Node.js Interview Questions Guide

This document is a **comprehensive Node.js interview preparation guide** containing 50 **carefully curated questions** with detailed explanations, architectural deep-dives, and production-grade best practices.

It is designed to help engineers master **Node.js internals (Event Loop, Streams, Libuv)** and build scalable backend systems, enabling them to confidently approach interviews at **FAANG companies, unicorn startups, and high-growth product-based firms**.

Who This Guide Is For

- **Backend Developers** looking to master asynchronous programming and system architecture.
- **Full Stack Engineers** aiming to strengthen their server-side expertise.
- **Frontend Developers** transitioning into backend development with Node.js.
- **Tech Leads & Architects** wanting to refine their understanding of Node.js performance, security, and scalability patterns.

No.	Questions	Level
1	How does Node.js handle concurrency given that it is single-threaded?	Beginner
2	What is the difference between `process.nextTick()` and `setImmediate()`?	Beginner
3	Explain the concept of “Stub” in Node.js testing.	Beginner
4	What is the purpose of `module.exports` vs. `exports`?	Beginner
5	What is the “Error-First Callback” pattern?	Beginner
6	What is the difference between `spawn` and `fork`?	Beginner
7	What are the key features of the `package.json` file?	Beginner
8	What is REPL?	Beginner
9	Explain the purpose of the `buffer` class.	Beginner
10	What is Middleware?	Beginner
11	What is the difference between `dependencies` and `devDependencies`?	Beginner
12	How do you update dependencies in a Node.js project?	Beginner
13	What is the global object in Node.js?	Beginner
14	What are the core modules of Node.js?	Beginner
15	What is `npm`?	Beginner
16	Explain the “Event Loop” in detail.	Intermediate
17	What are “Streams” and what are the different types?	Intermediate
18	How do you handle errors in Promises vs. Callbacks?	Intermediate
19	What is “Callback Hell” and how do you avoid it?	Intermediate
20	What is `process` in Node.js?	Intermediate
21	Explain `cluster` module in Node.js.	Intermediate

No.	Questions	Level
22	What is `libuv`?	Intermediate
23	How does `require()` work under the hood?	Intermediate
24	What are “Worker Threads”?	Intermediate
25	How do you handle JWT Authentication in Node?	Intermediate
26	What is the difference between `exec`, `execFile`, `spawn`, and `fork`?	Intermediate
27	Explain `EventEmitter`.	Intermediate
28	How do you prevent callback nesting without using Promises?	Intermediate
29	What is semantic versioning (SemVer)?	Intermediate
30	How do you handle unhandled exceptions in Node.js?	Intermediate
31	Explain the role of `NODE_ENV`.	Intermediate
32	How do you debug a Node.js application?	Intermediate
33	What is the difference between `null` and `undefined`?	Intermediate
34	What is CORS and how do you handle it in Node.js?	Intermediate
35	Explain the concept of “Backpressure” in Streams.	Intermediate
36	How does the V8 Engine work?	Advanced
37	Explain Memory Leaks in Node.js and how to debug them.	Advanced
38	What is the difference between `fs.readFile` and `fs.createReadStream`?	Advanced
39	How do you implement a scalable architecture in Node.js?	Advanced
40	What are Microtasks vs. Macrotasks?	Advanced
41	Explain the “Reactor Pattern” in Node.js.	Advanced
42	How does `util.promisify` work?	Advanced

No.	Questions	Level
43	What is the difference between TCP and HTTP in Node.js context?	Advanced
44	How can you secure a Node.js application against XSS and Injection?	Advanced
45	How does Node.js handle SSL/TLS?	Advanced
46	Explain the concept of “Buffers” relating to character encodings.	Advanced
47	How do you implement logging in a high-traffic Node.js app?	Advanced
48	What is the difference between `process.exit(0)` and `process.exit(1)`?	Advanced
49	How do you manage configuration across environments?	Advanced
50	How would you design a real-time chat application in Node.js?	Advanced

1. How does Node.js handle concurrency given that it is single-threaded?

Solution & Explanation:

Node.js uses an Event-Driven, Non-Blocking I/O model. While the main execution runs on a single thread (the Event Loop), Node.js offloads heavy I/O operations (like file reading, network requests, or database queries) to the system kernel or the libuv thread pool (written in C++).

When an asynchronous operation is initiated, Node.js registers a callback and moves on to the next task. Once the operation completes, the system signals Node.js, and the callback is pushed to the **Callback Queue** to be executed by the Event Loop.

```
// Example of non-blocking I/O
console.log("1. Start");
// This simulates a file read or network request taking 1 second
setTimeout(() => {
  console.log("2. Async Operation Done");
}, 1000);
```

```
console.log("3. End");

// Output:
// 1. Start
// 3. End
// 2. Async Operation Done
```

Common Mistakes:

- Assuming “single-threaded” means Node.js cannot do parallel processing (it does so via libuv for I/O).
- Blocking the main thread with heavy CPU calculations (e.g., large loops), which freezes the entire server.

Best Practices:

- Keep the main thread free for handling client requests.
- Offload CPU-intensive tasks to **Worker Threads** or separate microservices.

Practice Exercise: Write a script that reads a large file asynchronously vs. synchronously and measure the time difference in blocking the event loop.

Interview Scenario: “We have a high-traffic API. Why would Node.js be a better choice than a multi-threaded language like Java for handling thousands of concurrent simple requests?”

---

2. What is the difference between `process.nextTick()` and `setImmediate()`?

Solution & Explanation:

Both are used to schedule asynchronous code, but they fire at different times in the Event Loop:

- `process.nextTick()`: Fires **immediately** after the current operation completes, *before* the Event Loop continues to the next phase. It has higher priority than `setImmediate()`.

- `setImmediate()`: Fires in the **Check** phase of the Event Loop, after I/O callbacks are processed.

```
console.log("Start");

setImmediate(() => {
  console.log("setImmediate");
});

process.nextTick(() => {
  console.log("process.nextTick");
});

console.log("End");

// Output:
// Start
// End
// process.nextTick (Always runs before setImmediate)
// setImmediate
```

Common Mistakes:

- Using `process.nextTick()` recursively, which can starve the Event Loop (Preventing I/O operations from running).
- Thinking `setImmediate()` runs immediately (it actually runs on the *next* tick of the loop).

Best Practices:

- Use `setImmediate()` if you want to queue a callback behind pending I/O events.
- Use `process.nextTick()` only when you need to handle an error or execute something *before* the event loop proceeds.

Practice Exercise: Create a recursive function using `process.nextTick` and observe how it blocks I/O events (like a `setTimeout` not firing).

Interview Scenario: "I need to ensure a piece of code runs after the current function returns but before any other I/O events occur. Which function should I use?"

---

3. Explain the concept of “Stub” in Node.js testing.

Solution & Explanation:

A Stub is a dummy function or object used during testing to simulate the behavior of existing code. Stubs are used to:

1. Prevent side effects (e.g., prevent writing to a real database during tests).
2. Force specific code paths (e.g., force a function to throw an error to test error handling).

```
const sinon = require("sinon");
const fs = require("fs");

// We want to test a function that reads a file,
//but without actually reading from disk.
const readFileStub = sinon.stub(fs, "readFile");

// Force the stub to return a specific error
readFileStub.yields(new Error("File not found"), null);

fs.readFile("test.txt", (err, data) => {
  if (err) console.log("Caught expected error:", err.message);
});

// Restore the original function after test
readFileStub.restore();
```

Common Mistakes:

- Confusing Stubs (simulate behavior) with Mocks (verify behavior/expectations).
- Forgetting to restore the stub (using `.restore()`), which can break subsequent tests.

Best Practices:

- Use libraries like **Sinon.js** or **Jest** for clean stubbing.
- Always clean up stubs in the `afterEach` hook of your test runner.

Practice Exercise: Write a test for a function that makes an external API call. Stub the API call to return a 500 error and verify your function handles it gracefully.

Interview Scenario: “How would you test a function that charges a credit card without actually charging a real card every time the test runs?”

---

4. What is the purpose of `module.exports` vs. `exports`?

Solution & Explanation:

In Node.js CommonJS modules, module is the object representing the current module, and exports is a variable that points to module.exports.

- **`module.exports`**: The actual object that gets returned when you `require()` a module.
- **`exports`**: A shorthand reference to `module.exports`.

```
// file: math.js
// Valid: Attaching properties to exports

exports.add = (a, b) => a + b;

// Valid: Overwriting module.exports
module.exports = {
  add: (a, b) => a + b,
};

// INVALID: Breaking the reference

exports = { add: (a, b) => a + b };
// 'exports' no longer points to 'module.exports', so nothing is exported.
```

Common Mistakes:

- Assigning a new object directly to `exports` (e.g., `exports = { ... }`), which breaks the link to `module.exports`.

Best Practices:

- If you are exporting a single class or function, assign it directly to `module.exports`.

- If you are exporting multiple utility functions, attaching them to `exports.funcName` is fine.

Practice Exercise: Create a module that attempts to export an object using `exports = { ... }` and try to require it in another file to see why it fails.

Interview Scenario: "I tried to export my class using `exports = MyClass`, but when I require it, I get an empty object. Why?"

---

5. What is the "Error-First Callback" pattern?

Solution & Explanation:

The Error-First Callback pattern is a convention in Node.js where a callback function accepts error as its first argument and the result (data) as the second argument.

- If the operation fails, `error` contains the error object, and `result` is null.
- If the operation succeeds, `error` is null, and `result` contains the data.

```
const fs = require('fs');

fs.readFile('example.txt', 'utf8', (err, data) => {
    if (err) {
        // Check for error first
        console.error('Error reading file:', err);
        return; // Stop execution
    }

    // If no error, proceed with data
    console.log('File contents:', data);
});
```

Common Mistakes:

- Forgetting to `return` inside the `if (err)` block, causing the code to continue executing and potentially crash when accessing `data`.
- Throwing errors inside callbacks (which cannot be caught by a surrounding `try/catch`).

Best Practices:

- Always check `if (err)` immediately inside the callback.
- Consider wrapping legacy callback-based functions with `util.promisify` to use `async/await`.

Practice Exercise: Write a function that accepts a callback. If the input is negative, call the callback with an error. If positive, call it with `null` and the doubled value.

Interview Scenario: "Why does `fs.readFile` put the error object first in its callback? What happens if I don't handle it?"

---

6. What is the difference between `spawn` and `fork`?

Solution & Explanation:

Both are methods from the child_process module to create new processes.

- **`spawn()`**: Creates a new process to execute a command (e.g., system commands like `ls` or `git`). It streams data back (stdout/stderr). It does not create a dedicated V8 instance or communication channel by default.
- **`fork()`**: A special case of spawn specifically for creating **Node.js** processes. It creates a new V8 instance and establishes an IPC (Inter-Process Communication) channel, allowing the parent and child to send messages to each other using `.send()`.

```
const { spawn, fork } = require("child_process");

// Spawn example: Listing files

const ls = spawn("ls", ["-lh", "/usr"]);

// Fork example: Running a heavy calculation script

const child = fork("./heavy-script.js");

child.send({ start: true }); // Send message to child

child.on("message", (msg) => console.log("From child:", msg));
```

Common Mistakes:

- Using `fork` for non-Node.js commands (use `spawn` or `exec` instead).
- Spawning too many `fork` processes, as each one requires its own memory (approx 30MB+) and V8 instance.

Best Practices:

- Use `spawn` for external tools/shell commands.
- Use `fork` (or `worker_threads`) for heavy computation within Node.js.

Practice Exercise: Create a parent script that `forks` a child script. Have the parent send a number to the child, and the child return the factorial of that number.

Interview Scenario: "We need to run a Python script from our Node.js server. Should we use `fork` or `spawn`?"

---

7. What are the key features of the `package.json` file?

Solution & Explanation:

The package.json file is the manifest for a Node.js project. Key features include:

1. **Metadata**: Project name, version, description, author.
2. **Dependencies**: Lists libraries required for the app to run (`dependencies`) and libraries needed only for development (`devDependencies`).
3. **Scripts**: Custom command shortcuts (e.g., `npm start`, `npm test`).
4. **Main**: Entry point of the application (usually `index.js`).
5. **Engines**: Specifies which versions of Node.js or npm the project works with.

Common Mistakes:

- Manually editing versions in `package.json` without understanding Semantic Versioning (SemVer) symbols like `^` (compatible with minor) vs `~` (compatible with patch).

- Putting production dependencies (like `express`) into `devDependencies`.

Best Practices:

- Use `npm install --save` or `--save-dev` to update the file automatically rather than editing manually.
- Lock versions using `package-lock.json` to ensure consistency across environments.

Practice Exercise: Create a `package.json` script named “debug” that runs your app with the nodemon inspector enabled.

Interview Scenario: “If I clone your project and run `npm install`, how does npm know which versions of libraries to install?”

---

8. What is REPL?

Solution & Explanation:

REPL stands for Read-Eval-Print Loop. It is an interactive shell environment that comes with Node.js.

- **Read**: Reads user input (code).
- **Eval**: Evaluates the code.
- **Print**: Prints the result to the console.
- **Loop**: Loops back and waits for the next input.

You access it by typing `node` in your terminal without any arguments. It is useful for testing simple snippets or debugging.

Common Mistakes:

- Trying to write complex multi-file application logic entirely inside the REPL.
- Forgetting that variables defined in REPL are lost once you exit.

Best Practices:

- Use REPL for quick syntax checks or experimenting with standard library methods (e.g., checking how `path.join` works).
- Use the `_` variable in REPL to access the result of the last evaluated expression.

Practice Exercise: Open REPL, define a class, instantiate it, and call a method on it.

Interview Scenario: "How can I quickly check if a specific string method exists in the current Node.js version without creating a file?"

---

9. Explain the purpose of the `buffer` class.

Solution & Explanation:

The Buffer class handles binary data in Node.js. historically didn't handle binary streams well (it was designed for strings). Buffers are used to represent fixed-length sequences of bytes, which is essential for reading files, handling TCP streams, or processing image data.

```
// Create a buffer from a string
const buf = Buffer.from('Hello');
console.log(buf); // <Buffer 48 65 6c 6c 6f> (Hex representation)
console.log(buf.toString()); // 'Hello'
console.log(buf[0]); // 72 (ASCII for 'H')
```

Common Mistakes:

- Concatenating strings and Buffers without converting properly.
- Allocating unsafe memory using `Buffer.allocUnsafe()` without filling it immediately (can contain sensitive old data from memory).

Best Practices:

- Use `Buffer.alloc()` for safe, zero-filled buffers.
- Use `Buffer.from()` to convert data types.

Practice Exercise: Write a script that converts a UTF-8 string into a Base64 encoded buffer and prints it.

Interview Scenario: “We are receiving a raw image stream over a TCP socket. How do we store this data in memory before saving it to a file?”

---

10. What is Middleware?

Solution & Explanation:

Middleware functions are functions that have access to the request object (req), the response object (res), and the next middleware function in the application’s request-response cycle.

They can:

1. Execute code.
2. Modify the request and response objects.
3. End the request-response cycle.
4. Call `next()` to pass control to the next middleware.

```
const express = require('express');
const app = express();

// Simple logging middleware
const logger = (req, res, next) => {
    console.log(`${req.method} request to ${req.url}`);
    next(); // Pass control to the next handler
};

app.use(logger);

app.get('/', (req, res) => {
    res.send('Home Page');
});
```

Common Mistakes:

- Forgetting to call `next()`: This leaves the request hanging indefinitely.

- Sending a response (`res.send`) and then calling `next()` , causing an “Headers already sent” error.

Best Practices:

- Use middleware for cross-cutting concerns like logging, authentication, and parsing body data.
- Order matters: Place global middleware (like parsers) before route definitions.

Practice Exercise: Create a middleware that checks for a specific API key in the headers. If missing, return 401; otherwise, call `next()` .

Interview Scenario: “How would you implement a mechanism to log every incoming request’s timestamp without modifying every single route handler?”

---

11. What is the difference between `dependencies` and `devDependencies` ?

Solution & Explanation:

- **`dependencies`** : Packages required for the application to run in production (e.g., `express` , `mongoose` , `react`). Installed via `npm install <package>` .
- **`devDependencies`** : Packages needed only for development and testing (e.g., `jest` , `eslint` , `nodemon`). Installed via `npm install --save-dev <package>` .

When you run `npm install --production` (common in CI/CD pipelines), `devDependencies` are skipped.

Common Mistakes:

- Installing testing libraries or build tools in `dependencies` , which bloats the production build size.

Best Practices:

- Strictly separate run-time needs from build-time needs.

Practice Exercise: Initialize a project, install `lodash` as a dependency and `mocha` as a devDependency. Check `package.json` .

Interview Scenario: “We are deploying to AWS Lambda and want to keep our package size small. How does `package.json` help us exclude unnecessary testing libraries?”

---

12. How do you update dependencies in a Node.js project?

Solution & Explanation:

1. **`npm outdated`**: Lists packages that have newer versions available.
2. **`npm update`**: Updates packages to the latest version allowed by the SemVer rules in `package.json` (e.g., updates `1.2.1` to `1.2.9` but not `2.0.0`).
3. **`npm install <package>@latest`**: Forces an upgrade to the absolute latest version.

Common Mistakes:

- Running `npm update` and expecting major version upgrades (e.g., v4 to v5). npm respects the carets (`^`) in package.json.
- Deleting `package-lock.json` to “fix” issues (this defeats the purpose of deterministic builds).

Best Practices:

- Use tools like **Dependabot** or **Renovate** for automated updates.
- Always run tests after updating dependencies.

Practice Exercise: Install an old version of a library (e.g., `express@4.0.0`). Use `npm outdated` to see the latest, then update it.

Interview Scenario: “A security vulnerability was found in one of our dependencies. How do you safely update it?”

---

13. What is the global object in Node.js?

Solution & Explanation:

In browser , the global object is window. In Node.js, the global object is global.

Variables or functions attached to global are available everywhere in the application without requiring imports. Standard globals include setTimeout, console, process, and Buffer.

Common Mistakes:

- Polluting the `global` namespace with custom variables. This makes code hard to debug and causes naming collisions.
- Assuming `window` or `document` exists in Node.js (they don't).

Best Practices:

- **Avoid** attaching custom variables to `global`. Use module exports to share data.

Practice Exercise: Set a variable `global.myVar = 10` in one file and try to log it in another file without requiring the first one. (Do this to understand it, then never do it again in production).

Interview Scenario: "I have a configuration object I need everywhere. Should I attach it to `global`?" (Answer: No, export it as a module).

---

14. What are the core modules of Node.js?

Solution & Explanation:

Node.js comes with built-in modules that don't need to be installed via npm. Common ones include:

1. **`fs`**: File System (reading/writing files).
2. **`http` / `https`**: Creating servers and making requests.
3. **`path`**: Utilities for handling file paths.
4. **`events`**: The Event Emitter class.
5. **`os`**: Operating System information.

6. `crypto`: Cryptography (hashing, encryption).

Common Mistakes:

- Trying to `npm install fs` (it's built-in).
- Using string concatenation for paths instead of `path.join()`, which causes issues across Windows/Linux.

Best Practices:

- Prefer built-in modules over external libraries for simple tasks (e.g., use `fs` instead of a heavy file utility library if possible).

Practice Exercise: Create a script that uses `os` to print the total memory and `path` to print the current file's extension.

Interview Scenario: "Do I need to install a library to check the server's free memory?"

---

15. What is `npm`?

Solution & Explanation:

npm (Node Package Manager) is the default package manager for Node.js. It serves two purposes:

1. **CLI Tool**: Used to install, manage, and publish packages.
2. **Repository**: A massive online registry of open-source packages (npmjs.com).

Common Mistakes:

- Confusing `npm` (the tool) with `Node.js` (the runtime).
- Using `npm install -g` (global) for project-specific dependencies.

Best Practices:

- Use local installations for project dependencies.
- Use `.npmrc` files to save config settings for the project.

Practice Exercise: Search for a package on npmjs.com, install it, use it, and then uninstall it.

Interview Scenario: "What is the difference between installing a package globally vs. locally?"

---

16. Explain the "Event Loop" in detail.

Solution & Explanation:

The Event Loop is the mechanism that allows Node.js to perform non-blocking I/O operations. It has specific phases, and it cycles through them:

1. **Timers**: Executes callbacks from `setTimeout` and `setInterval`.
2. **Pending Callbacks**: Executes I/O callbacks deferred to the next loop iteration.
3. **Idle, Prepare**: Internal use only.
4. **Poll**: Retrieves new I/O events; executes I/O related callbacks.
5. **Check**: Executes `setImmediate()` callbacks.
6. **Close Callbacks**: Executes close events (e.g., `socket.on('close', ...)`).

Between every phase, Node.js checks the **Microtask Queue** (`process.nextTick` and Promises) and drains it completely.

Common Mistakes:

- Thinking the event loop runs on multiple threads (it's one thread).
- Blocking the loop with `JSON.parse` on large objects or complex Regex, which halts all phases.

Best Practices:

- Understand the phases to know when your code will execute.
- Monitor Event Loop Lag using tools like `perf_hooks` or APM solutions.

Practice Exercise: Create a script mixing `setTimeout`, `setImmediate`, `Promise.resolve`, and `process.nextTick` to predict and verify the output order.

Interview Scenario: "Why does a `Promise` resolve before a `setTimeout` even if the timeout is 0ms?"

---

17. What are "Streams" and what are the different types?

Solution & Explanation:

Streams are collections of data meant to be handled piece-by-piece (chunks) rather than all at once. This makes them memory-efficient for large data.

Types:

1. **Readable**: Source of data (e.g., `fs.createReadStream`, HTTP Request).
2. **Writable**: Destination for data (e.g., `fs.createWriteStream`, HTTP Response).
3. **Duplex**: Both readable and writable (e.g., TCP Sockets).
4. **Transform**: Duplex streams that modify data as it passes through (e.g., zlib compression).

```
const fs = require('fs');

const readable = fs.createReadStream('input.txt');
const writable = fs.createWriteStream('output.txt');

// Pipe reads from input and writes to output efficiently
readable.pipe(writable);
```

Common Mistakes:

- Reading a 2GB file into memory using `fs.readFile` (crashes the app).
- Forgetting to handle stream 'error' events (crashes the app).

Best Practices:

- Always use streams for file handling and network responses.

- Use `pipeline` (from `stream` module) instead of `.pipe()` for better error handling cleanup.

Practice Exercise: Create a Transform stream that takes text input, converts it to uppercase, and pipes it to stdout.

Interview Scenario: "A user uploads a 5GB video file. How do you save it to disk without exhausting the server's RAM?"

---

18. How do you handle errors in Promises vs. Callbacks?

Solution & Explanation:

- **Callbacks**: Use the "Error-First" pattern (`if (err) return...`).
- **Promises**: Use `.catch()` method.
- **Async/Await**: Use `try...catch` blocks.

```
// Promise
getData()
    .then(result => console.log(result))
    .catch(err => console.error(err));

// Async/Await
async function main() {
    try {
        const result = await getData();
        console.log(result);
    } catch (err) {
        console.error(err);
    }
}
```

Common Mistakes:

- Using `async/await` without `try/catch`, causing Unhandled Promise Rejections which can crash the process in newer Node versions.
- Nesting `.then()` calls (Promise Hell) instead of chaining them.

Best Practices:

- Prefer `async/await` for readability.
- Ensure a global `unhandledRejection` listener is set up as a safety net.

Practice Exercise: Convert a nested callback-based function ("Callback Hell") into a clean `async/await` structure.

Interview Scenario: "What happens if an error is thrown inside an async function and there is no try/catch block?"

---

19. What is "Callback Hell" and how do you avoid it?

Solution & Explanation:

Callback Hell is heavily nested callbacks (shaped like a pyramid) which make code unreadable and hard to debug.

Solutions:

1. **Modularization**: Break callbacks into separate named functions.
2. **Promises**: Chain operations (`.then().then()`).
3. **Async/Await**: Write async code linearly.
4. **Control Flow Libraries**: Libraries like `async.js` (less common now).

Common Mistakes:

- Mixing callbacks and Promises, leading to inconsistent error handling.

Best Practices:

- Adopt `async/await` as the standard.

Practice Exercise: Take a code snippet with 4 nested `fs.readFile` calls and refactor it using `fs.promises` .

Interview Scenario: "Refactor this pyramid of doom code on the whiteboard."

20. What is `process` in Node.js?

Solution & Explanation:

The process object provides information about, and control over, the current Node.js process. It is a global object.

- **`process.env`**: Environment variables.
- **`process.argv`**: Command line arguments.
- **`process.exit()`**: Exits the process.
- **`process.cwd()`**: Current working directory.
- **`process.memoryUsage()`**: Memory usage stats.

Common Mistakes:

- Hardcoding secrets instead of using `process.env`.
- Using `process.exit(0)` inside a library, unexpectedly killing the user's application.

Best Practices:

- Use libraries like `dotenv` to load environment variables into `process.env`.
- Handle `SIGTERM` signals via `process.on('SIGTERM')` for graceful shutdowns.

Practice Exercise: Write a script that prints "Hello [Name]" where [Name] is passed as a command line argument (using `process.argv`).

Interview Scenario: "How do we distinguish between Development and Production environments in code?" (Answer: `process.env.NODE_ENV`).

---

21. Explain `cluster` module in Node.js.

Solution & Explanation:

Since Node.js is single-threaded, it runs on one CPU core. The Cluster module allows you to create child processes (workers) that run simultaneously and share the same server port. This allows a Node.js app to utilize multi-core systems.

```
const cluster = require('cluster');
const http = require('http');
const numCPUs = require('os').cpus().length;

if (cluster.isMaster) {
    // Fork workers
    for (let i = 0; i < numCPUs; i++) {
        cluster.fork();
    }
} else {
    // Workers can share any TCP connection
    http.createServer((req, res) => {
        res.writeHead(200);
        res.end('Hello World');
    }).listen(8000);
}
```

Common Mistakes:

- Assuming memory is shared between workers (it is not; each has its own heap).
- Not handling worker crashes (the master needs to listen for 'exit' and re-fork).

Best Practices:

- Use process managers like **PM2** which handle clustering automatically, rather than writing raw cluster code.

Practice Exercise: Write a simple server, cluster it, and kill a worker process manually to see if the master respawns it.

Interview Scenario: "We have an 8-core server, but our Node app is only using 12% CPU. How do we fix this?"

---

22. What is `libuv`?

Solution & Explanation:

libuv is a multi-platform C++ library that provides support for asynchronous I/O based on event loops. It abstracts the operating system's non-blocking I/O operations (like epoll on Linux, kqueue on OSX, IOCP on Windows).

It manages the Thread Pool (default size 4) used for file system operations, DNS lookups, and crypto hashing.

Common Mistakes:

- Thinking itself executes in libuv (JS executes in V8; libuv handles the I/O plumbing).

Best Practices:

- If you have many blocking file operations, you can increase the thread pool size via `UV_THREADPOOL_SIZE`.

Practice Exercise: Change `UV_THREADPOOL_SIZE` to 1 and run multiple `crypto.pbkdf2` calls to see them execute sequentially instead of in parallel.

Interview Scenario: "What is the relationship between V8 and libuv?"

---

23. How does `require()` work under the hood?

Solution & Explanation:

When you call require('module'):

1. **Resolving**: Finds the absolute path of the file.
2. **Loading**: Loads the file content.
3. Wrapping: Wraps the code in an IIFE (Immediately Invoked Function Expression) to provide module scope:

(function(exports, require, module, **filename, **dirname) { ...code... })

4. **Evaluating**: Runs the code.
5. **Caching**: Caches the `module.exports` result so subsequent requires of the same file return the object immediately.

Common Mistakes:

- Circular dependencies (Module A requires B, B requires A), returning an incomplete object.
- Thinking code inside a module runs every time it is required (it runs only once due to caching).

Best Practices:

- Avoid circular dependencies.
- Use caching to your advantage for Singleton patterns.

Practice Exercise: Create a module that logs a message when it runs. Require it twice in another file. Observe that the log only appears once.

Interview Scenario: "If I require the same configuration file in 10 different places, does Node read the file from the disk 10 times?"

---

24. What are "Worker Threads"?

Solution & Explanation:

The worker_threads module enables the use of threads that execute in parallel. Unlike cluster (which uses processes), Workers share memory (via SharedArrayBuffer). They are useful for CPU-intensive tasks (image resizing, video compression, complex math) within the same process.

Common Mistakes:

- Using Workers for I/O tasks (standard Node async I/O is already more efficient for this).
- Creating a new Worker for every single request (high overhead).

Best Practices:

- Use a **Worker Pool** to reuse threads rather than creating/destroying them constantly.

Practice Exercise: Write a script that calculates Fibonacci numbers. Offload the calculation of a large Fibonacci number to a worker thread so the main thread stays responsive.

Interview Scenario: “How can we perform video encoding in Node.js without blocking the web server?”

---

25. How do you handle JWT Authentication in Node?

Solution & Explanation:

JSON Web Tokens (JWT) are used for stateless authentication.

1. **Login**: User sends credentials. Server verifies and signs a JWT with a secret key.
2. **Response**: Server sends the token back.
3. **Request**: Client sends the token in the `Authorization` header (`Bearer <token>`).
4. **Verify**: Middleware verifies the token signature. If valid, `req.user` is populated.

Common Mistakes:

- Storing sensitive data (like passwords) inside the JWT payload (it is only base64 encoded, not encrypted).
- Using a weak secret key.

Best Practices:

- Set a short expiration time (e.g., 15 mins) and use Refresh Tokens.
- Store JWTs in HttpOnly cookies to prevent XSS attacks.

Practice Exercise: Implement a simple Express route that generates a token and a middleware that protects another route by verifying that token.

Interview Scenario: “Why is JWT considered ‘stateless’ compared to session-based auth?”

---

26. What is the difference between `exec`, `execFile`, `spawn`, and `fork`?

Solution & Explanation:

- **`spawn`**: Streams data, no buffer limit. Best for large data or long-running processes.
- **`fork`**: Like spawn, but for Node.js modules. Adds IPC channel.
- **`exec`**: Runs a command in a shell. Buffers the output (has a max buffer size). Good for small output.
- **`execFile`**: Like exec, but runs a file directly (no shell). Slightly more efficient/secure.

Common Mistakes:

- Using `exec` for commands that produce huge output (crashes if buffer limit exceeded).
- Passing unsanitized user input to `exec` (Shell Injection vulnerability).

Best Practices:

- Default to `spawn` or `execFile`. Use `exec` only if you need shell features (like pipes `|` or redirects `>`).

Practice Exercise: Use `exec` to run `ls` and print the output. Then try `spawn` to do the same.

Interview Scenario: "I want to resize an image using ImageMagick CLI. Which function should I use?"

---

27. Explain `EventEmitter`.

Solution & Explanation:

The events module is the backbone of Node.js. Many objects (streams, http server) inherit from EventEmitter.

You can emit named events and register listeners for them.

```
const EventEmitter = require('events');
const myEmitter = new EventEmitter();

// Listener
myEmitter.on('greet', (name) => {
    console.log(`Hello ${name}`);
});

// Emitter
myEmitter.emit('greet', 'Alice');
```

Common Mistakes:

- Forgetting that events are synchronous by default. If a listener blocks, the emitter blocks.
- Not handling the 'error' event (causes the process to crash).

Best Practices:

- Always listen for 'error'.
- Use `.once()` if you only want to listen to the first occurrence.

Practice Exercise: Build a custom Logger class that extends EventEmitter and emits a 'log' event every time a message is logged.

Interview Scenario: "How does the HTTP server trigger your callback function when a request arrives?" (Answer: It emits a 'request' event).

---

28. How do you prevent callback nesting without using Promises?

Solution & Explanation:

Before Promises were standard, we used Named Functions (Modularization).

Instead of defining anonymous functions inline, define them separately and pass the function reference.

```
// Nested
fs.readFile('a.txt', (err, data) => {
```

```
    fs.writeFile('b.txt', data, (err) => { ... });
});

// Flattened
function handleWrite(err) { ... }

function handleRead(err, data) {
    if (err) return handleError(err);
    fs.writeFile('b.txt', data, handleWrite);
}
fs.readFile('a.txt', handleRead);
```

Common Mistakes:

- Losing the `this` context when passing object methods as callbacks.

Best Practices:

- While this is a valid technique, Modern Node.js developers should use `async/await`.

Practice Exercise: Take a nested callback function and refactor it using named functions.

Interview Scenario: "You are maintaining a legacy codebase running Node v4. How do you clean up the code?"

---

29. What is semantic versioning (SemVer)?

Solution & Explanation:

Version format: MAJOR.MINOR.PATCH (e.g., 1.5.2).

- **MAJOR**: Breaking changes.
- **MINOR**: New features (backward compatible).
- **PATCH**: Bug fixes (backward compatible).

`package.json` symbols:

- `^1.2.0`: Updates to `1.x.x` (Minor & Patch).

- `~1.2.0`: Updates to `1.2.x` (Patch only).

Common Mistakes:

- Upgrading Major versions without checking the changelog (breaks the app).

Best Practices:

- Understand `^` vs `~`. Use `npm ci` in production to install exact versions from the lockfile.

Practice Exercise: Explain what `^2.4.1` allows installing vs `~2.4.1`.

Interview Scenario: "Our build failed because a library updated automatically. How do we prevent this?"

---

30. How do you handle unhandled exceptions in Node.js?

Solution & Explanation:

You can listen to the uncaughtException event on the process object.

However, this is a last resort. The process is in an undefined state and should be restarted.

```
process.on('uncaughtException', (err) => {
console.error('CRITICAL ERROR:', err);
process.exit(1); // Force exit
});
```

Common Mistakes:

- Trying to "resume" the application after an uncaught exception. The system might be corrupted (e.g., locked database connections).

Best Practices:

- Log the error, perform synchronous cleanup, and let the process crash so the process manager (like PM2 or Kubernetes) can restart it fresh.

Practice Exercise: Create a script that throws an error, catch it with `uncaughtException`, log it, and exit.

Interview Scenario: "Is it safe to keep the server running after an `uncaughtException`?"

---

31. Explain the role of `NODE_ENV`.

Solution & Explanation:

NODE_ENV is an environment variable used to signal the environment type (e.g., development, production, test).

- In **Production**: Express.js caches view templates, generates less verbose error messages, and optimizations are enabled.
- In **Development**: More logging, no caching.

Common Mistakes:

- Forgetting to set `NODE_ENV=production` when deploying. This can make the app 3x slower.

Best Practices:

- Always set `NODE_ENV=production` on live servers.

Practice Exercise: Write a script that behaves differently (logs vs doesn't log) based on `NODE_ENV`.

Interview Scenario: "Why is my Express app 50% slower in production than on my local machine?" (Hint: check NODE_ENV).

---

32. How do you debug a Node.js application?

Solution & Explanation:

1. `console.log`: Simple tracing.

2. **Node Inspector**: Run `node --inspect index.js` and open `chrome://inspect` in Chrome DevTools to step through code.
3. **IDE Debuggers**: VS Code has a built-in debugger (configure `launch.json`).
4. `util.debuglog`: Conditional logging.

Common Mistakes:

- Leaving `console.log` statements in production code (pollutes logs and affects performance).

Best Practices:

- Use a proper logger like **Winston** or **Pino**.
- Use the Inspector for complex logic bugs.

Practice Exercise: Run a script with `--inspect-brk`, attach Chrome DevTools, and step through the code line by line.

Interview Scenario: "The server is crashing, but there are no error logs. How do you investigate?"

---

33. What is the difference between `null` and `undefined`?

Solution & Explanation:

- `undefined`: A variable has been declared but not assigned a value. It is the default value of uninitialized variables and function arguments.
- `null`: An assignment value. It is used to intentionally represent "no value" or "empty".

Common Mistakes:

- Checking `if (variable)` assuming it catches both (it does, but it also catches `0` and `false`).
- Assuming `typeof null` is 'null' (it is actually `'object'` - a legacy JS bug).

Best Practices:

- Use `null` when you want to explicitly clear a variable.
- Use strict equality (`===`) checks.

Practice Exercise: Create a function that returns `undefined` if argument is missing, and `null` if argument is explicitly invalid.

Interview Scenario: "If I query the database and no record is found, should I return null or undefined?"

---

34. What is CORS and how do you handle it in Node.js?

Solution & Explanation:

CORS (Cross-Origin Resource Sharing) is a browser security feature that restricts web pages from making requests to a different domain than the one that served the web page.

To allow it, the server must send specific headers (e.g., Access-Control-Allow-Origin).

In Express:

```
const cors = require('cors');
app.use(cors()); // Enable all CORS requests
```

Common Mistakes:

- Thinking CORS is a server-side security feature (it protects users/browsers, not the server).
- Allowing `*` (all origins) in a sensitive application.

Best Practices:

- Whitelist specific domains in production: `app.use(cors({ origin: 'https://myapp.com' }))` .

Practice Exercise: Create two servers on different ports. Try to fetch data from one to the other using client-side JS and observe the CORS error. Fix it using the `cors`

package.

Interview Scenario: “My React app on port 3000 cannot talk to my Node API on port 5000. Why?”

---

35. Explain the concept of “Backpressure” in Streams.

Solution & Explanation:

Backpressure occurs when the Readable stream (source) produces data faster than the Writable stream (destination) can consume it. This causes memory to fill up.

Node.js streams handle this automatically when using .pipe(). The writable stream signals the readable stream to pause until it catches up.

Common Mistakes:

- Manually reading ‘data’ events and writing them without checking `stream.write()` return value (which returns `false` if the buffer is full).

Best Practices:

- Use `.pipe()` or `pipeline()` which manages backpressure automatically.

Practice Exercise: Create a fast readable stream and a slow writable stream. Observe how Node.js pauses the reading.

Interview Scenario: “Why shouldn’t I just read a file and immediately write it to the response object using event listeners?”

---

36. How does the V8 Engine work?

Solution & Explanation:

V8 (by Google) compiles directly to native machine code.

1. **Parsing**: Converts code to Abstract Syntax Tree (AST).
2. **Ignition (Interpreter)**: Converts AST to Bytecode and executes it.

3. **TurboFan (Compiler)**: Takes “hot” (frequently used) functions from bytecode and optimizes them into machine code for faster execution.

4. **Garbage Collection**: Automatically frees memory (Orinoco).

Common Mistakes:

- Writing code that changes object “shapes” (adding/deleting properties dynamically), which de-optimizes the code in V8.

Best Practices:

- Initialize all object properties in the constructor to keep hidden classes consistent.

Practice Exercise: Research “V8 Hidden Classes” and write a script that demonstrates optimization/de-optimization.

Interview Scenario: “Why is it slower to delete properties from an object than to set them to null?”

---

37. Explain Memory Leaks in Node.js and how to debug them.

Solution & Explanation:

A Memory Leak happens when the application retains references to objects that are no longer needed, preventing the Garbage Collector from freeing them.

Causes:

- Global variables.
- Unclosed event listeners (Observer pattern).
- Closures holding onto large scopes.

Debugging:

1. Start Node with `--inspect`.
2. Take **Heap Snapshots** in Chrome DevTools.
3. Compare snapshots over time to see what objects are accumulating.

Common Mistakes:

- Forgetting to remove listeners (`removeListener`) when a connection closes.

Best Practices:

- Use `WeakMap` or `WeakSet` if you don't want to prevent garbage collection.
- Stress test your app to identify leaks early.

Practice Exercise: Create a deliberate memory leak by pushing objects to a global array every 10ms. Analyze it using Chrome Heap Snapshot.

Interview Scenario: "Our server crashes with 'Heap Out of Memory' every 2 days. What is your debugging strategy?"

---

38. What is the difference between `fs.readFile` and `fs.createReadStream`?

Solution & Explanation:

- `fs.readFile` : Loads the **entire** file into memory (Buffer) and then fires the callback. Fast for small files, catastrophic for large files.
- `fs.createReadStream` : Reads the file in **chunks** (default 64kb). Memory usage remains constant regardless of file size.

Common Mistakes:

- Using `readFile` for serving static assets (videos, large logs) in a web server.

Best Practices:

- Default to streams (`createReadStream`) for anything that could potentially be larger than a few MBs.

Practice Exercise: Monitor process memory usage while reading a 100MB file using both methods.

Interview Scenario: "How would you deliver a 2GB CSV file to a client via API?"

---

39. How do you implement a scalable architecture in Node.js?

Solution & Explanation:

1. **Microservices**: Split the monolith into smaller services.
2. **Load Balancing**: Use Nginx or HAProxy to distribute traffic across multiple Node instances.
3. **Clustering**: Use PM2 to utilize all CPU cores.
4. **Caching**: Use Redis to cache frequent DB queries.
5. **Message Queues**: Use RabbitMQ or Kafka for asynchronous communication between services.

Common Mistakes:

- Keeping state (sessions, websocket connections) in the process memory. (Use Redis instead so state is shared across scaled instances).

Best Practices:

- Build “Stateless” applications (12-Factor App methodology).

Practice Exercise: Design a system diagram for a Node.js chat app that needs to handle 100k concurrent users.

Interview Scenario: “How do we scale our socket.io server across multiple machines?” (Answer: Redis Adapter).

---

40. What are Microtasks vs. Macrotasks?

Solution & Explanation:

- **Macrotasks (Tasks)**: `setTimeout`, `setInterval`, `setImmediate`, I/O operations.
- **Microtasks**: `process.nextTick`, `Promise` callbacks.

Priority: The Event Loop processes **all** available Microtasks after **every single** Macrotask (and before moving to the next phase). This means Microtasks can starve the Event Loop

if they recursively trigger new Microtasks.

Common Mistakes:

- Assuming `setTimeout(fn, 0)` runs before a `Promise.resolve().then(fn)` (Promise runs first).

Best Practices:

- Be careful with recursive Promises or nextTicks.

Practice Exercise: Predict the output of a script mixing `setTimeout`, `setImmediate`, `Promise`, and `process.nextTick`.

Interview Scenario: "Why does my `Promise` callback run before my `setTimeout`?"

---

41. Explain the "Reactor Pattern" in Node.js.

Solution & Explanation:

The Reactor Pattern is the design pattern Node.js uses to handle non-blocking I/O.

1. **Resources** (I/O) emit events.
2. **Event Demultiplexer** (libuv) listens to these resources and pushes events to the Event Queue.
3. Event Loop iterates the queue and executes the associated Handler (callback).

This allows a single thread to manage thousands of concurrent connections.

Common Mistakes:

- Blocking the Reactor (Event Loop) destroys the pattern's efficiency.

Best Practices:

- Keep callbacks lightweight.

Practice Exercise: Draw the Reactor Pattern flow on a whiteboard.

Interview Scenario: "Explain the theoretical model behind Node's non-blocking I/O."

42. How does `util.promisify` work?

Solution & Explanation:

It converts a function that follows the “Error-First Callback” style into a function that returns a Promise.

It effectively wraps the original function, returning a Promise that resolves if the callback gets data, and rejects if the callback gets an error.

```
const util = require('util');
const fs = require('fs');
const readFileAsync = util.promisify(fs.readFile);

// Now you can use await
await readFileAsync('file.txt');
```

Common Mistakes:

- Trying to promisify functions that don’t follow the `(err, value)` callback standard.

Best Practices:

- Use `fs.promises` instead of manual promisification for the `fs` module.

Practice Exercise: Create a custom function with a callback and promisify it manually (wrapper) vs using `util.promisify`.

Interview Scenario: “I have a legacy library using callbacks. How do I use it with async/await?”

---

43. What is the difference between TCP and HTTP in Node.js context?

Solution & Explanation:

- **TCP (`net` module)**: The transport layer protocol. It establishes a raw connection (stream of bytes). No headers, no methods (GET/POST). Fast and low-level.

- **HTTP (`http` module)**: The application layer protocol built **on top** of TCP. It adds structure (Headers, Body, Status Codes).

Common Mistakes:

- Using HTTP for real-time gaming (too much overhead; use TCP or UDP).

Best Practices:

- Use HTTP for APIs/Websites.
- Use TCP (via WebSockets or raw sockets) for real-time, low-latency needs.

Practice Exercise: Create a raw TCP server using `net.createServer` that echoes back whatever data it receives.

Interview Scenario: "Why would we use the `net` module instead of `express`?"

---

44. How can you secure a Node.js application against XSS and Injection?

Solution & Explanation:

1. **XSS (Cross-Site Scripting)**: Sanitize user input (remove HTML tags). Use libraries like `xss` or `dompurify`. Set **CSP (Content Security Policy)** headers using `helmet`.
2. **SQL/NoSQL Injection**: Never concatenate strings for queries. Use **Parameterized Queries** (SQL) or ODMs like Mongoose (which sanitizes mostly by default).

Common Mistakes:

- Trusting user input.
- Using `eval()` or `new Function()`.

Best Practices:

- Use **Helmet.js** middleware (`app.use(helmet())`) to set security headers automatically.

Practice Exercise: Create a simple form that accepts text and displays it. Try to inject a `<script>alert(1)</script>`. Then fix it using a sanitizer.

Interview Scenario: “How does `helmet` middleware improve security?”

---

45. How does Node.js handle SSL/TLS?

Solution & Explanation:

The https module allows creating secure servers. It requires a private key and a public certificate.

Node.js uses OpenSSL for TLS.

In production, SSL is usually terminated at the Reverse Proxy level (Nginx/AWS ALB) rather than in the Node.js app itself, to save CPU.

Common Mistakes:

- Committing private keys to GitHub.
- Handling SSL encryption in the Node process (slows down the single thread).

Best Practices:

- Offload SSL termination to Nginx or a Load Balancer.

Practice Exercise: Generate a self-signed certificate and create an HTTPS server in Node.js.

Interview Scenario: “Why do we put Nginx in front of Node.js for HTTPS?”

---

46. Explain the concept of “Buffers” relating to character encodings.

Solution & Explanation:

Buffers store raw binary data. When converting Buffer to String, an encoding is needed (default is UTF-8).

If you receive a stream of data, a multi-byte character (like emoji 🚀) might get split between two chunks. Using buffer.toString() on a partial chunk produces garbage characters ().

Solution: Use StringDecoder module, which handles incomplete multi-byte characters correctly.

Common Mistakes:

- Converting streams to strings chunk-by-chunk without `StringDecoder`.

Best Practices:

- Always be aware of encoding when transforming binary data.

Practice Exercise: Split a buffer containing a multi-byte character in half and try to decode the halves separately vs using StringDecoder.

Interview Scenario: "Why are my emojis showing up as question marks in the data stream?"

---

47. How do you implement logging in a high-traffic Node.js app?

Solution & Explanation:

1. **Structured Logging**: Log as JSON (using **Pino** or **Bunyan**), not plain text. This allows tools like ELK Stack or Datadog to parse logs.
2. **Levels**: Use Debug, Info, Warn, Error properly.
3. **Async Logging**: Writing to console/file is blocking (synchronous). Use logging libraries that write to a stream asynchronously or ship logs to a sidecar process.

Common Mistakes:

- Using `console.log` (it blocks the event loop when writing to stdout in some TTY contexts).
- Logging sensitive data (passwords/tokens).

Best Practices:

- Use **Pino** (fastest logger).

- Log to stdout and let the container orchestrator (Docker/K8s) handle the file writing.

Practice Exercise: Configure Pino to log to a file and rotate the file daily.

Interview Scenario: "Why is `console.log` bad for production?"

---

48. What is the difference between `process.exit(0)` and `process.exit(1)`?

Solution & Explanation:

- **0**: Success. The process finished normally.
- **1 (or non-zero)**: Error. The process crashed or failed.
- **Uncaught Exception**: Default exit code is 1.

CI/CD pipelines and Docker containers rely on these codes to know if a build/deploy failed.

Common Mistakes:

- Using `process.exit(0)` when an error occurred, causing the CI pipeline to think everything passed.

Best Practices:

- Always use `1` for errors.

Practice Exercise: Write a script that exits with code 1. Run it in terminal `node script.js && echo "Success"`. Observe "Success" is NOT printed.

Interview Scenario: "Our Docker container restarts successfully even though the app failed to connect to the DB. Why?" (Answer: You might be exiting with code 0).

---

49. How do you manage configuration across environments?

Solution & Explanation:

Use Environment Variables.

Library: dotenv (for local dev) or config / custom-env.

Do NOT commit .env files.

Structure:

- `default.json` (Common config)
- `production.json` (Overrides)
- Environment variables override everything.

Common Mistakes:

- Committing passwords to config files.
- Complex `if-else` logic in code to check environments.

Best Practices:

- The “12-Factor App” config principle: Store config in the environment.

Practice Exercise: Set up a project using the `config` package and override a value using a command-line environment variable.

Interview Scenario: “How do we rotate database passwords without changing the code?”

---

50. How would you design a real-time chat application in Node.js?

Solution & Explanation:

Architecture:

1. **Protocol**: WebSockets (using Socket.io or **ws**).
2. **Server**: Node.js (excellent for holding open connections).
3. **Scaling**: Multiple Node instances.

4. **Synchronization**: **Redis Pub/Sub**. When User A sends a message to Server 1, Server 1 publishes it to Redis. Server 2 (where User B is connected) subscribes and pushes the message to User B.

5. **Persistence**: MongoDB or Cassandra for storing chat history.

Common Mistakes:

- Storing socket connections in a local array (fails when scaling to multiple servers).

Best Practices:

- Use Redis Adapter for Socket.io.

Practice Exercise: Build a basic chat app with Socket.io that broadcasts messages to all connected clients.

Interview Scenario: “Design WhatsApp backend using Node.js.”