## 

## **Node.js Async Superpowers — What Happens Inside vs Outside Your Machine** 🌐🖥️

## 🔹 1. File System – fs.readFile(), fs.writeFile()

When you read or write files in Node using fs.readFile() or fs.writeFile(), everything happens **inside the machine**.

**const fs = require('fs'); // built-in module**

**// 👇 Asynchronous, non-blocking file read**

**fs.readFile('data.txt', 'utf8', (err, data) => {**

**if (err) {**

**console.error('❌ Error reading file:', err);**

**return;**

**}**

**console.log('📄 File content:', data);**

**});**

**1️⃣ V8 starts executing my JS code (main thread)**

js

CopyEdit

fs.readFile('data.txt', 'utf8', callback);

👉 This line runs **synchronously** in the main thread.  
But the function itself is **asynchronous**, so Node **doesn’t wait** for the file to be read.

**2️⃣ Node.js gives the file task to libuv (the async manager)**

libuv says:  
*“This is a file system task. I’ll send it to my thread pool — let the main thread stay free!”*

✅ So now V8 is done with this line — it moves on.

**3️⃣ libuv uses the Thread Pool (background workers)**

* Node has 4 threads by default (can be changed)
* One thread takes the job
* It uses OS-level APIs (open, read, close) to read the file from your disk

📦 This happens **outside** the JS engine

**4️⃣ Once the file is read, the result comes back**

* Thread finishes the file read ✅
* libuv now says:

“Hey Event Loop, here’s the callback — run it when the main thread is free.”

**5️⃣ The callback is added to Event Loop (Poll Phase)**

* Node checks:

“Is the Call Stack empty?”

🟢 If yes → the callback is pushed to the Call Stack  
🔁 Now V8 **executes your callback function** like:

js

CopyEdit

console.log('📄 File content:', data);

📌 **There’s no internet or external system involved.**  
All operations — JS execution, threading, OS-level file I/O — happen on your own Node.js server.

**🔹 2. HTTP — http.get(), fetch(), axios()**

**🌐 What Happens When You Make a Network Request in Node.js?**

**Example:**

js

CopyEdit

const http = require('http');

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

// handle response

});

Or

js

CopyEdit

fetch('https://example.com') // (in browser or Node with node-fetch)

**✅ What Happens Inside Your Machine**

**1️⃣ V8 Executes the JS Code**

* Your code (http.get() or fetch()) is executed inside the **V8 engine**
* V8 says: “This is an asynchronous operation. Let’s delegate it.”

**2️⃣ libuv Takes Over**

* Node.js uses libuv to manage all async operations.
* libuv prepares a **TCP socket connection** (low-level networking)
* This is done using OS features like:
  + epoll (Linux)
  + kqueue (macOS)
  + IOCP (Windows)

🧠 At this point, **no network call has gone out yet** — we're still setting things up **inside your machine**.

**3️⃣ The OS Sends the HTTP Request**

* Once the socket is ready, the actual **HTTP request is sent over the internet**
* This is the moment when your machine **crosses the network boundary**

**🔴 What Happens Outside Your Machine**

**4️⃣ Request Hits a Remote Server**

* The IP of example.com is resolved using DNS
* The request travels over:
  + Routers
  + Internet cables
  + Possibly proxies / firewalls

→ and reaches the server hosting example.com

**5️⃣ Remote Server Processes the Request**

* That server might be:
  + A Node.js API
  + A Python/Java backend
  + An NGINX reverse proxy
* It receives your request, processes it, and sends back a response

**6️⃣ Response Comes Back Into Your Machine**

* Your **operating system** receives the incoming response via the socket
* Then it's handed over to **libuv** and queued in the **Event Loop**
* When the Event Loop is ready, your callback (e.g. in res => {}) runs in the **Call Stack**

**How a Socket is Opened and Closed — Step-by-Step Breakdown (socket lifecycle)**

**✅ 1. Your JS Code Calls a Network API**

Example:

const http = require('http');

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

// handle response

});

Or

const net = require('net');

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

At this point, V8 executes the code, and Node.js delegates the async task to **libuv**.

**✅ 2. libuv Asks OS to Create a TCP Socket**

* libuv uses low-level OS APIs to create a socket
* A **file descriptor (FD)** is assigned to the socket
* The OS starts the **TCP 3-Way Handshake**

**🤝 3. TCP 3-Way Handshake (Connection Establishment)**

Client Server

────── SYN ───────▶ (Client says: "Wanna connect?")

◀──── SYN + ACK ───── (Server replies: "Sure, I’m ready")

────── ACK ───────▶ (Client confirms: "Cool, let’s go")

Arrow mark indicates who sends whom

SYN ACK are packets

✅ After this, the **socket is open** and data can now flow in both directions.

**✅ 4. Data Transmission Through the Socket**

Now that the socket is open:

* HTTP request is sent
* Remote server responds
* Data flows through the open socket (bi-directional)

In Node.js:

socket.write('GET / HTTP/1.1\r\nHost: example.com\r\n\r\n');

its like send ing a request

**🔚 5. Socket Close (Connection Termination)**

Once data exchange is done, the socket is closed either:

**a) Gracefully (Normal Case)**

→ This uses a **4-way TCP termination handshake**:

Client Server

FIN ───────▶ ("I'm done sending")

◀───── ACK

◀───── FIN ("Me too")

ACK ───────▶

**b) Forcefully**

→ Using socket.destroy() or on error/timeouts.

In Node.js:

socket.end(); // graceful close

socket.destroy(); // force close

code example –

const net = require('net'); // 📦 Import Node's built-in 'net' module for raw TCP socket communication

// 1️⃣ Create a TCP connection to example.com on port 80 (HTTP default)

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

console.log('✅ [connect] Socket connected');

// 2️⃣ Send a basic HTTP GET request to the server after connection is established

// - \r\n means line breaks (as per HTTP protocol)

// - We write directly to the TCP stream (not using HTTP module here)

socket.write('GET / HTTP/1.1\r\nHost: example.com\r\n\r\n');

// 🔥 This sends the raw HTTP request manually over the socket

});

let fullData = ''; // Used to collect all chunks of response data

// 3️⃣ Triggered every time data (chunk) is received from the server

socket.on('data', chunk => {

console.log('📥 [data] Received chunk:', chunk.toString());

fullData += chunk.toString(); // Accumulate chunks to build full response

});

// 4️⃣ Triggered when the server ends the connection (half-close)

// Means: server says “I’m done sending”

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

console.log('🔚 [end] Connection closed by server');

console.log('📦 [result] Full response:', fullData); // Print the full assembled response

});

// 5️⃣ Triggered when the socket is fully closed (both sides confirm)

// Can also include if it was closed due to an error

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

console.log(`❎ [close] Socket closed${hadError ? ' due to error' : ''}`);

});

// 6️⃣ If there's any error (DNS fail, timeout, etc.) this runs

socket.on('error', err => {

console.error('❌ [error] Socket error:', err);

});

// 7️⃣ Optional: You can also close the socket manually like this:

// socket.end(); // This sends FIN to server, initiating graceful shutdown

# ❓ When Is socket.on('close') Triggered in Node.js?

### ✅ Simple Answer:

close is triggered **after the socket is fully closed** — regardless of whether it ended **normally** or because of an **error**.

# 🧠 How Node.js Knows It's Time to Close a Socket

## ✅ Short Answer:

👉 The **Operating System (OS)** tells Node.js (via libuv) that:

“The remote party (server/client) has ended the connection.”

Node does **not guess** — it gets an event from the OS.

**🧠 How Does the OS Know It’s Time to Close a Socket?**

**✅ Short Answer:**

The **TCP protocol inside your OS** has a built-in **state machine** that follows strict rules.  
When it receives a **FIN** or **RST** packet from the remote system, it knows:

“Okay, the remote side is done — I should start closing the socket.”

**🔁 Full Step-by-Step Breakdown**

**⚙️ Every OS (Linux, Windows, macOS) runs a TCP stack at the kernel level.**

When a socket is opened:

yaml

CopyEdit

Socket State: ESTABLISHED

**🧩 Then, when one party sends a FIN (Finish) signal:**

* It means: “I’m done sending data, you can close the connection from your side now.”
* The OS receives this packet, and moves the socket to:

makefile

CopyEdit

State: CLOSE\_WAIT

**🧭 TCP State Machine (Simplified)**

Let’s walk through a **normal 4-step TCP shutdown**:

arduino

CopyEdit

Client Server

────── FIN ───────────────▶ // Client is done sending

◀───── ACK // Server acknowledges

◀───── FIN // Server also done

────── ACK ───────────────▶ // Client says cool, I’m done

Now both sides move to:

makefile

CopyEdit

State: CLOSED

And **OS closes the socket descriptor** 🧨

**🔍 So in a sentence:**

Your OS closes the socket only after receiving a **FIN or RST** TCP packet from the remote system, and confirming both sides are done — as per the official TCP state machine.

## 🔐 Special Cases:

| **Case** | **What Happens?** |
| --- | --- |
| 🟢 Remote sends FIN | OS goes to CLOSE\_WAIT, then CLOSED |
| ❌ Remote crashes (sends RST) | OS forcefully shuts socket |
| 🕒 Timeout (no FIN received) | OS can auto-close after a period |
| 👨‍💻 You call destroy() | OS instantly kills the socket (RST) |

**🧠 How Node.js Handles Multiple User Requests Without Creating a Thread per Request**

**🚫 The Old Way (Traditional Thread-Per-Request Model):**

In languages like Java or PHP:

* Every user request creates a new thread 🧵
* If 10,000 users connect → 10,000 threads → 😱 massive memory usage, CPU load

**✅ Node.js (and libuv) does it differently:**

**Node.js uses a single thread with an Event Loop + OS-level magic (like epoll, kqueue) to handle thousands of users without blocking.**

**🔧 How it really works:**

**1️⃣ libuv sets up a socket per connection**

Every user hitting the server gets a socket 🔌 — a unique communication pipe

**2️⃣ These sockets are passed to the OS’s kernel-level event notification system:**

* **Linux:** epoll
* **macOS:** kqueue

These are super smart 👮‍♂️ tools provided by the OS

**3️⃣ These tools watch all sockets at once (non-blocking)**

If any socket gets activity (new request/data)  
→ They **notify libuv**: “⚠️ There’s activity on socket #57!”

✅ This means Node doesn’t sit and wait — it only reacts **when needed**

**4️⃣ Node’s Event Loop picks up the task and runs your callback**

So even if **1 million users** connect:

* Node doesn’t create 1M threads ❌
* Instead, it uses **one thread** with **OS-powered monitoring** ✅

**👀 Visual Recap (from your image):**

👮‍♂️ epoll (the police) says:

“Hey Node, there’s activity on a socket!”

🎩 Node (libuv) says:

“Thanks! I’ll take it from here.”

This is how it handles **high concurrency** 🔥

**🧠 Final One-Liner (Interview Ready):**

“Node.js uses OS-level tools like epoll to watch thousands of sockets at once without threads. libuv listens for activity, and the Event Loop processes it — all in a single thread.”

Net Vs http -  
  
const net = require('net');

// Step 1: Create raw TCP connection

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

console.log('✅ [net] Connected');

// Step 2: Manually send raw HTTP request (text)

socket.write('GET / HTTP/1.1\r\nHost: example.com\r\n\r\n');

});

let fullData = '';

socket.on('data', chunk => {

// Step 3: Collect raw response in parts

Console.log(chunk.toString())

fullData += chunk.toString();

});

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

console.log('🔚 [net] Connection closed');

console.log('📦 [net] Full Response:\n', fullData);

});

Output chunk –

### 🧾 What You’ll See in Output (Sample):

php-template

CopyEdit

📥 [net] Raw chunk:

HTTP/1.1 200 OK

Content-Type: text/html; charset=UTF-8

Content-Length: 1256

Connection: close

<html>

<head><title>Example Domain</title></head>

<body>...</body>

</html>

➡️ It’s 100% **raw text**, you have to **manually split** headers and body like this:

js

CopyEdit

const [rawHeaders, html] = fullData.split('\r\n\r\n');

console.log('🔍 Headers:\n', rawHeaders);

console.log('🧾 Body:\n', html);

code Explanation

We use net.createConnection() to open a raw TCP socket to port 80 of example.com — this is just a pipe, it doesn’t understand HTTP.

We then manually write a raw HTTP GET request string, like how browsers speak to servers — this is just a line of text, including headers like Host.

The server sends back an HTTP response (status, headers, HTML body), but it comes in raw text — not parsed.

We collect the response in fullData as chunks. When the connection ends, we log the entire raw response.

📌 Result: You get everything (headers + HTML) jumbled in one string. If you want just the body, you have to manually split the response using .split('\r\n\r\n').

### ❌ Problems:

* You do **everything yourself**
* No HTTPS support
* No JSON parsing, no status code access
* Tedious and error-prone

**🌐 Case 2: Using https — High-level HTTP Client**

**✅ Code:**

const https = require('https');

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

console.log('✅ Status:', res.statusCode);

console.log('📋 Headers:', res.headers); // Already parsed!

let body = '';

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

console.log('📥 [https] Body chunk:\n', chunk.toString());

body += chunk.toString();

});

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

console.log('🔚 Full Parsed Body:\n', body);

});

});

### 🧾 What You’ll See in Output (Sample):

javascript

CopyEdit

✅ Status: 200

📋 Headers: {

'content-type': 'text/html; charset=UTF-8',

'content-length': '1256',

connection: 'close',

...

}

📥 [https] Body chunk:

<html>

<head><title>Example Domain</title></head>

<body>...</body>

</html>

**🧠 Explanation (Plain Language):**

1. We use https.get() — Node.js internally sets up a **secure TLS connection**, builds the HTTP request for you, and parses everything.
2. You get the response object res with(parced header object)
   * res.statusCode: already parsed
   * res.headers: already parsed
   * Response body: cleanly streamed as chunks
3. You just collect the body and print it. No need to parse or split strings. It handles:
   * Headers
   * Redirection
   * Chunking
   * Encoding
   * HTTPS handshake

📌 Result: Clean, structured, and **developer-friendly** output.

**🔥 Final Understanding:**

 net: You see **the raw HTTP protocol** (HTTP/1.1, headers, and body as a long string)

 https: You get **only the body chunks**, and all metadata (status, headers) is **already parsed into structured objects**

* net is raw and powerful, but **you must implement everything** — from HTTP headers to chunk parsing.
* https is built on top of net, giving you **automatic TLS**, **parsed headers**, and an easy-to-use interface.

🧠 So unless you're building a custom protocol (SMTP, Redis, FTP), always use https for secure HTTP requests.

TLS -

**TLS** stands for **Transport Layer Security**.  
It’s the **protocol** that **secures all data** sent between two machines — like your browser and a server — by **encrypting** it.

**🌐 1️⃣ https → Automatic TLS (Secure)**

js

CopyEdit

const https = require('https');

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

console.log('✅ Status:', res.statusCode);

console.log('📋 Headers:', res.headers);

});

**🔍 What tells you this is TLS (encrypted)?**

* https module: the **'s'** in https stands for **Secure**
* It **automatically**:
  + Uses **TLS** to encrypt traffic
  + Verifies the server’s certificate
* Port used: **443** (standard for HTTPS)

✔️ **TLS is automatically applied** — you can’t even access the raw socket unless you dig deep with tls.connect().

**🧱 2️⃣ net → Raw TCP (No TLS)**

js

CopyEdit

const net = require('net');

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

console.log('✅ [net] Connected');

// Manually send raw HTTP string

socket.write('GET / HTTP/1.1\r\nHost: example.com\r\n\r\n');

});

**🔍 What proves this is not TLS?**

* Using net module = **plain TCP**
* Port used: **80** (default for unencrypted HTTP)
* You manually write raw HTTP requests
* No automatic encryption, no certificate checks

✔️ **No TLS involved**, everything goes over the network as plain text.

**If 1000 Socket Activities Are Detected at Once — How Does Node Handle It?**

**✅ Step-by-Step Internal Breakdown:**

**1️⃣ 1000 sockets are open (users connected)**

Suddenly all 1000 sockets get activity (maybe 1000 users sent a message at once) 🚀

**2️⃣ OS (epoll, kqueue) Detects Activity**

* epoll (Linux) or kqueue (macOS) is **monitoring all 1000 sockets**
* It doesn't get overwhelmed — it's made for this! 💪
* It detects: “Socket #1, #2, ..., #1000 have incoming data!”

🧠 These are **I/O readiness notifications** - **not actual data processing**

**3️⃣. OS Notifies libuv in One Batch.**

Hey libuv, these 1000 sockets are readable!

✅ libuv gets this batch and queues the callbacks associated with each socket. Stored in macritask queue

**4️⃣. libuv Queues All 1000 Events into the Poll Phase.**

**Queue -**

1. socket1.on('data', ...)

2. socket2.on('data', ...)

3. socket3.on('data', ...)

...

1000. socket1000.on('data', ...)

Each one is scheduled, but not executed immediately.

**5️⃣. Event-Loop Starts Processing ONE BY ONE**

Even though all 1000 are ready - Node.js is **single-threaded**  
So it handles one event (callback) at a time.

**🔁 Process -**

▶ socket1.on('data') → runs

▶ socket2.on('data') → runs next

▶ ...

▶ socket1000.on('data') → runs last

**❗ Important:**

* These 1000 events are not executed **in parallel**
* But they are **not lost or skipped** — they are **queued and processed** in the event loop ✅

**💡 What If Processing Each Takes Time?**

If a callback is heavy (e.g., has a for loop or blocking CPU task), it slows everything down.

✅ That’s why we write **non-blocking**, fast callbacks — to keep the loop moving.

**🔹 3. Timers – setTimeout(), setInterval()**

These are internal mechanisms to delay or repeat execution.

When you do:

js

CopyEdit

setTimeout(() => console.log("Done"), 1000)

**✅ Everything happens inside:**

* setTimeout is handled by libuv’s **timer manager**
* It keeps track of the time and waits internally
* After 1 second, libuv places the callback into the callback queue
* The event loop picks it up and runs it in JS (V8)

📌 **No external system or OS call is needed beyond basic timekeeping.**  
So timers are fully handled inside the Node.js machine.

**🔹 4. DNS – dns.lookup() vs dns.resolve()**

Node provides two ways to resolve domains:

:

## 🔹 dns.lookup()

"Asks the **OS** for the IP — works like a browser."

## 🔹 dns.resolve()

"Asks the **DNS server** directly — gets full DNS records."

🎯 One-liner to remember:

"lookup = OS-level, resolve = internet-level"

💡 Use lookup to **connect**,  
💡 Use resolve to **inspect DNS**.

**🔹 5. Crypto – crypto.pbkdf2(), crypto.randomBytes()**

When you use Node’s crypto functions to hash passwords or generate secure tokens:

crypto.pbkdf2('secret', 'salt', 10000, 64, 'sha512', cb);

**✅ Everything happens inside:**

* libuv thread pool handles this operation
* It's CPU-heavy, so it’s done in the background without blocking JS
* The crypto logic and computation happen **inside the server**, not via the internet

📌 No data leaves the machine. No external system is involved.

**🔐 crypto.randomBytes() in Node.js**

**✅ What it does:**

crypto.randomBytes() generates **cryptographically strong random data**.

That means it’s **secure, unpredictable**, and suitable for things like:

* Generating tokens
* Session IDs
* Password reset links
* Unique salts for password hashing

**🧪 Example: Generate Random Bytes**

js

CopyEdit

const crypto = require('crypto');

const buffer = crypto.randomBytes(16); // 16 bytes

console.log(buffer);

console.log(buffer.toString('hex')); // Convert to hex string (32 characters)

**🧠 Output:**

php-template

CopyEdit

<Buffer 93 4a 85 c2 ...>

934a85c27cbe5c1fc34bb2e17b36499e

**🔍 How it Works (Internally):**

* crypto.randomBytes(size) asks the **OS** to give secure random bytes
* It uses system-level APIs:
  + On Linux/macOS: /dev/urandom
  + On Windows: CryptGenRandom
* Node.js wraps these with libuv and gives you the result as a Buffer

## ⚠️ Why Not Math.random()?

| **Math.random()** | **crypto.randomBytes()** |
| --- | --- |
| ❌ Not secure | ✅ Cryptographically secure |
| ✅ Fast, lightweight | ❌ Slightly heavier (uses system) |
| ❌ Predictable in some cases | ✅ Truly random |

Use Math.random() for animations/games  
Use crypto.randomBytes() for **security-related things**

## ✨ Real Use Case — Generate Token

js

CopyEdit

const token = crypto.randomBytes(32).toString('hex'); // 64-char token

console.log('Reset token:', token);

## 🧠 Interview One-Liner:

"crypto.randomBytes() gives you secure, unpredictable random data by asking the OS. It’s used for tokens, salts, and secure IDs — not for math or games."

# 🧂 What is a Salt in Password Hashing?

## 🧠 The Problem:

If two users have the **same password**, their hash will be **identical**.

js

CopyEdit

hash('password123') => 'abf8e...'

So if a hacker gets access to the database, they can **guess** users with same passwords, or use **precomputed tables** (rainbow tables) to reverse the hash.

## ✅ The Solution: Add a ****Salt****

A **salt** is a random string added to the password **before hashing**.

js

CopyEdit

const password = 'password123';

const salt = 'a2f93b4c5e8d';

const saltedPassword = password + salt;

hash(saltedPassword);

Now every password — even if same — will have **different hashes** because the salt is different 🔐

## 📌 Where does crypto.randomBytes() come in?

You use it to **generate the salt**:

js

CopyEdit

const crypto = require('crypto');

const salt = crypto.randomBytes(16).toString('hex'); // generate 16-byte salt

Then combine with password before hashing:

js

CopyEdit

const password = 'password123';

const salted = password + salt;

const hash = crypto.createHash('sha256').update(salted).digest('hex');

# 🔐 1️⃣ const salt = crypto.randomBytes(16).toString('hex');

### ✅ What this does:

It **generates a unique random salt** — which is just a string of unpredictable characters used for hashing.

### 🔍 Breakdown:

#### 🧱 crypto.randomBytes(16)

* This asks Node.js to generate **16 bytes** of **cryptographically secure random data**
* It uses OS-level random generators (/dev/urandom or CryptGenRandom)
* Returns a **Buffer object** like:

php-template

CopyEdit

<Buffer 49 7a bc 5e a1 ...>

#### 🔁 .toString('hex')

* Converts that Buffer into a **hexadecimal string**
* Each byte = 2 hex characters → 16 bytes = **32 characters**

Example:

js

CopyEdit

'a3b9c25e9f1d6f8e7c0a21b4f8d1ab2e'

✅ Final salt is a **unique 32-character hex string**

## 🧠 Why it’s used:

To add randomness to password hashing, making every hash unique — even for the same password!

# 🔐 2️⃣ const hash = crypto.createHash('sha256').update(salted).digest('hex');

This line creates the **final hashed password** from your salted input using the **SHA-256 hashing algorithm**.

### 🔍 Breakdown:

#### ⚙️ crypto.createHash('sha256')

* Initializes a new **hash object** using the SHA-256 algorithm
* SHA-256 is a 256-bit secure hash algorithm
* 🔒 Used for digital signatures, authentication, etc.

#### 🔁 .update(salted)

* Feeds the input string to the hash function
* Here, salted = password + salt
* This is what you're actually hashing

Example:

js

CopyEdit

const salted = 'password123' + 'a3b9c25e9f1d6f8e7c0a21b4f8d1ab2e';

#### 📦 .digest('hex')

* Finalizes the hashing process
* Converts the result (binary) into a **hex string**
* This is your final stored password hash

Example output:

bash

CopyEdit

'b7e2c0f598748f1577e4a5c04f0a0d9dc9b2edc2a02c0f4a1a3b145ccf2e3f7b'

**🤔 Why convert the hashed value from binary to hex before storing?**

**⚠️ First, what is a hash?**

When we hash something (like a password), we get **binary data** (raw bytes), not readable text.

Example:

js

CopyEdit

const hash = crypto.createHash('sha256').update('someInput').digest(); // No 'hex'

console.log(hash);

You’d get:

cpp

CopyEdit

<Buffer 68 23 af 11 d2 ...> // raw binary

**🧠 So why convert it to 'hex' or 'base64'?**

**✅ Reason #1: Storage Compatibility (Databases, Filesystems, JSON)**

* Most databases (like MySQL, MongoDB, Postgres) are **text-optimized**
* Raw binary data (Buffer) may:
  + Break encoding (e.g., UTF-8)
  + Cause corruption
  + Require special handling (BLOB fields, base64 conversions)
* Text formats (like JSON, log files) **can’t safely store binary**

✅ So we convert binary → **hex** or **base64**, both are safe strings.

**✅ Reason #2: Readability and Debugging**

* A hex string is human-readable (e.g. af1c23a...)
* You can easily log or copy-paste hashes
* Makes debugging or token comparisons easy

**✅ Reason #3: Consistency and Portability**

* Hex strings are **always the same length** for a given hash algorithm:
  + SHA-256 → 256 bits → 64 hex characters
* Easy to compare across systems, databases, platforms

**🔐 What if Two Users Have the Same Password?**

**⚠️ Problem Without Salt:**

Let’s say we store passwords like this:

js

CopyEdit

// No salt — just hash the password

hash('password123'); // SHA256 or bcrypt or anything

If two users use 'password123', they’ll have:

css

CopyEdit

User A: hash('password123') => abc123...

User B: hash('password123') => abc123... ❗ Same!

🔓 A hacker can guess the hash means 'password123'  
❌ Bad practice — vulnerable to **rainbow table** attacks.

**✅ Solution: Use a Unique Salt for Each User**

**1️⃣ When user registers:**

js

CopyEdit

const password = 'password123';

const salt = crypto.randomBytes(16).toString('hex'); // Different for each user

const hash = sha256(password + salt);

✅ Now even if both users have same password — their **hashes will be different!**

**2️⃣ Example:**

js

CopyEdit

// User A

password: 'password123'

salt: 'a1b2c3'

hash: sha256('password123a1b2c3') => abc123...

// User B

password: 'password123'

salt: 'x9y8z7'

hash: sha256('password123x9y8z7') => 9f8d6a...

✅ Result: **Completely different hashes** in DB  
❌ Hacker can't tell that both used same password

**🧠 How It Works During Login:**

When user logs in:

1. You fetch the **user’s salt from DB**
2. Recreate the hash with their entered password + stored salt
3. Compare result with stored hash

js

CopyEdit

if (hash(enteredPassword + storedSalt) === storedHash) {

✅ Login success

}

**🎯 Interview One-Liner:**

“We use a unique salt for each user, so even if two users have the same password, their final hashes are different. This prevents hackers from knowing who used the same password or using rainbow tables to crack them.”

**❓If Hashes Can’t Be Decrypted…**

**…What Happens When a User Forgets Their Password?**

**🧠 Important Truth:**

**Yes, hashes are one-way and irreversible**  
👉 So we cannot "recover" or "decrypt" the original password  
❌ There is **no way** to get the user's password back from the stored hash

**✅ So What Do Real Systems Do?**

Instead of recovering the password,  
**we help the user reset it**

**🔁 Password Reset Flow (Not Recovery):**

Here’s how big apps handle it securely:

**🔹 Step 1: User clicks “Forgot Password?”**

**🔹 Step 2: Backend generates a secure, one-time token**

js

CopyEdit

const resetToken = crypto.randomBytes(32).toString('hex');

✅ This token is:

* Unique
* Random
* Has a short expiration time (e.g. 10–15 min)

**🔹 Step 3: Token is sent to the user’s email as a link**

txt

CopyEdit

Click to reset your password:

https://example.com/reset-password?token=ab1c23...

**🔹 Step 4: User clicks the link, sets a new password**

You hash the **new password + new salt**, and update the DB:

js

CopyEdit

const newSalt = crypto.randomBytes(16).toString('hex');

const newHash = hash(newPassword + newSalt);

✅ No recovery needed — you’re replacing the old hash.

**⚠️ Problem:**

**What if a hacker steals the reset token and the user's new password during a password reset?**

**🔥 What can go wrong?**

**🧨 Case 1: Hacker steals the reset token**

* They can **open the reset link** and change the password
* Take over the account

**🧨 Case 2: Hacker uses a keylogger or fake site**

* Sees what password the user types
* Also sees the **new password + reset token**

**😱 If they have both:**

* The hacker now knows the new password
* And the reset token
* ✅ They can **hijack the account easily**

**✅ Solution (What Real Apps Do to Prevent This)**

**🔒 1. Always use HTTPS**

* Prevents token and password from being seen during network transfer

**🔐 2. Store reset tokens in hashed form**

js

CopyEdit

const hashedToken = hash(token); // Store in DB

* So even if hacker sees token, it’s useless without matching hash

**⏳ 3. Expire tokens quickly**

* Valid for 10–15 minutes only
* Only works **once** per user

**🚨 4. Alert the user**

* Email/SMS: “Your password was just reset. Was this you?”

**🛡️ 5. Add 2FA (two-factor login)**

* Even if password is stolen, hacker can't log in without OTP

**🧠 Final Summary for Interviews**

“If a hacker steals both the reset token and new password, they can hijack the account. But this is solved by using HTTPS, hashing tokens, expiring them fast, alerting users, and enabling 2FA. The problem isn’t with hashing — it’s with token misuse or user compromise.”

**🧠 crypto.randomBytes() vs crypto.pbkdf2() — Node.js Crypto APIs**

**✅ 1️⃣ crypto.randomBytes(size)**

➡️ **Used to generate random data (like salts or tokens)**

**🔧 Purpose:**

To generate **unpredictable**, cryptographically secure **random bytes**

js

CopyEdit

const crypto = require('crypto');

const salt = crypto.randomBytes(16); // 16 bytes = 128 bits

console.log(salt.toString('hex'));

**📦 Output:**

* A Buffer of random bytes
* Commonly used for:
  + Salts (for password hashing)
  + Token generation (e.g. reset links)
  + Nonces

**✅ 2️⃣ crypto.pbkdf2(password, salt, iterations, keylen, digest, callback)**

➡️ **Used to hash passwords securely**

**🔧 Purpose:**

To **derive a secure key** from a password using:

* Salt
* Iterations
* Digest algorithm (e.g., sha512)

js

CopyEdit

crypto.pbkdf2('password123', salt, 100000, 64, 'sha512', (err, derivedKey) => {

console.log(derivedKey.toString('hex'));

});

**🔥 Why it’s powerful:**

* Slows down brute-force attacks
* **More secure than plain hashing** (like sha256(password))
* Built-in defense: salt + stretching

**❓ Can We Hash Without pbkdf2()?**

**✅ YES — You can use simple hash functions like:**

js

CopyEdit

const crypto = require('crypto');

const password = 'password123';

const hash = crypto.createHash('sha256')

.update(password)

.digest('hex');

console.log('Hash:', hash);

🔍 This gives you a hash of the password — BUT…

**⚠️ Why It’s NOT Secure for Passwords**

Hashing passwords directly with sha256, md5, or sha1 is **NOT recommended** ❌

Here’s why 👇

**❌ Problem #1: Too Fast**

* SHA-256 is **blazing fast** — which is great for general hashing (like files)
* But for passwords, that’s dangerous ⚠️  
  ➤ Hackers can brute-force millions of guesses per second

**❌ Problem #2: No Iterations or Key Stretching**

* pbkdf2() repeats hashing thousands of times (e.g. 100,000)
* This makes brute-forcing **very slow**
* Simple sha256() does it just **once** — easy to crack

**❌ Problem #3: No Built-In Salt**

* crypto.createHash() doesn’t support salting by default
* You have to **manually add salt**
* Still doesn’t slow attackers down

**✅ Why pbkdf2() is Better**

* Adds a **salt**
* Repeats hashing **many times** (slow by design)
* Makes brute-force attacks **practically useless**

**🔹 6. External APIs – Using fetch(), axios(), or any HTTP client**

If your code calls an external service like this:

fetch('https://api.github.com/users')

Then:

**✅ The setup and socket creation happens inside your machine**

But…

**🔴 The actual HTTP request travels across the internet to another machine**

And the response is sent back from outside → to your machine

📌 This is similar to http.get()

**🧠 Final Summary (in Words)**

* **File system, timers, crypto, thread pool** operations → ✅ happen **entirely inside** your Node.js server
* **HTTP and API requests** → ❌ go **outside** to talk to remote servers
* **DNS** → Can be ✅ inside (if using lookup) or ❌ outside (if using resolve)

**Node.js File System Operations (fs module)**

**1️⃣ fs.readFile() — Asynchronous File Read**

fs.readFile('file.txt', 'utf8', (err, data) => {

if (err) throw err;

console.log(data);

});

**🔧 Internal Flow:**

1. V8 runs the code.
2. fs.readFile() is passed to **libuv**.
3. libuv assigns task to **Thread Pool**.
4. Thread uses **OS** to read the file.
5. Once done, result is sent back.
6. libuv pushes callback to **Event Loop (Poll Phase)**.
7. Callback runs in **main thread (V8)**.

✅ Non-blocking — continues executing other code.

**2️⃣ fs.readFileSync() — Synchronous File Read**

const data = fs.readFileSync('file.txt', 'utf8');

console.log(data);

**🔧 Internal Flow:**

1. V8 calls fs.readFileSync()
2. The thread **blocks** until OS reads the file.
3. File data is returned immediately.

❌ Blocking — halts execution until file is read.

**3️⃣ fs.writeFile() — Asynchronous File Write**

fs.writeFile('output.txt', 'Hello, world!', err => {

if (err) throw err;

console.log('✅ File written!');

});

**🔧 Internal Flow:**

* Similar to readFile()
* Handled by libuv → Thread Pool → OS → Callback

✅ Doesn’t block main thread.

**4️⃣ fs.writeFileSync() — Synchronous File Write**

fs.writeFileSync('output.txt', 'Hello, world!');

console.log('✅ File written!');

**🔧 Internal Flow:**

* V8 runs and **waits** until file is written.

❌ Blocking.

**5️⃣ fs.appendFile() / fs.appendFileSync() — Add Content to File**

fs.appendFile('log.txt', 'New log\n', err => {...});

fs.appendFileSync('log.txt', 'New log\n');

* Same flow as write (async vs sync difference)

**6️⃣ fs.unlink() / fs.unlinkSync() — Delete a File**

fs.unlink('temp.txt', err => {...});

fs.unlinkSync('temp.txt');

* Removes a file
* Async uses Thread Pool; Sync blocks.

**7️⃣ fs.rename() / fs.renameSync() — Rename or Move File**

fs.rename('old.txt', 'new.txt', err => {...});

fs.renameSync('old.txt', 'new.txt');

## 📝 fs.writeFile() Overwrites the File

js

CopyEdit

fs.writeFile('output.txt', 'New content', err => {

if (err) throw err;

console.log('✅ File written');

});

### ⚠️ Important Behavior:

* If the file **already exists**, fs.writeFile() will **completely overwrite** its content with the new data.
* It does **not append** — it **replaces** the content entirely.