => console.log("2️⃣ outer nextTick"));

Promise.resolve().then(() => console.log("3️⃣ outer Promise"));

setTimeout(() => {

console.log("4️⃣ outer setTimeout");

}, 0);

setImmediate(() => {

console.log("5️⃣ outer setImmediate");

});

// 🔸 I/O Phase

fs.readFile(\_\_filename, () => {

console.log("6️⃣ fs.readFile callback (Poll Phase)");

process.nextTick(() => console.log("7️⃣ inner nextTick"));

Promise.resolve().then(() => console.log("8️⃣ inner Promise"));

setTimeout(() => {

console.log("9️⃣ inner setTimeout");

process.nextTick(() => console.log("🔟 nested nextTick"));

Promise.resolve().then(() => console.log("🔁 nested Promise"));

setImmediate(() => console.log("🔂 nested setImmediate"));

setTimeout(() => console.log("🕒 nested setTimeout"), 0);

}, 0);

setImmediate(() => console.log("🧭 inner setImmediate"));

});

console.log("✅ end");

## 🔢 STEP-BY-STEP EXECUTION (With Phases Explained)

### 1️⃣ SYNC PHASE (Call Stack)

js

CopyEdit

console.log("1️⃣ start");

console.log("✅ end");

✅ Immediate execution

🧾 Output:

sql

CopyEdit

1️⃣ start

✅ end

### 2️⃣ MICROTASKS (After Sync)

js

CopyEdit

process.nextTick(...) → 2️⃣

Promise.then(...) → 3️⃣

🧾 Output:

sql

CopyEdit

2️⃣ outer nextTick

3️⃣ outer Promise

### 3️⃣ TIMERS PHASE

js

CopyEdit

setTimeout(...); // 4️⃣

🧾 Output:

arduino

CopyEdit

4️⃣ outer setTimeout

### 4️⃣ CHECK PHASE

js

CopyEdit

setImmediate(...); // 5️⃣

🧾 Output:

sql

CopyEdit

5️⃣ outer setImmediate

### 5️⃣ POLL PHASE — fs.readFile() callback (if ready)

js

CopyEdit

console.log("6️⃣ fs.readFile callback");

🧾 Output:

java

CopyEdit

6️⃣ fs.readFile callback (Poll Phase)

### 6️⃣ MICROTASKS INSIDE I/O CALLBACK

js

CopyEdit

process.nextTick(...) → 7️⃣

Promise.then(...) → 8️⃣

🧾 Output:

kotlin

CopyEdit

7️⃣ inner nextTick

8️⃣ inner Promise

### 7️⃣ CHECK PHASE → setImmediate inside fs callback

js

CopyEdit

setImmediate(...) → 🧭

🧾 Output:

kotlin

CopyEdit

🧭 inner setImmediate

### 8️⃣ TIMERS PHASE (Next Round) → inner setTimeout fires

js

CopyEdit

console.log("9️⃣ inner setTimeout");

🧾 Output:

arduino

CopyEdit

9️⃣ inner setTimeout

### 9️⃣ MICROTASKS INSIDE THAT TIMEOUT

js

CopyEdit

nextTick → 🔟

Promise.then → 🔁

🧾 Output:

javascript

CopyEdit

🔟 nested nextTick

🔁 nested Promise

### 🔟 CHECK PHASE → nested setImmediate

js

CopyEdit

setImmediate(...) → 🔂

🧾 Output:

CopyEdit

🔂 nested setImmediate

### 🕒 TIMERS PHASE AGAIN → nested setTimeout

js

CopyEdit

setTimeout(...) → 🕒

🧾 Output:

arduino

CopyEdit

🕒 nested setTimeout

## ✅ FINAL OUTPUT ORDER:

sql

CopyEdit

1️⃣ start

✅ end

2️⃣ outer nextTick

3️⃣ outer Promise

4️⃣ outer setTimeout

5️⃣ outer setImmediate

6️⃣ fs.readFile callback (Poll Phase)

7️⃣ inner nextTick

8️⃣ inner Promise

🧭 inner setImmediate

9️⃣ inner setTimeout

🔟 nested nextTick

🔁 nested Promise

🔂 nested setImmediate

🕒 nested setTimeout

**⚙️ How Code Executes in Node.js When Both Sync & Async Operations Are Present**

**🧪 Sample Code (From Notes):**

var a = 427920;

var b = 127221;

https.get("https://api.fbi.com", (res) => {

console.log(res?.secret);

});

function multiplyFn(x, y) {

const result = x \* y;

return result;

}

var c = multiplyFn(a, b);

console.log(c);

**🔁 What Happens Behind the Scenes (Step-by-Step)**

**🔸 1. GEC (Global Execution Context) is Created**

* The **JS engine** starts by creating a **Global Execution Context**
* It’s pushed onto the **Call Stack**
* JS runs all code **synchronously** (line-by-line) inside the GEC

**🔸 2. Memory Phase**

* JS allocates memory to:
  + a, b, c → inside **Memory Heap**
  + Garbage Collector watches unused memory for cleanup later

**🔸 3. https.get() (Async API Call Detected)**

* JS engine **doesn't wait** for the result
* It **offloads** this task to **libuv**
* libuv:
  + Registers the API call
  + Attaches the callback function
  + Hands the task to the OS (for networking)
* Meanwhile, JS engine **moves on to the next line**

**🔸 4. multiplyFn(a, b) (Sync Function Call)**

* A **new Execution Context** is created for multiplyFn
* It’s pushed to the **Call Stack**
* Code runs immediately:
  + a \* b is evaluated
  + result is returned and assigned to c
* This EC is **popped** from the stack

**🔸 5. console.log(c) Runs**

* c now holds the result of the multiplication
* It is logged immediately (sync)

**🔸 6. Event Loop Takes Over**

* Now, JS has completed all sync code
* **Call Stack is empty**

**🔸 7. libuv Checks for Async Completion**

* If the API call has completed:
  + libuv places the **callback** (CB) inside the **Callback Queue**
* The **Event Loop** checks:
  + Is Call Stack empty? ✅ Yes
  + Then → move callback from queue → into the Call Stack
  + Callback executes: console.log(res?.secret)

**🔸 8. Garbage Collector Cleans Memory**

* Once all code is done
* Call stack is empty
* JS's Garbage Collector frees memory (unused variables)

**🔌 1️⃣ What is a Socket Connection?**

* A **socket** is a *continuous*, low-level **connection** between two computers.
* It allows **real-time data exchange**, like:
  + Chat apps 💬
  + Multiplayer games 🎮
  + Live stock updates 📈

🧠 You use the net module in Node.js to create raw **TCP socket** connections.

js

CopyEdit

const net = require('net');

const socket = net.createConnection({ host: 'example.com', port: 80 });

👉 This connects directly to a server and keeps the connection open until closed manually.

**🌐 2️⃣ What is an HTTP Request?**

* An **HTTP request** is a **one-time** message sent from client to server to request a web resource.
* You use it when:
  + Loading a web page
  + Fetching API data
  + Posting form data

🧠 You use the http or https module in Node.js:

js

CopyEdit

const http = require('http');

http.get('http://example.com', res => {

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

res.on('end', () => console.log("✅ Done"));

});

👉 The connection is **short-lived** — once data is received, the connection closes.