**Advance Questions**

**List of Questions**

1. **Explain the Node.js event loop and its phases. How does it handle asynchronous operations?**
2. **What is the difference between process.nextTick(), setImmediate(), and setTimeout() in Node.js?**
3. **How does Node.js handle concurrency given it is single-threaded?**
4. **What are streams in Node.js? Explain the types of streams and how you would use them.(Already done in previous notes)**
5. **What are the differences between callbacks, promises, and async/await in Node.js? When should you use each?**
6. **Explain how you would manage error handling in asynchronous Node.js code.**
7. **How does the Node.js module system work? What’s the difference between CommonJS and ES Modules?**
8. **What are buffers in Node.js? When and why are they used?( Already done in previous notes)**
9. **Explain how you would create a simple HTTP server in Node.js and handle routing manually without any frameworks.**
10. **How do you handle child processes in Node.js? What are use cases for spawning child processes?**
11. **What is the purpose of the cluster module in Node.js? How do you implement multi-core utilization?**
12. **How does the V8 engine optimize JavaScript code execution in Node.js?**
13. **What are some best practices for securing a Node.js application?**
14. **How can you debug a Node.js application? What tools or methods do you prefer?**
15. **What is middleware in the context of Node.js frameworks like Express? How would you write custom middleware?**

**How the Event Loop Works?**

When a Node.js application runs, the event loop starts, processes the synchronous code first, and then moves to handle asynchronous tasks. The execution follows these steps:

**1. Initialization**

When Node.js starts, it loads the script, executes synchronous code, and registers any asynchronous tasks (e.g., timers, I/O requests, network operations).

**2. Execution of Input Script**

* The call stack executes synchronous code first.
* Any asynchronous operations (setTimeout, fs.readFile, network requests) are delegated to libuv.

**3. Handling Asynchronous Operations with libuv**

Node.js uses a special C library called libuv to handle asynchronous operations. This library manages a thread pool that offloads heavy tasks (like file I/O, database operations, or network requests) that would otherwise block the event loop. The thread pool contains several threads that perform tasks like:

* File system I/O (fs.readFile)
* Network requests (HTTP, TCP, DNS)
* Timers (setTimeout, setInterval)
* Compression and cryptographic tasks

**4. Callback Execution**

Once the thread pool completes its tasks, it sends callbacks to the event queue. The event loop processes these callbacks, but only when the call stack is empty (i.e., when no synchronous code is currently executing)

**5. Event Loop Phases**

The event loop goes through multiple phases, each designed to handle a different set of operations. It checks for events, handles asynchronous callbacks, and executes tasks in the correct order.

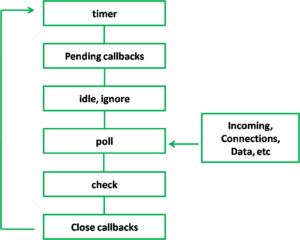
**6. Callback Execution from Event Queue**

After the call stack is empty, the event loop picks tasks from the event queue and sends them to the call stack for execution. These tasks could include:

* Completing network requests
* Processing I/O events
* Handling timers like setTimeout or setInterval

The following diagram is a proper representation of the event loop in a Node.js server:

**Q1. What are the phases of the Node.js event loop? What happens in each phase?**



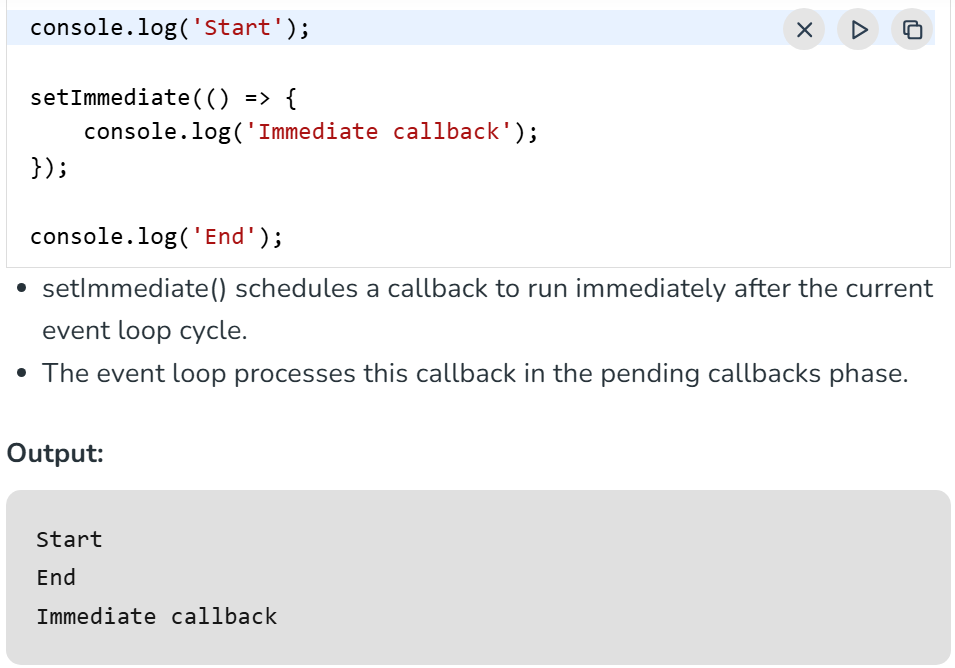
**1. Timers Phase**

This phase processes timers that have been set using setTimeout() and setInterval().



**2. Pending Callbacks**

This phase executes I/O-related callbacks that were deferred from the previous loop cycle.

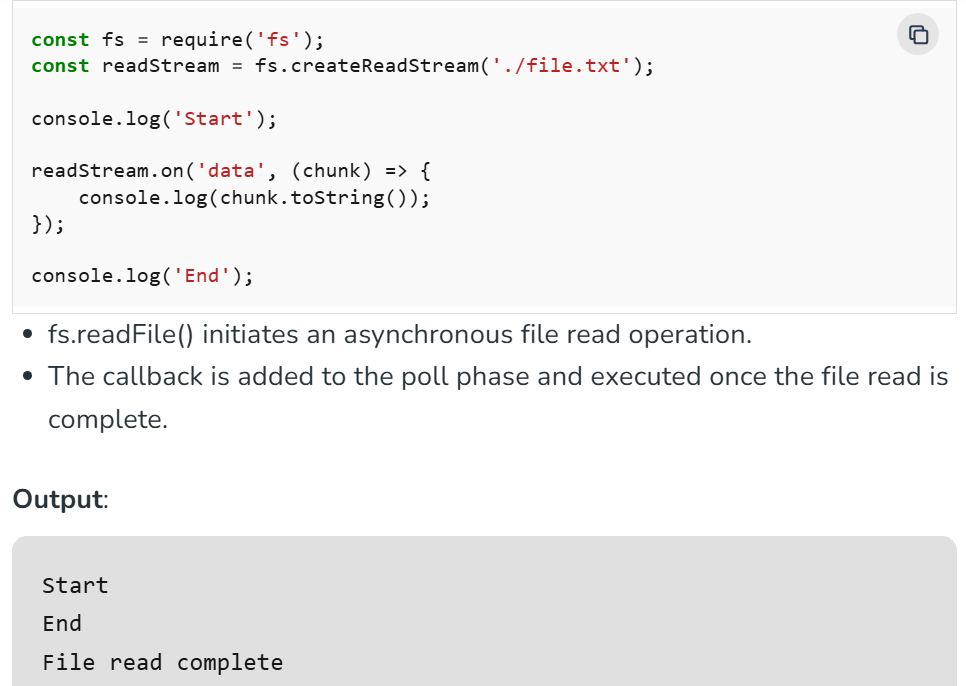


**3. Idle, Prepare** (Internal use only)

This phase is used internally by Node.js for background tasks.

**4. Poll Phase (Main Phase)**

The Poll phase executes most of the tasks like- I/O, file reading, HTTP requests and much more.

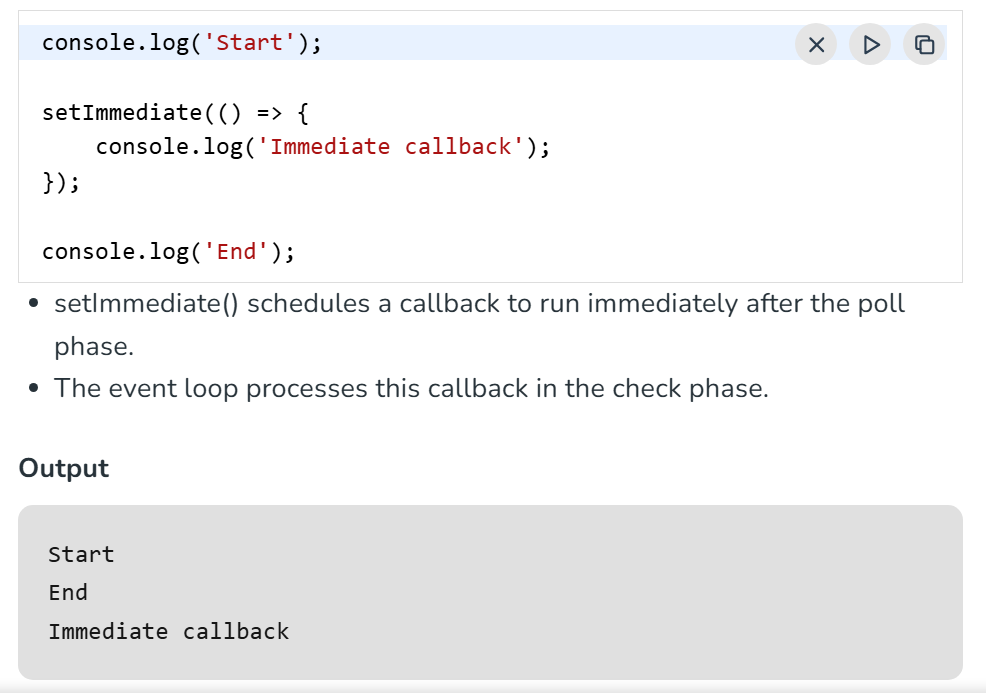


**5. Check Phase**

This phase allows the event loop to execute callbacks immediately after the **poll** phase has completed. If the **poll** phase becomes idle and scripts have been queued with **setImmediate()**, the event loop may continue to the **check** phase rather than waiting.

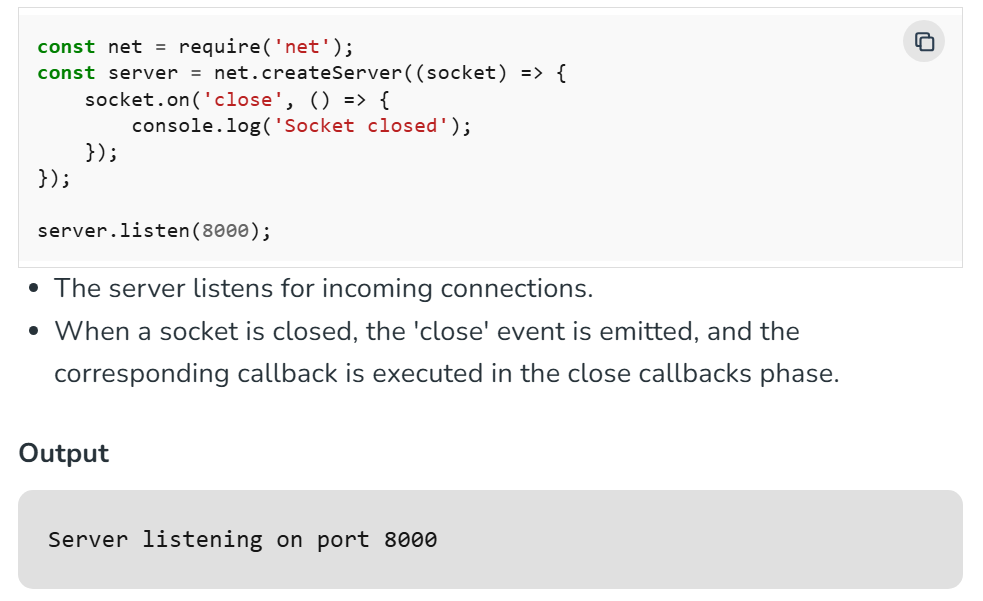
**setImmediate()** is actually a special timer that runs in a separate phase of the event loop. It uses a libuv API that schedules callbacks to execute after the **poll** phase has completed.

Generally, as the code is executed, the event loop will eventually hit the **poll** phase where it will wait for an incoming connection, request, etc. However, if a callback has been scheduled with **setImmediate()** and the **poll** phase becomes idle, it will end and continue to the **check** phase rather than waiting for **poll** events.



6. **Close Callbacks Phase**

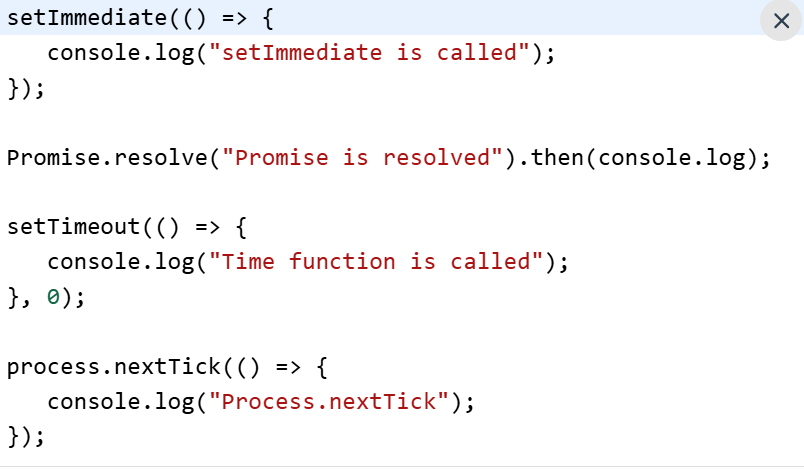
This phase executes callbacks for closed connections like sockets, streams, and event emitters.

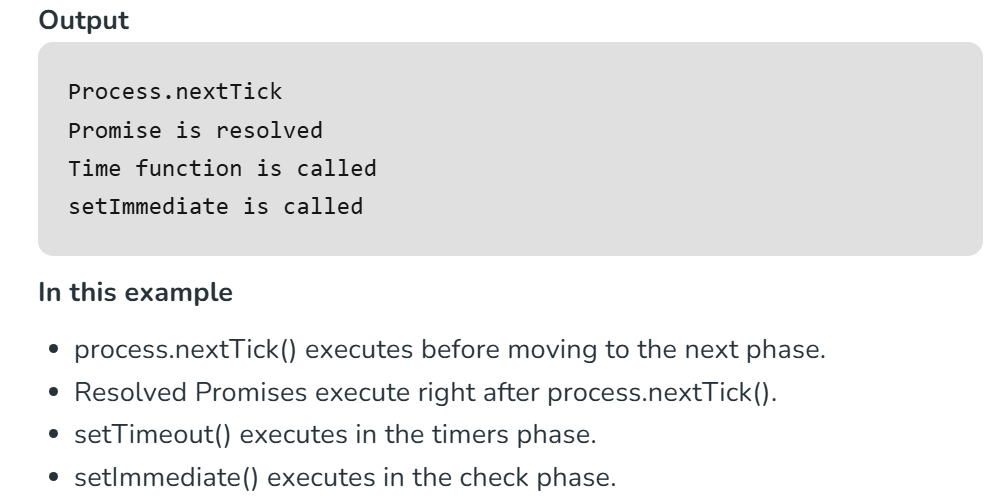


**process.nextTick() and Promises in the Event Loop**

Apart from these phases there is also process.nextTick() and promise callback which has the highest priority in the event loop. It executes after every phase before moving to the next phase.

* process.nextTick() callbacks are always executed before the event loop moves to the next phase.
* Resolved Promise callbacks are processed immediately after process.nextTick().





**Step-by-step: How Node.js Handles Asynchronous Operations**

1. **You call an async function**  
   Example: fs.readFile('file.txt', callback) or setTimeout(callback, 1000).
2. **Node.js sends the operation off**  
   The operation is handed over to:
   * The **libuv thread pool** (for file I/O, DNS, etc.), or
   * The **system APIs** (for network requests, timers).
3. **Node.js continues running other code**  
   It does **not block** or wait for the operation to finish.
4. **Async operation runs in the background**  
   The system or thread pool works on the task independently.
5. **Operation completes**  
   When done, the result or event is **queued** as a callback inside the **event loop**.
6. **Event loop picks up the callback**  
   During its phases, the event loop checks if the callback is ready to run.
7. **Callback is executed on the main thread**  
   Node.js runs the callback function, using the result of the async operation.

**Q2. What is the difference between process.nextTick(), setImmediate(), and setTimeout() in Node.js?**

| **Function** | **When It Runs** | **Timing Compared to Others** | **Typical Use Case** |
| --- | --- | --- | --- |
| process.nextTick() | Immediately after current op ends | **Before** I/O, timers, setImmediate | Run ASAP, before the event loop continues |
| setImmediate() | Check phase, after poll phase | After I/O callbacks, **after nextTick** | Run after I/O events, next tick has passed |
| setTimeout() | Timers phase, after specified delay | After nextTick & setImmediate (if delay=0) | Run after a delay |

**Q** **3. How does the Node.js module system work? What’s the difference between CommonJS and ES Modules?**

**How the Node.js Module System Works**

* Node.js uses a **module system** to organize and reuse code.
* Each file in Node.js is treated as a **separate module** with its own scope.
* Modules can **export** variables, functions, or objects and **import** them in other files.
* Node.js **loads** modules using the require() function (CommonJS) or import statement (ES Modules).
* When you require() or import a module, Node.js locates it, loads it, executes it once, and caches it for subsequent uses.

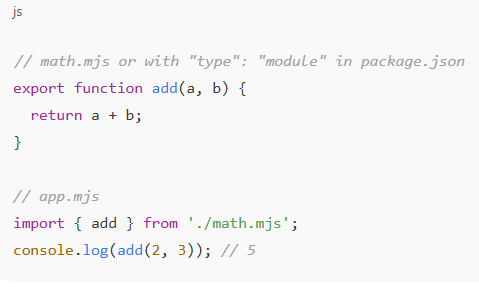
**CommonJS (CJS)**

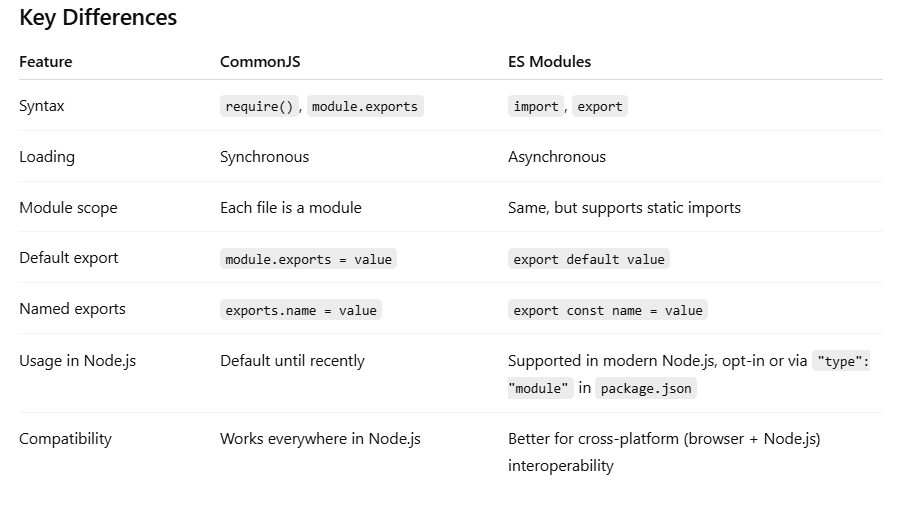
* **Syntax:** Uses require() to import and module.exports or exports to export.
* **Loading:** Modules are loaded **synchronously** (blocking) during runtime.
* **Default in Node.js:** This has been Node’s default module system for years.



**ES Modules (ESM)**

* **Syntax:** Uses import and export statements.
* **Loading:** Modules are loaded **asynchronously** and support static analysis (better for optimization).
* **Standardized:** Part of the ECMAScript standard, supported natively in browsers and now in Node.js (since v12+ with flags, stable in later versions).





**Q4. Explain how you would create a simple HTTP server in Node.js and handle routing manually without any frameworks.**

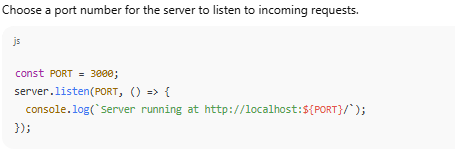
* 1. Import http module.



* 1. Create a server using createServer method.



* 1. Start the server using listen method.





**Q5. What is child process?**

A child process is a **separate process** that is created and managed by the parent process (your NodeJS application).

This allows you to run system commands (like listing files or checking memory usage)

* Execute scripts (like Python, Bash, or another NodeJS script)
* Perform CPU-intensive operations without blocking the main event loop
* Create multiple processes that run in parallel (multi-threading in NodeJS)

**Methods to Create Child Processes**

There are 4 methods used by child Process

1. **spawn() method**

* The[spawn() method](https://www.geeksforgeeks.org/what-is-spawn-in-node-js/) launches a new process with a specified command and arguments.
* **Providing streams for input/output.**
* It's **ideal for handling large outputs or long-running processes**.



**In this example**

* spawn('ls', ['-lh', '/usr']): Executes the ls command with -lh and /usr as arguments.
* child.stdout.on('data', ...): Listens for data from the standard output.
* child.stderr.on('data', ...): Listens for data from the standard error.
* child.on('close', ...): Listens for the close event, indicating the process has finished.

**2. fork() method**

* The fork() method is a special case of spawn().
* **It is used specifically for spawning new NodeJS processes.**
* It establishes an IPC (Inter-Process Communication) channel between the parent and child processes, allowing them to communicate via message passing.



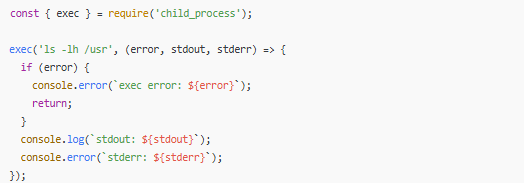
**In this example**

* **f**ork('child.js'): Spawns a new NodeJS process running the child.js module.
* child.on('message', ...): Listens for messages from the child process.
* child.send('Hello from parent'): Sends a message to the child process.

**3. exec() method**

Executes a command in a shell and buffers the output (stdout/stderr) in memory.

* Good for short commands with small output.

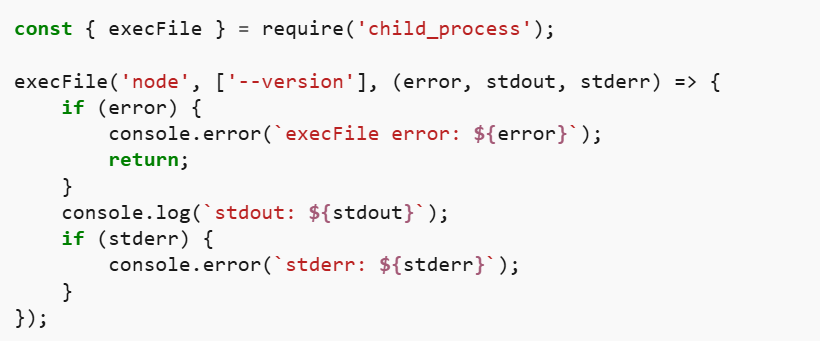


**In this example**

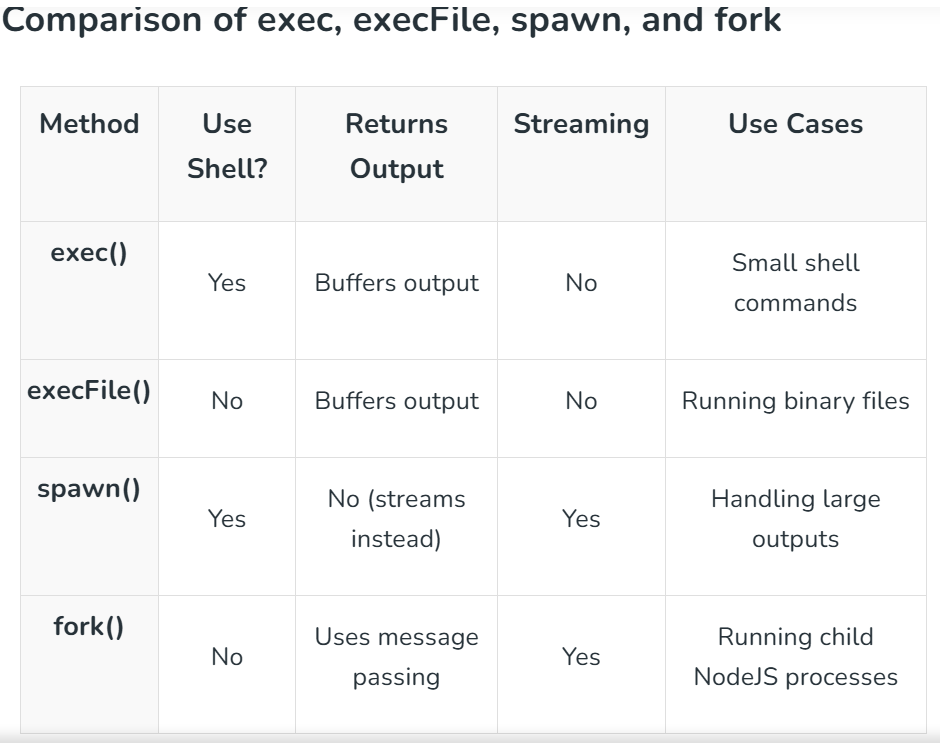
* exec('ls -lh /usr', ...): Executes the ls command with -lh and /usr as arguments.
* The callback receives error, stdout, and stderr as parameters.
* Logs the standard output and error to the console.

**4. execFile() method**

The execFile() method runs an executable file directly without spawning a shell, making it more efficient than exec() for running binaries and scripts directly.



* execFile('node', ['--version'], ...): Executes the NodeJS binary with the --version flag.
* The callback receives error, stdout, and stderr as parameters.
* Logs the standard output and error to the console.



**Best Practices for Using Child Processes**

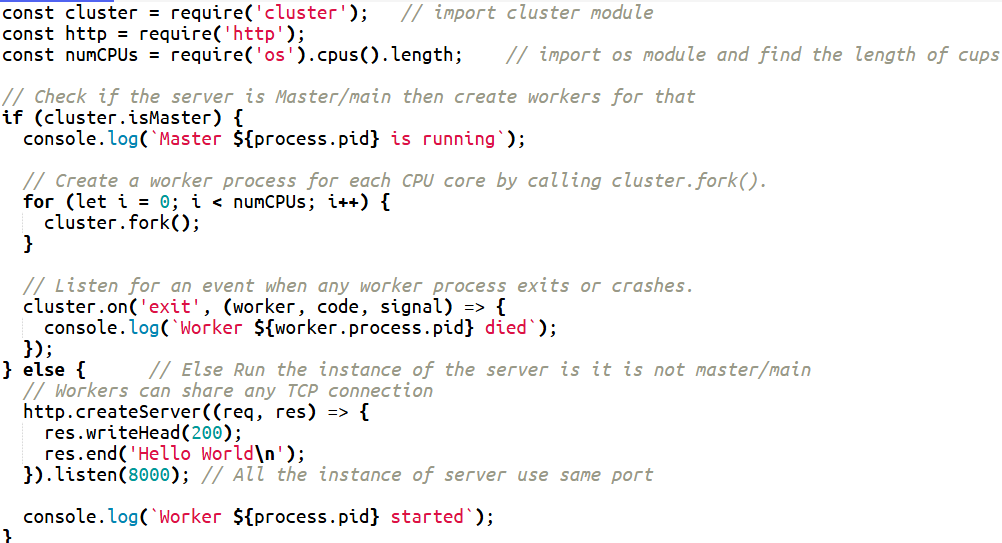
* Use exec() for simple shell commands (when output size is small).
* Use execFile() for executing binary files (faster & safer).
* Use spawn() for handling large data or real-time output streaming.
* Use fork() for multi-processing and parallel execution in NodeJS.
* Always handle errors properly (use stderr and error events).
* Use asynchronous methods whenever possible to avoid blocking the event loop.

**Q6. What is the purpose of the cluster module in Node.js?**

**Node.js cluster module** is use to create multiple Node.js processes to spread the work across CPU cores. It’s great when you want separate processes for better reliability.

**Why use the cluster module?**

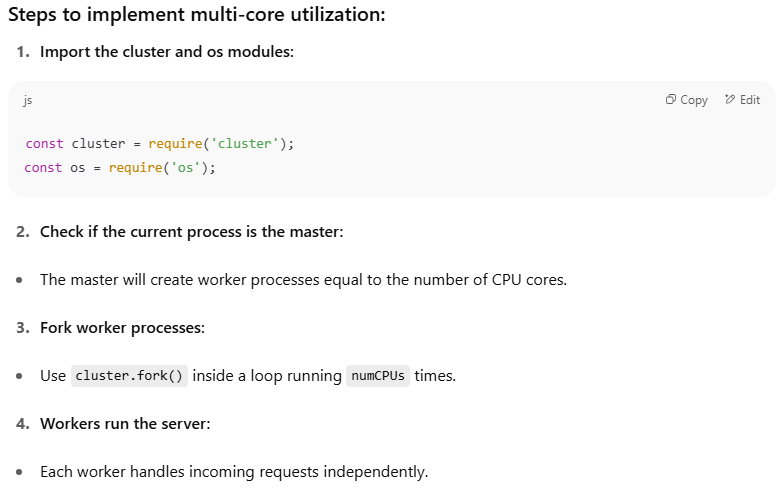
* **Node.js runs JavaScript in a single thread** by default, so it can only use one CPU core.
* The cluster module lets you **fork multiple worker processes**, each running an instance of your server.
* These workers share the same port and can handle requests in parallel.
* This improves **scalability** and **throughput**.
* If one worker crashes, other workers keep running, improving **fault tolerance**.



**Summary:**

* **Purpose:** Improve Node.js app performance by using multiple CPU cores.
* **How:** Create multiple worker processes that share the same server port.
* **Benefits:** Better CPU utilization, higher concurrency, fault tolerance.

**Q. How do you implement multi-core utilization?**



**Q7. How does the V8 engine optimize JavaScript code execution in Node.js?**

### 1. ****Parsing and Abstract Syntax Tree (AST) Generation****

* V8 first parses the JavaScript source code and converts it into an **Abstract Syntax Tree (AST)**, which represents the structure of the code.

### 2. ****Ignition: The Interpreter****

* V8 initially uses an interpreter called **Ignition**.
* Ignition translates the AST into **bytecode** (a lower-level, more efficient representation than source code).
* Running the bytecode is faster than interpreting the original source code directly.
* This allows V8 to start running the code quickly without waiting for full compilation.

### 3. ****Profiling and Hot Code Detection****

* While executing bytecode, V8 **profiles the running code** to identify "hot" functions or loops — i.e., parts of the code that are executed frequently or are performance-critical.
* This profiling gathers runtime information about types of variables, function calls, etc.

### 4. ****TurboFan: The Just-In-Time (JIT) Compiler****

* Once V8 identifies hot code, it passes it to the optimizing JIT compiler called **TurboFan**.
* TurboFan uses the runtime profiling data to perform **aggressive optimizations**:
  + **Inlining functions** to reduce call overhead.
  + **Constant folding** (calculating constant expressions at compile time).
  + **Dead code elimination** (removing code that will never run).
  + **Type specialization** (generating optimized machine code based on observed types).
* TurboFan compiles the hot code into highly efficient **machine code** for the CPU.

### 5. ****Deoptimization and Bailout****

* If assumptions made during optimization become invalid (for example, variable types change), V8 can **deoptimize** and revert to less optimized bytecode execution, maintaining correctness.

6. **Garbage Collection (GC) Optimization**

* V8 includes an efficient garbage collector to reclaim memory from objects that are no longer needed, helping keep performance smooth.

**Q8. What are some best practices for securing a Node.js application?**

### 1. ****Keep Dependencies Updated****

* Regularly update Node.js and npm packages to patch vulnerabilities.
* Use tools like npm audit or Snyk to scan for vulnerable dependencies.

### 2. ****Validate and Sanitize Input****

* Always validate and sanitize user inputs to prevent injection attacks (e.g., SQL injection, NoSQL injection, command injection).
* Use validation libraries like Joi or express-validator, **Validator.js**.

### 3. ****Use HTTPS****

* Serve your app over HTTPS to encrypt data in transit.
* Use certificates from trusted CAs (e.g., Let's Encrypt).

### 4. ****Manage Environment Variables Securely****

* Store secrets (API keys, DB passwords) in environment variables or secure vaults.
* Avoid hardcoding credentials or pushing them to version control.

### 5. ****Implement Proper Authentication and Authorization****

* Use strong authentication methods like OAuth, JWT, or sessions.
* Implement role-based access control (RBAC) to limit what users can do.
* Protect against brute-force attacks (e.g., rate limiting, account lockout).

### 6. ****Use Helmet for HTTP Security Headers****

* Add security-related HTTP headers (Content-Security-Policy, X-Frame-Options, etc.) using [Helmet](https://github.com/helmetjs/helmet).

js

CopyEdit

const helmet = require('helmet');

app.use(helmet());

### 7. ****Prevent Cross-Site Scripting (XSS)****

* Escape or sanitize user-generated content before rendering.
* Use libraries like DOMPurify or frameworks with built-in XSS protections.

**What is Cross-Site-Scripting**

* Cross-Site Scripting (XSS) is a type of security vulnerability that allows attackers to inject malicious scripts into web pages viewed by other users.
* This can lead to various harmful outcomes, such as stealing sensitive information (like cookies or session tokens), defacing websites, or redirecting users to malicious sites.
* XSS attacks exploit the trust a user has in a particular website and the trust a website has in the user's browser.

### 8. ****Prevent Cross-Site Request Forgery (CSRF)****

* Use CSRF tokens (e.g., with **csurf middleware** in Express).

### 9. ****Avoid Using eval() and Unsafe Functions****

* Avoid eval(), new Function(), and similar functions that execute arbitrary code.

### 10. ****Limit Request Size and Rate****

* Use body parsers that limit payload size.
* Implement rate limiting with middleware like express-rate-limit.

### 11. ****Secure Session Management****

* Use secure, HttpOnly cookies for sessions.
* Set proper cookie attributes: secure, SameSite, HttpOnly.

### 12. ****Log and Monitor****

* Log important security events.
* Use monitoring tools to detect suspicious behavior.

### 13. ****Error Handling****

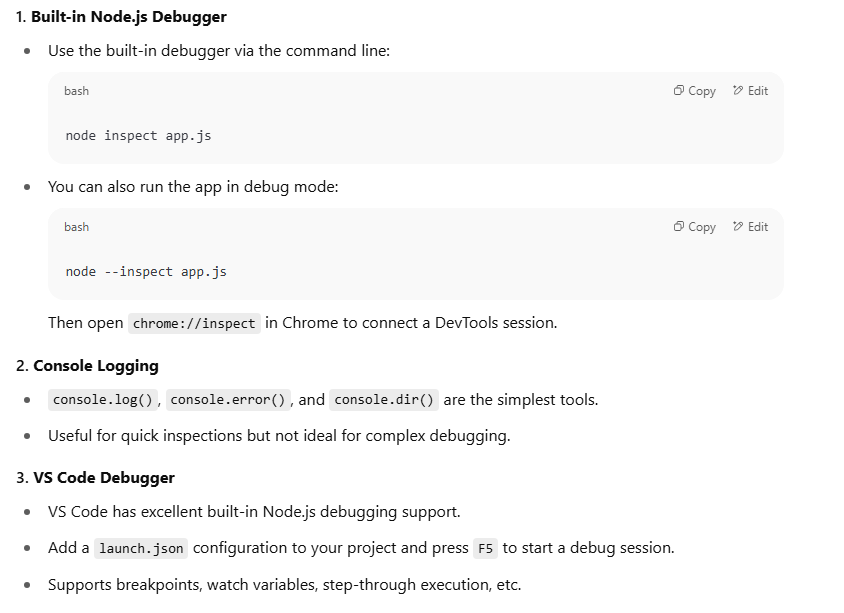
* Don’t leak stack traces or sensitive info in error messages returned to clients.

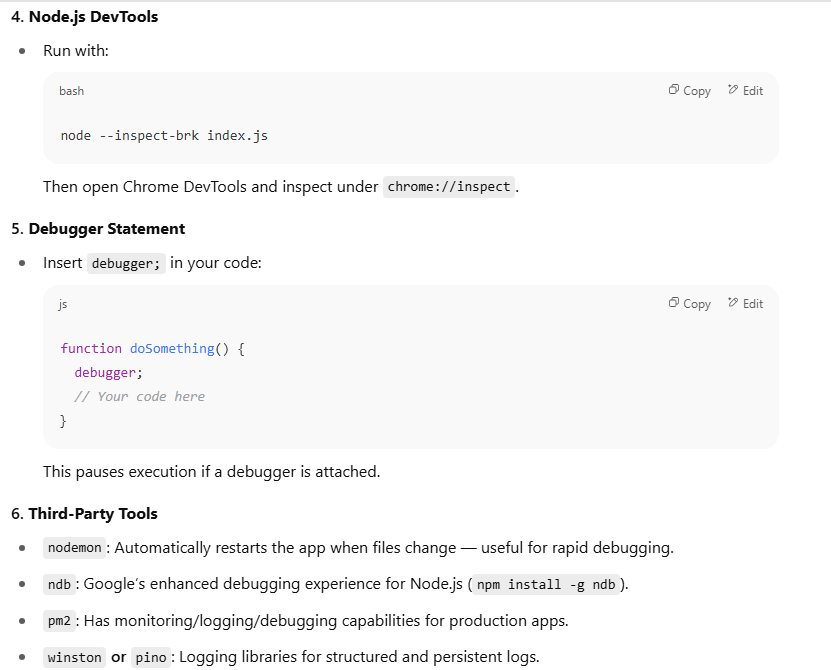
**Q9.** **How can you debug a Node.js application? What tools or methods do you prefer?**

**What is Node Debugging?**

**Debugging**is the process of finding and fixing errors (bugs) in your code. It's an important part of software development, as almost all code has some bugs at some point.

* They help in identifying runtime errors.
* Improves code efficiency by detecting memory leaks.
* Enables developers to monitor application execution flow.
* Ensures error-free and reliable applications.

****

****

**Node.js asynchronous programming interview questions**

**List Of Questions**

**Core Concepts**

1. **What is the event loop in Node.js, and how does it handle asynchronous operations?**
2. **How is asynchronous programming handled in Node.js? Name and compare the different techniques.**
   * Callbacks
   * Promises
   * async/await
3. **Explain the difference between microtasks and macrotasks in Node.js.**
4. **What are the advantages and disadvantages of using async/await over Promises and Callbacks?**
5. **How does Node.js handle I/O-bound vs CPU-bound tasks?**

**🔹 Practical / Scenario-Based**

1. **You have multiple asynchronous API calls to make. How would you execute them concurrently and then wait for all to finish?**
   * Discuss Promise.all, Promise.allSettled, etc.
2. **How would you implement a retry mechanism for a failed asynchronous operation?**
3. **What happens if you forget to await an async function?**
4. **You have a piece of code that uses fs.readFile and you want to convert it to use Promises. How would you do that?**
5. **How would you handle errors in a chain of asynchronous calls using Promises vs async/await?**

**🔹 Code-Based / Debugging**

1. ❓ *What’s the output of the following code? Explain why.*

js

CopyEdit

console.log('Start');

setTimeout(() => {

console.log('Timeout');

}, 0);

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

console.log('Promise');

});

console.log('End');

1. **How would you handle a situation where multiple asynchronous database calls need to be made in a specific order?**
2. **Explain how Promise.race() works with an example.**
3. **Can you create your own custom Promise wrapper function? How does it work?**

**🔹 Advanced / Design**

1. **How do you avoid "callback hell"? Can you show refactoring from nested callbacks to Promises or async/await?**
2. **How would you handle thousands of concurrent asynchronous requests without overwhelming the system?**

* Concepts like throttling, batching, or queueing

1. **How would you detect and handle memory leaks in long-running asynchronous operations in Node.js?**

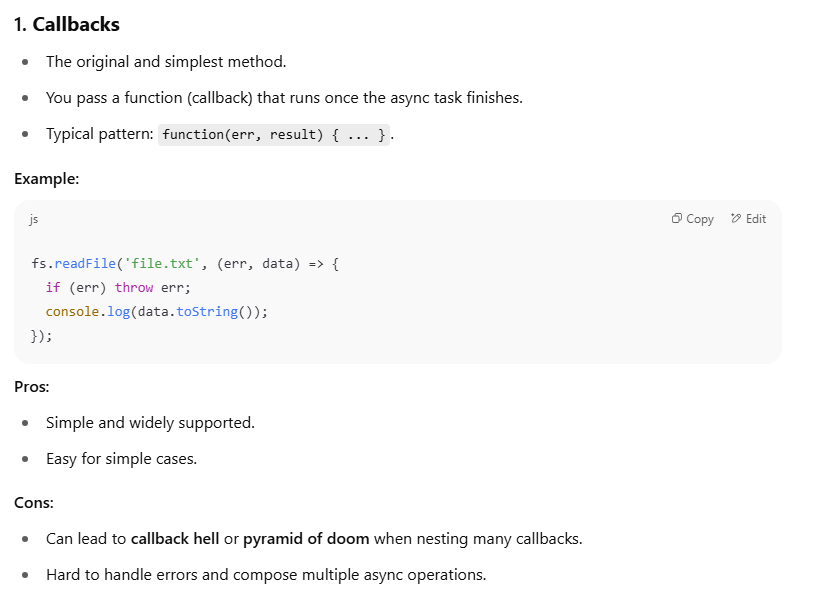
**Q1. How is Asynchronous Programming Handled in Node.js?**

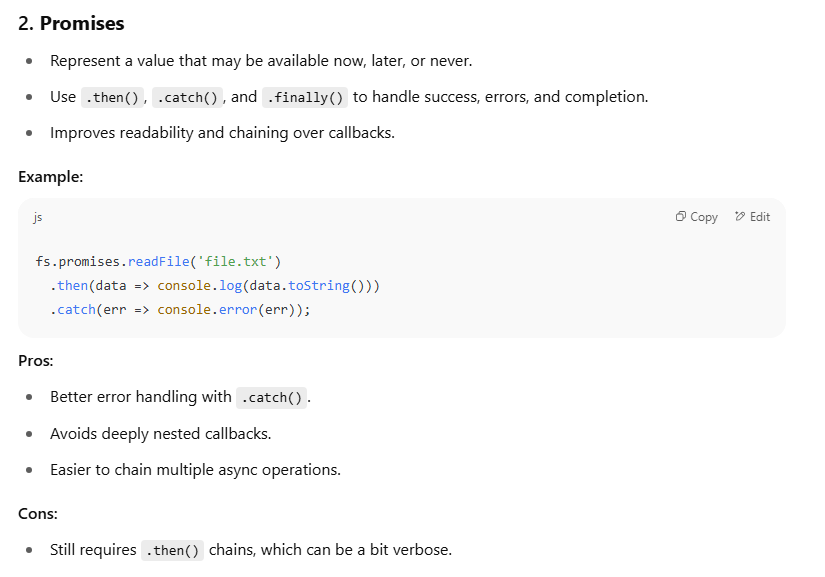
Node.js uses an **event-driven, non-blocking I/O model**. This means:

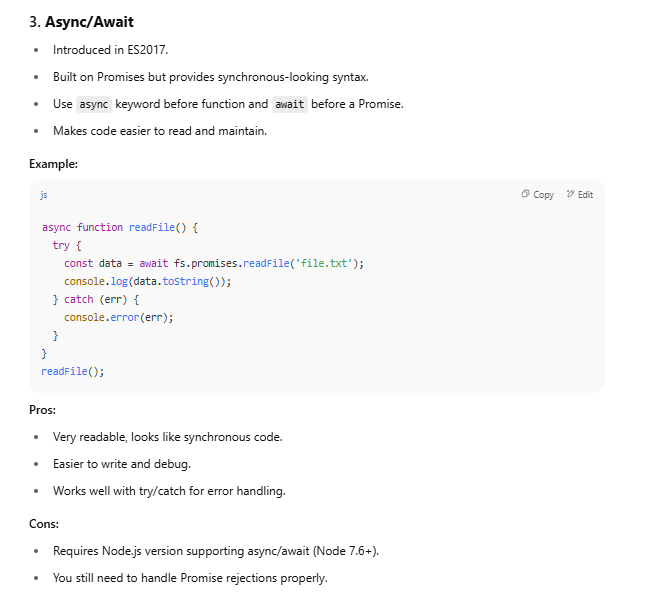
* When an asynchronous operation (e.g., reading a file, querying a database, making an HTTP request) is initiated, Node.js offloads it to the system or thread pool.
* Instead of waiting (blocking) for the operation to complete, Node.js continues executing other code.
* Once the async operation finishes, its **callback** or **promise** resolves, and the event loop schedules the associated handler to run.

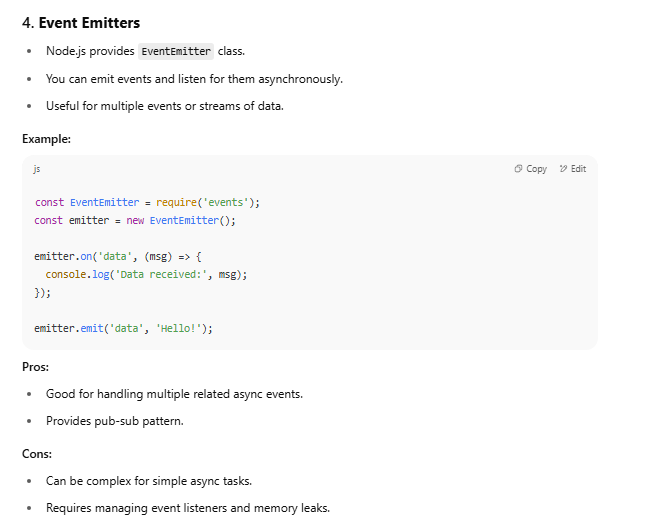
This approach keeps Node.js apps efficient and scalable by preventing blocking.

**Different Techniques for Asynchronous Programming in Node.js**

****

****

****

****

**Q2. Explain the difference between microtasks and macrotasks in Node.js.**

**Micro-task**

A **micro-task** is a short callback function that is executed **after the currently running JavaScript code completes**, but **before the next macro-task starts**.  
It is used for operations that should happen as soon as possible, like:

* Promise.then()
* queueMicrotask()
* MutationObserver

**Macro-task**

A **macro-task** is a larger, independent unit of work that gets executed by the JavaScript engine in the **main task queue**, often scheduled by external APIs or timers.  
Examples include:

* setTimeout()
* setInterval()
* DOM events (click, input)
* I/O tasks (fetch, load)

****

Console.log(‘END’);



**Q3. What are the advantages and disadvantages of using async/await over Promises and Callbacks?**

**Advantages of async/await**

1. **Cleaner and More Readable Code**

* Code looks more like synchronous code.
* Easier to follow and reason about, especially for long chains of asynchronous logic.

2. **Error Handling with try/catch**

* Handle errors just like synchronous code using try/catch.

3. **Avoids Callback Hell**

* Prevents deeply nested callbacks.
* Makes complex async flows more maintainable.

4. **Improved Debugging**

* Easier to set breakpoints and trace stack traces compared to nested callbacks or chained Promises.

**❌ Disadvantages of async/await**

**1. Only Works with Promises**

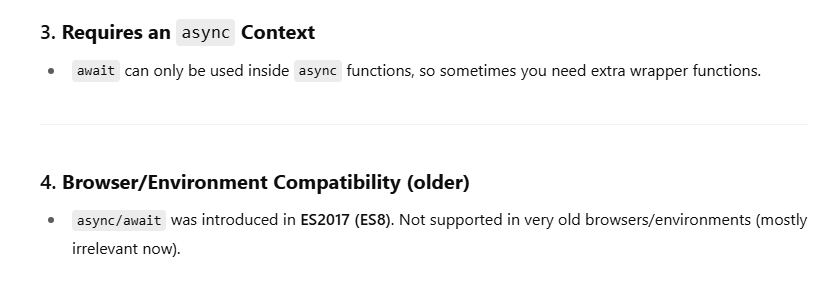
* You **can’t use await with traditional callbacks** or non-Promise-based async APIs.
* You often need to wrap older callback-style APIs (e.g., using util.promisify in Node.js

2. **Sequential Execution by Default**

* If you're not careful, await can cause **tasks to run one after the other**, even if they don’t depend on each other.

❗ Slower than Promise.all if not optimized.





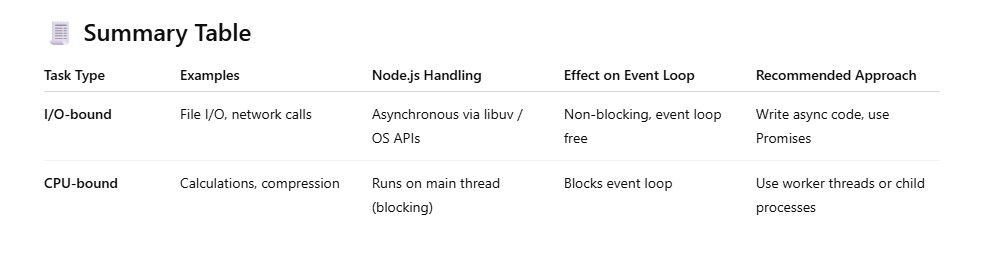
**Q4. How does Node.js handle I/O-bound vs CPU-bound tasks?**

1. **I/O-bound Tasks**

* **What:** Tasks that spend most of their time waiting for external operations to complete, such as:
  + Reading/writing files
  + Network requests (HTTP calls, database queries)
  + Timers (setTimeout, setInterval)
* **How Node.js handles them:**
  + Uses a **non-blocking, asynchronous model**.
  + Delegates I/O operations to **libuv’s thread pool** or the OS async APIs.
  + While waiting for I/O, the main event loop **continues running other JavaScript code**.
  + When I/O finishes, a callback or Promise resolution is queued and executed in the event loop.
* **Effect:**  
  Node.js handles thousands of I/O-bound tasks concurrently and efficiently without blocking the single main thread.

2. **CPU-bound Tasks**

* **What:** Tasks that require heavy computation and keep the CPU busy, such as:
  + Large calculations
  + Data processing
  + Image or video processing
  + Cryptography or compression
* **How Node.js handles them:**
  + These tasks **run on the single main JavaScript thread** by default.
  + **CPU-heavy tasks block the event loop**, causing delays in handling other requests.
  + Node.js is **not ideal for CPU-bound tasks** when run synchronously.
* **Solutions for CPU-bound tasks:**
  + Offload heavy work to **worker threads** (introduced in Node.js 10+).
  + Use **child processes** to run separate Node.js processes.
  + Offload to native addons or external services.
  + Use asynchronous APIs whenever possible.

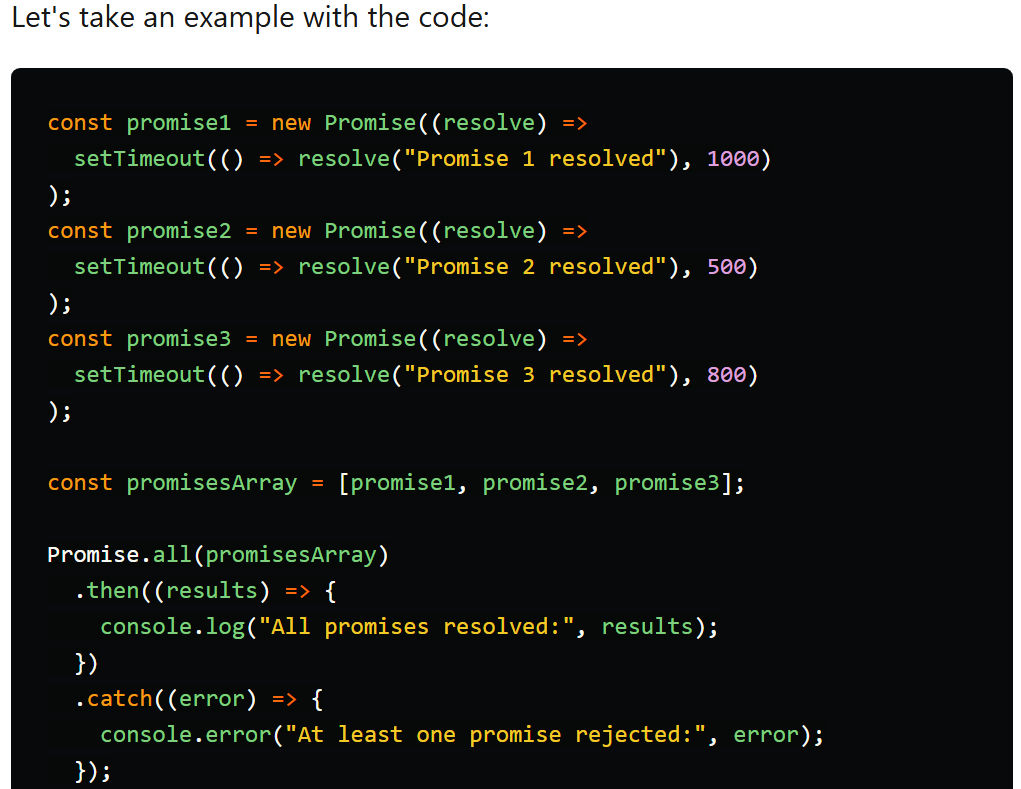
**Q5. You have multiple asynchronous API calls to make. How would you execute them concurrently and then wait for all to finish?**

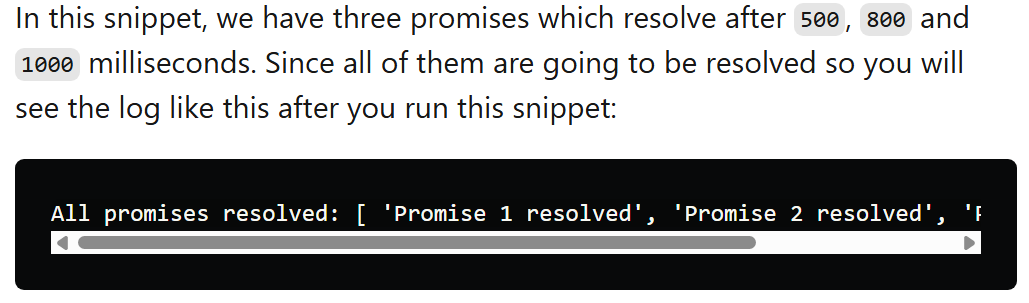
**Discuss Promise.all, Promise.allSettled, etc.**

## Promise.all()

Promise.all method in JavaScript is used to handle multiple promises concurrently/parallel and wait for all of them to resolve.

Note: **ALL PROMISES MUST BE RESOLVED FOR IT TO RETURN A RESPONSE. If any of the promise reject then overall output is reject.**





If any of the promise rejects, it won't wait for others to resolve and will return with reason of the promise that is rejected. Look at example below to get the better idea about this:



## Promise.allSettled()

The Promise.allSettled method is used to handle multiple promises concurrently/Parallel, just like Promise.all, but it waits for all the promises to settle (either resolve or reject) before proceeding.

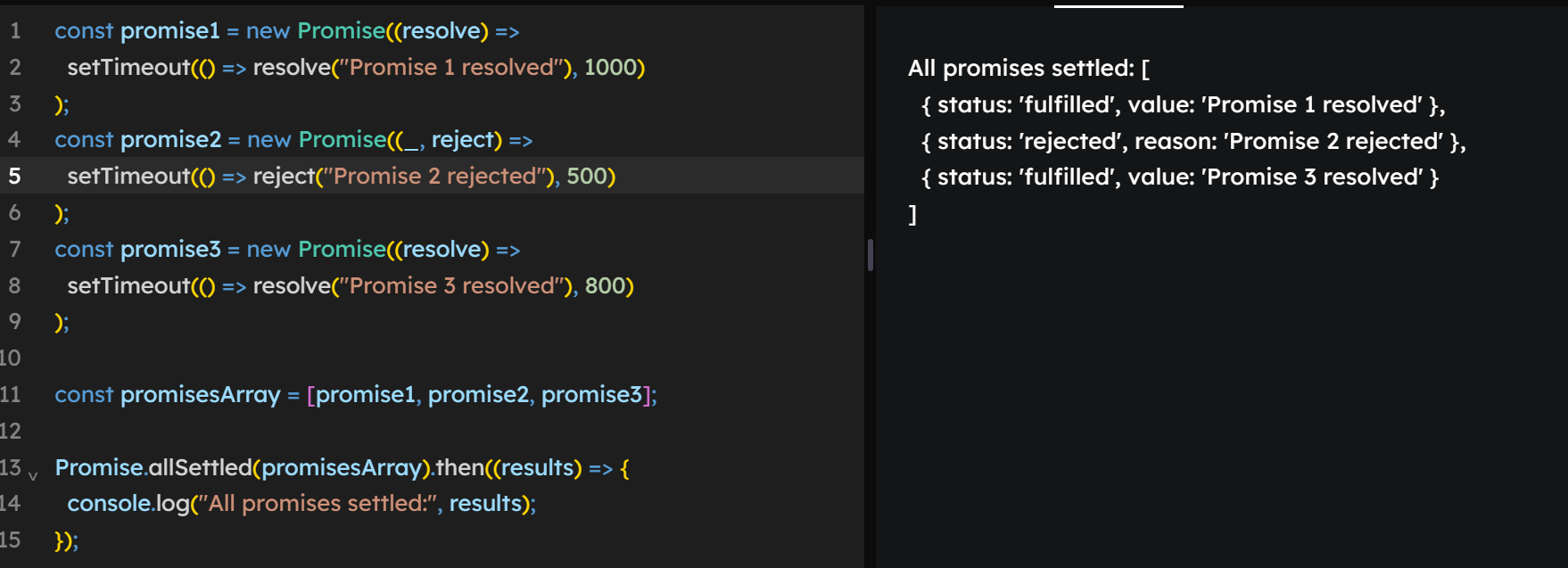
It **returns an array of objects** representing the outcomes of the input promises, including their values or reasons for rejection.

Note : Promise.allSettled() always returns array of objects with status key which denotes fulfilled or rejected.

* If a promise is fulfilled then you can get response with value key.
* If the promise is rejected then you can find the reason in reason key.



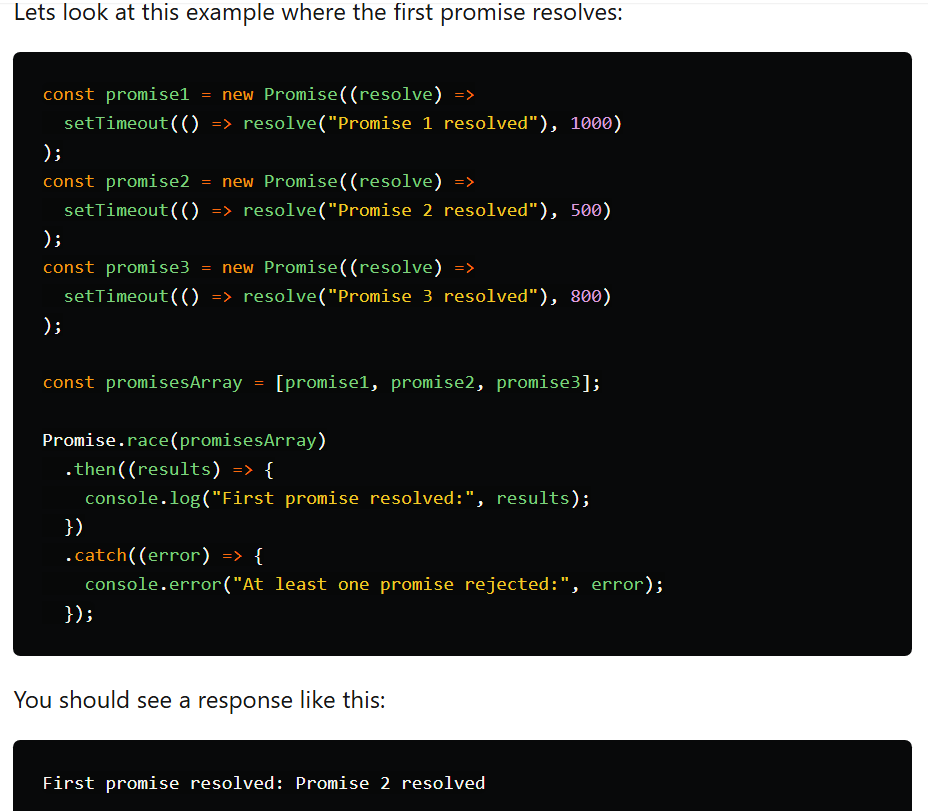
Now we will try and reject promise2 like we did previously and see the response:

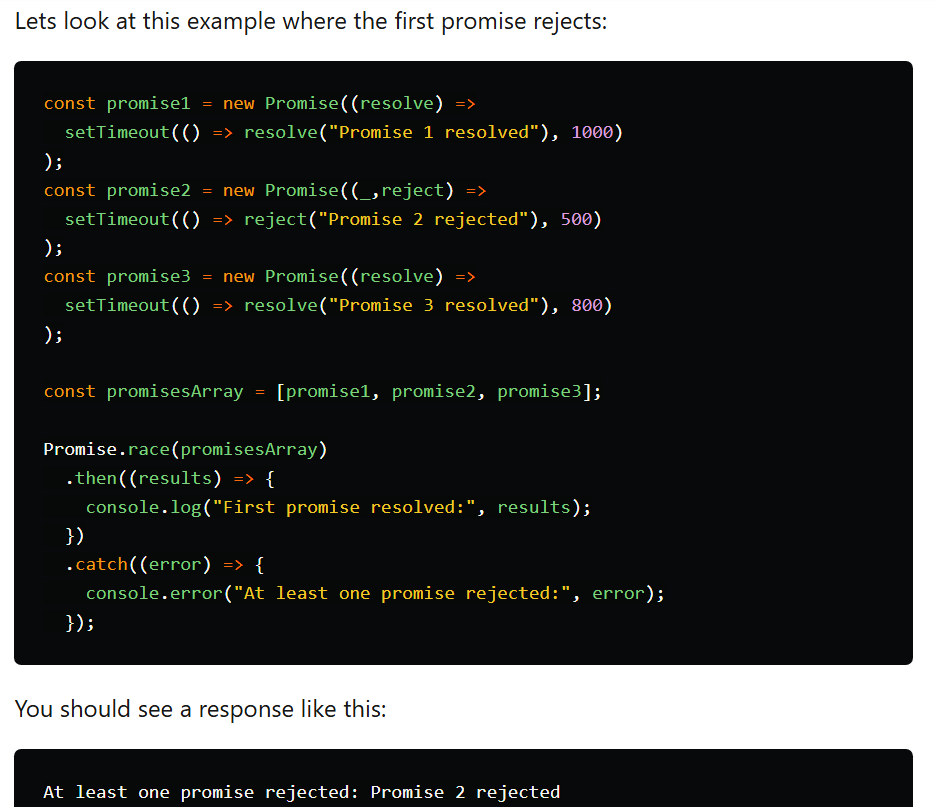


## Promise.race()

The Promise.race method in JavaScript is used to handle multiple promises concurrently, but it resolves or rejects as soon as the **first promise in the input array settles, either by resolving or rejecting**.  
This can be useful when you're interested in the result of the first promise to complete, regardless of whether it's a success or failure.

note: **race** does not care if all are resolved or all are rejected. It will give you **first settled result** whether it is resolved or rejected.



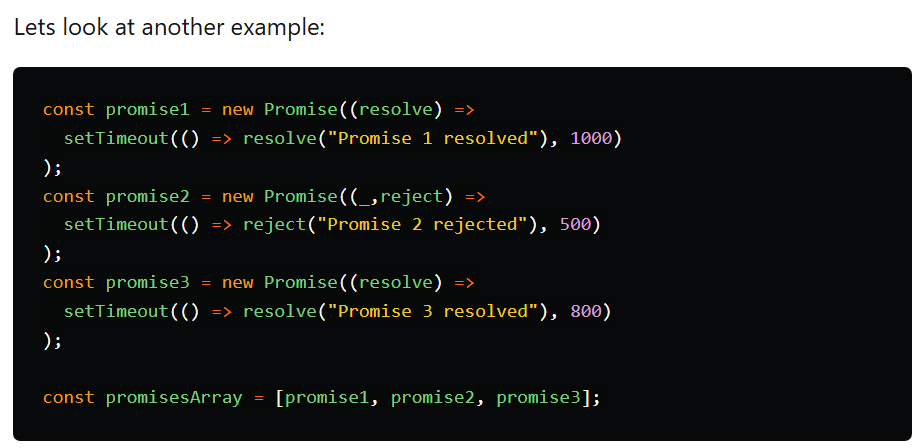


## Promise.any()

It is somewhat similar to race method but with few minor differences:

* It will return with **first resolved promise**.
* If all promises are rejected, it will give you an aggregated error.





If you run this snippet in which the promise with shortest delay promise2 of 500ms is rejected and promise3 which has next shortest delay 800ms then:

* race will return promise2 with rejected reason
* any will return promise3 with resolved result.



**Q6.** **What is a Retry Mechanism?**

A **retry mechanism** is a way to **automatically re-attempt** a failed operation — like an API call or file read — **a few times** before giving up.**Why is it useful?**

Sometimes, failures are **temporary**, such as:

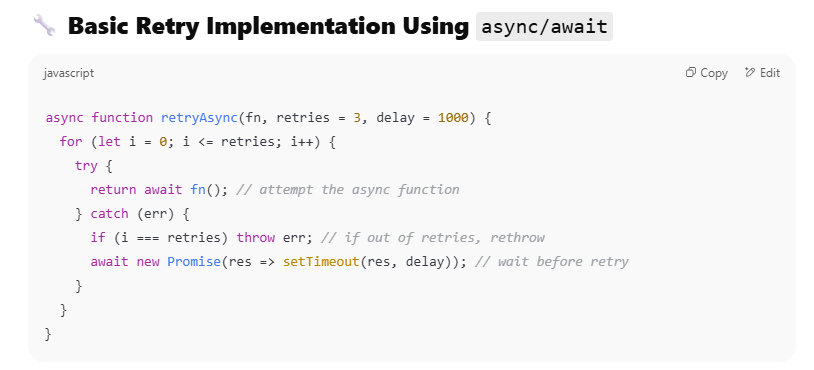
* Network hiccups 🌐
* Server overload or timeout 🕒
* Third-party API delay

Instead of failing immediately, your code can **retry** the operation after a short wait.

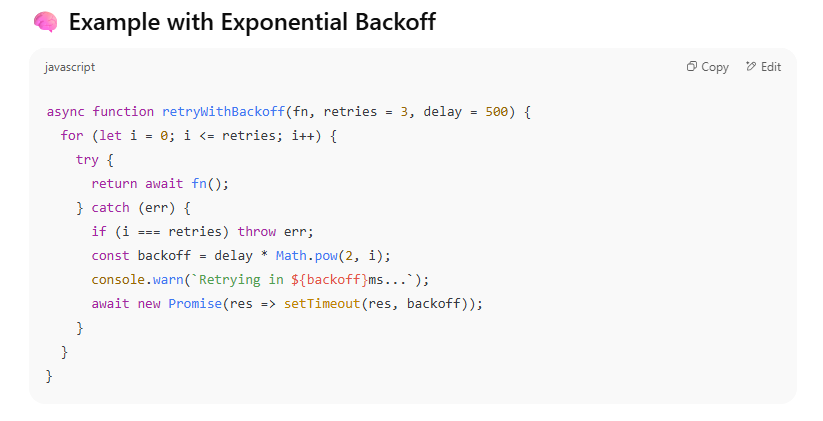
**Basic Retry Logic in Code:**

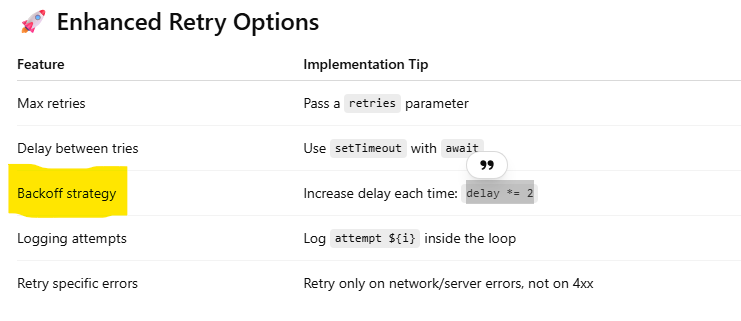
1. Try the operation (e.g., fetch data).
2. If it fails:
   * Wait a bit (optional).
   * Try again.
3. Repeat this for a **limited number of times** (e.g., 3 attempts).
4. If it still fails, show an error.

**Handle async operation failuer using retry**



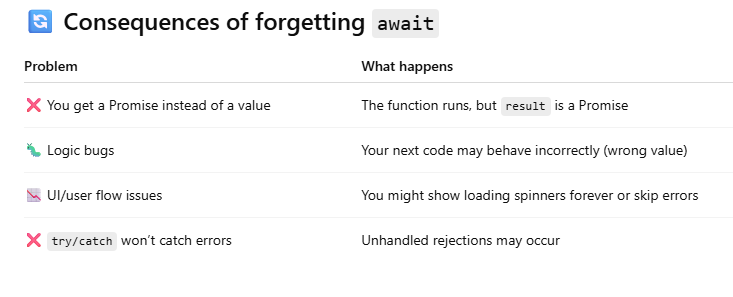


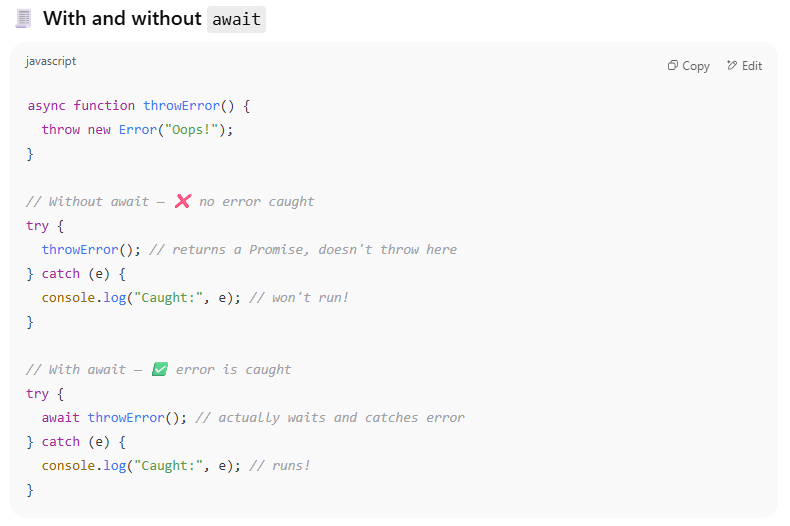




**Q7.** **What happens if you forget to await an async function?**

If you **call an async function without await**, it will **start running**, but instead of getting the result, **you’ll get a Promise**.

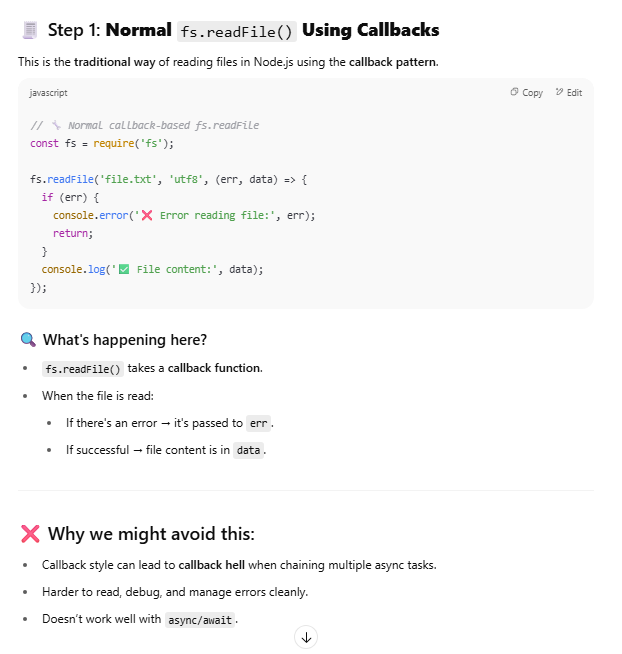




## Summary

* Forgetting await means:
  + You get a **Promise**, not a value.
  + Errors won't be caught in try/catch.
  + Your logic may break silently.
* **Always use await (or .then)** when you want to use the actual result of an async function.

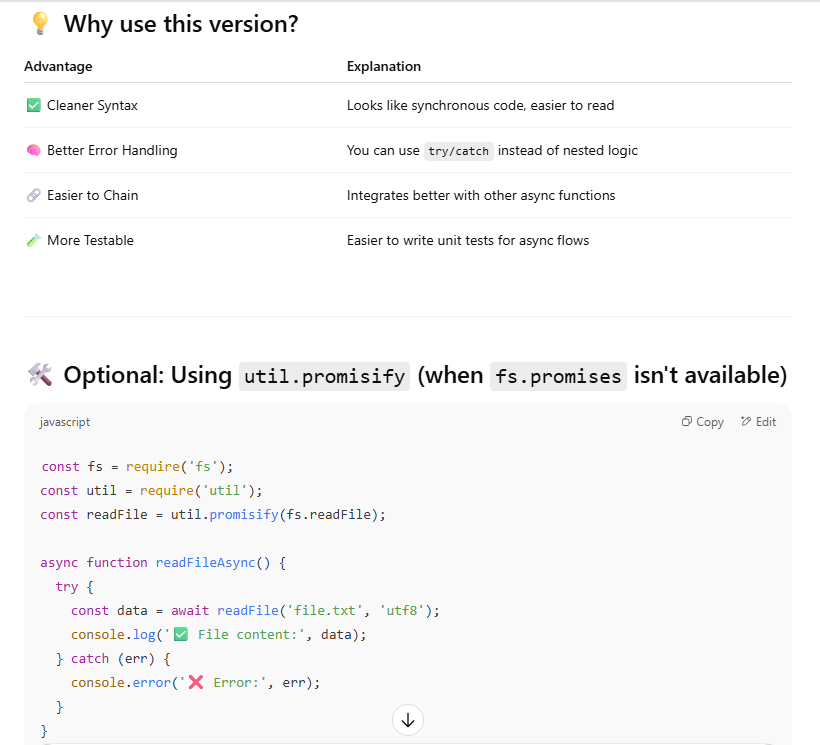
**Q8.** **You have a piece of code that uses fs.readFile and you want to convert it to use Promises. How would you do that?**



There are 2 ways

* **Use fs.promises.readfile**
* Using util.promisify (when fs.promises isn't available) - Optional

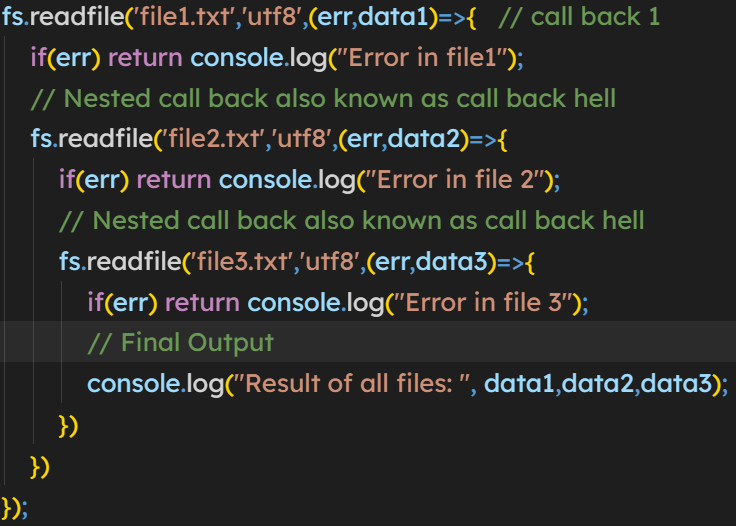




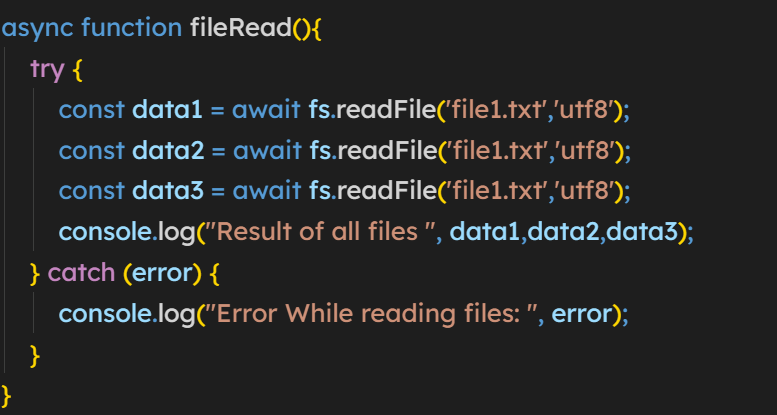
**Q9.** **How do you avoid "callback hell"? Can you show refactoring from nested callbacks to Promises or async/await?**

**1. Callback hell** (aka “pyramid of doom”) happens when you nest too many callbacks, making code:

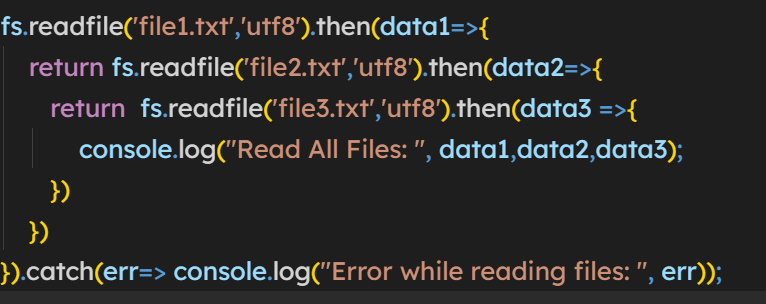
* Hard to read 📉
* Hard to maintain 🛠
* Difficult to handle errors ❌



**2. Change it to async/await**

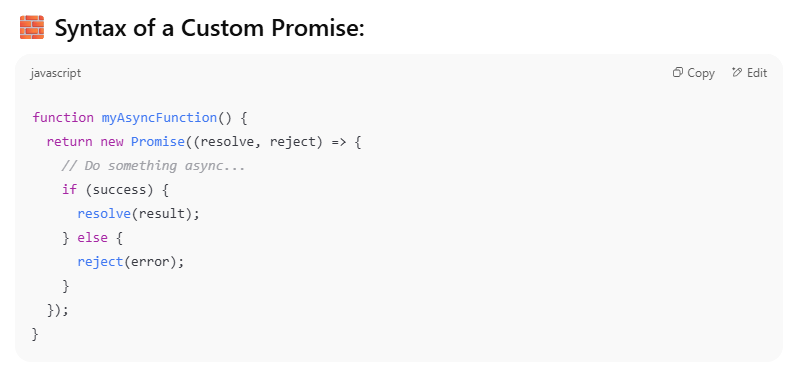
****

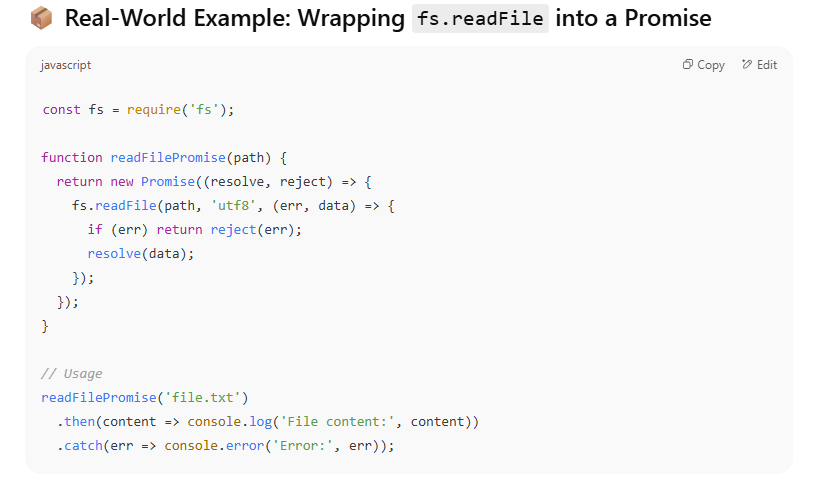
**3. Change it to Promise**

****

**Q9. Can you create your own custom Promise wrapper function? How does it work?**

A **Promise wrapper** is a function that returns a new Promise and allows you to wrap any logic — especially **asynchronous or callback-based operations** — in a modern, Promise-based interface





**Q10. How would you handle thousands of concurrent asynchronous requests without overwhelming the system?**

* **Concepts like throttling, batching, or queueing**

1. **Throttling**

**Throttling** limits the **number of tasks running at the same time**. Instead of launching 1,000 requests at once, you launch only a few (e.g. 10) concurrently.

**Why use it?**

To **avoid overloading** the system or hitting **rate limits** on APIs.



**Require p-limit to handle the limit of API calls and overcome**

2. **Batching**

**Batching** means grouping tasks into **chunks** and running one group at a time.

**Why use it?**

To control **burst load**, give time between groups, or respect API call limits like "100 requests per minute".



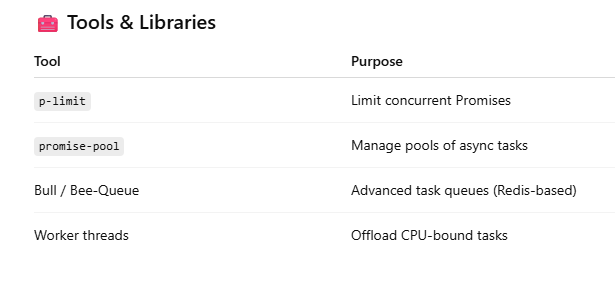
3. **Queueing**

**Queueing** puts tasks into a **waiting line** and processes them **one by one** or with limited concurrency. It's like a worker queue system.

📌 **Why use it?**

* Fine control over task flow.
* Useful when tasks are added dynamically.
* Helps manage **both concurrency and order**.





**Q11. What Is a Memory Leak?**

A **memory leak** happens when your app **keeps allocating memory** (e.g., in variables, closures, or event listeners) but **never releases it**, even when it's no longer needed.

Over time:

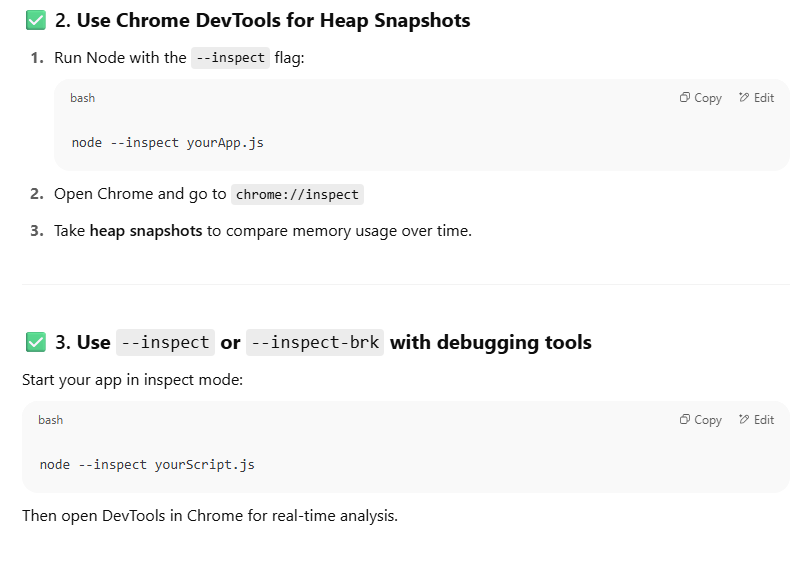
* Memory usage grows 📈
* The process slows down 🐌
* Eventually, it crashes with OutOfMemory ❌

**How to Detect Memory Leaks in Node.js**

1. **Monitor Memory Usage**

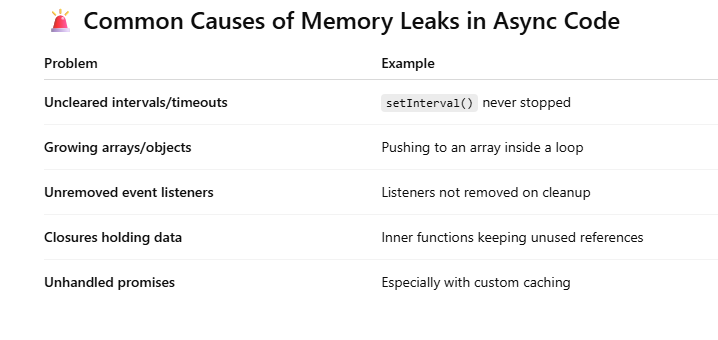
Use built-in tools to monitor memory growth:

****

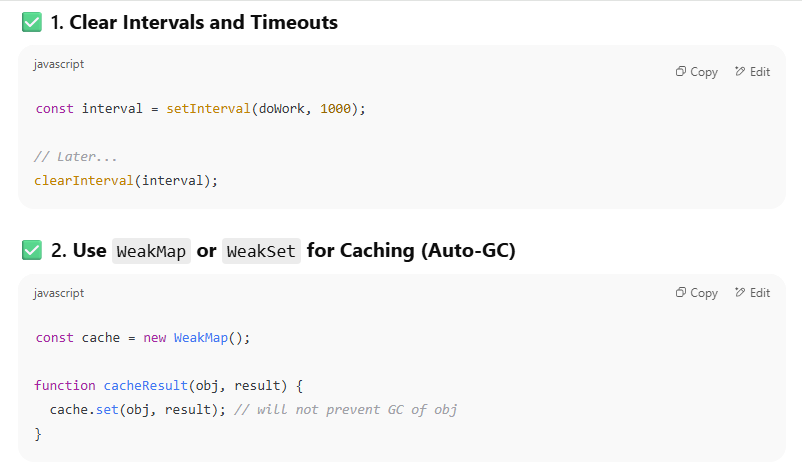
****

4. **Use** clinic.js **or** heapdump **for deeper analysis**

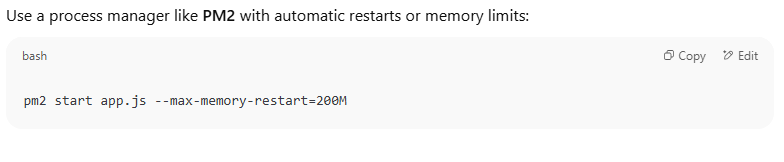
* [clinic](https://clinicjs.org/): CLI tool that gives you flame graphs, event loop usage, and memory usage.
* [heapdump](https://www.npmjs.com/package/heapdump): Captures memory snapshots from running apps.



**🛠️ How to Prevent / Handle Memory Leaks**

****

**Restart Long-running Workers Periodically**

****

****

**CRUD**

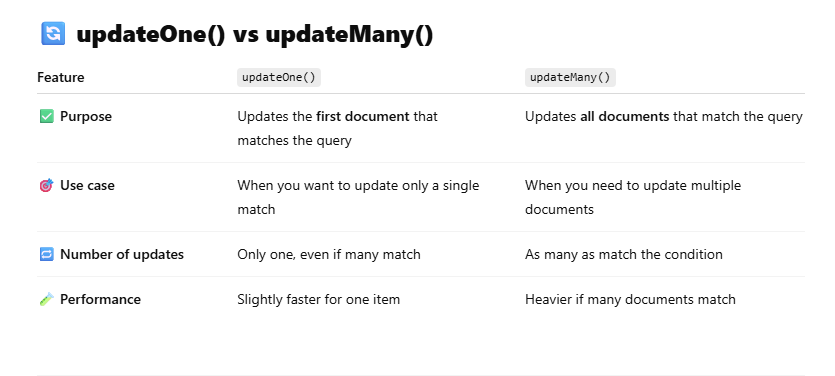
**Q1. How do you insert, read, update, and delete a document in MongoDB?**

****

****

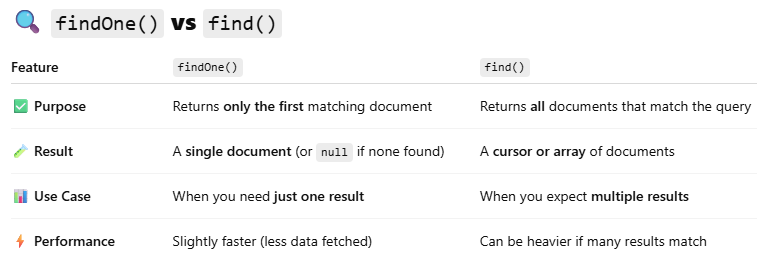
****

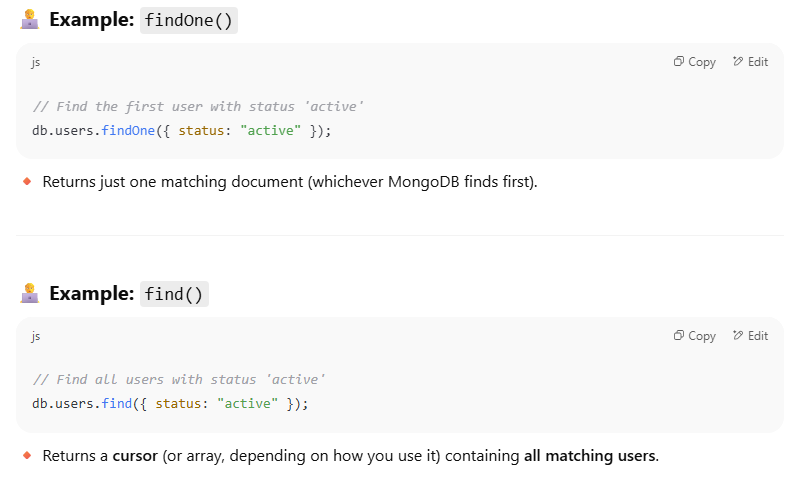
**Q3. What’s the difference between updateOne and updateMany?**

****

****

**Q4. What’s the difference between findOne() and find()?**



****