Node js interview questions

Let’s start with the **5 most common and must-know Node.js interview questions** with **answers and explanations**. These are frequently asked by interviewers to test your core understanding of Node.js.

✅ 1. What is Node.js? How is it different from traditional web server technologies like Apache?

Answer:
Node.js is an **open-source, cross-platform JavaScript runtime environment** that executes JavaScript code outside of a web browser, using the **V8 engine** (from Chrome).

Differences:

- Node.js uses **non-blocking, event-driven architecture** which makes it **lightweight and efficient**.

- Traditional web servers like Apache follow a **multi-threaded** approach, whereas Node.js uses a **single-threaded** event loop to handle multiple clients.

Why it matters:
Understanding Node's architecture is critical in explaining how it handles concurrent requests efficiently without creating multiple threads.

✅ 2. What is the Event Loop in Node.js? How does it work?

Answer:
The **Event Loop** is the core of Node.js's asynchronous programming model. It allows Node.js to perform **non-blocking I/O operations** — even though it runs in a **single thread**.

Working:

1. Node.js registers async operations (like file reading or HTTP requests) and continues executing.

2. When the operation completes, the callback is pushed to the **event queue**.

3. The Event Loop picks callbacks from the queue and executes them when the call stack is empty.

Why it matters:
Interviewers want to know if you understand how Node handles async behavior without threads.

✅ 3. What is the difference between `process.nextTick()`, `setImmediate()`, and `setTimeout()` in Node.js?

Answer:

Function	Execution Timing
`process.nextTick()`	Executes **before** the next event loop iteration
`setImmediate()`	Executes on the **next iteration** of the event loop
`setTimeout(fn, 0)`	Executes after a **minimum delay of 0ms**, not immediate

Example:

```
setTimeout(() => console.log("timeout"), 0);
setImmediate(() => console.log("immediate"));
process.nextTick(() => console.log("nextTick"));
```

Output:

```
nextTick
timeout or immediate (order not guaranteed)
```

Why it matters:
This tests your understanding of the Node.js concurrency model and microtasks/macrotasks.

✅ 4. What are Streams in Node.js? What are the types?

Answer:
Streams are used to handle **reading/writing large volumes of data** efficiently (e.g., files, network data).

Types of Streams:

1. **Readable** – can read data (e.g., `fs.createReadStream`)
2. **Writable** – can write data (e.g., `fs.createWriteStream`)
3. **Duplex** – can read and write (e.g., TCP socket)
4. **Transform** – duplex but can also modify data (e.g., `zlib.createGzip()`)

Why it matters:
Efficient data handling using streams is a critical performance technique in backend systems.

✅ 5. What is middleware in Express.js? How does it work?

Answer:
Middleware in Express.js is a **function that has access to**:

- `req` (request object)
- `res` (response object)
- `next` (function to pass control to the next middleware)

Syntax:

```
app.use((req, res, next) => {
  console.log("Middleware executed");
  next(); // pass to next middleware/route
});
```

Types of middleware:

- Application-level
- Router-level
- Error-handling
- Built-in (e.g., `express.json()`)
- Third-party (e.g., `cors`, `body-parser`)

Why it matters:
Middleware is a fundamental concept in building Node.js APIs with Express.

✅ 6. What is the difference between synchronous and asynchronous functions in Node.js?

Answer:

- **Synchronous functions** block the execution of code. Each operation waits for the previous one to complete.

- **Asynchronous functions** allow code execution to continue while waiting for operations (like reading files or making HTTP requests) to complete.

Example:

```
// Synchronous
const fs = require('fs');
const data = fs.readFileSync('file.txt', 'utf8');
console.log(data);

// Asynchronous
fs.readFile('file.txt', 'utf8', (err, data) => {
  if (err) throw err;
  console.log(data);
});
```

Why it matters:
Efficient backend systems rely on non-blocking (asynchronous) operations to handle multiple requests at once.

✅ 7. What are the different ways to handle asynchronous code in Node.js?

Answer:

Callbacks

```
 fs.readFile('file.txt', callback);
```

1.

Promises

```
 fs.promises.readFile('file.txt')
  .then(data => console.log(data))
  .catch(err => console.error(err));
```

2.

Async/Await

```
 async function readFile() {
  try {
    const data = await fs.promises.readFile('file.txt');
    console.log(data);
  } catch (err) {
    console.error(err);
  }
```

}

3.

Why it matters:
Understanding these patterns shows you're comfortable with real-world asynchronous code, essential for APIs, database calls, etc.

✅ 8. What is a callback hell and how can it be avoided?

Answer:
Callback Hell refers to deeply nested callbacks, making the code hard to read and maintain.

Example of callback hell:

```
doSomething(function(result) {
  doSomethingElse(result, function(newResult) {
    doThirdThing(newResult, function(finalResult) {
      console.log(finalResult);
    });
  });
});
```

How to avoid it:

- Use **named functions**
- Use **Promises**
- Use **Async/Await** (most preferred)

Why it matters:
Clean and readable async code is a sign of a good developer.

✅ 9. What is the difference between `require()` and `import` in Node.js?

Answer:

Feature	`require()`	`import`
Syntax	CommonJS	ES6 Modules
Support	Default in Node.js	Needs `"type": "module"` in `package.json`

Dynamic Import	Not supported natively	Supported via `import()`

Example:

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

// ES6 Modules
import fs from 'fs';
```

Why it matters:
Modern Node.js supports both module systems. Interviewers test your understanding of module loading.

✅ 10. How does Node.js handle child processes, and why would you use them?

Answer:
Node.js can run child processes using the `child_process` module to perform heavy computations or run scripts in parallel.

Types of child process methods:

- `exec` – runs a command in a shell, buffers the output
- `spawn` – launches a new process with a stream interface
- `fork` – used to create a new Node.js process (specially for JS files)

Example:

```
const { exec } = require('child_process');

exec('ls -l', (error, stdout, stderr) => {
  if (error) {
    console.error(`Error: ${error.message}`);
    return;
  }
  console.log(`Output: ${stdout}`);
});
```

Why it matters:
Shows you understand how to scale Node apps or manage system-level tasks efficiently.

Awesome! Let's continue with the **next 5 Node.js interview questions (11 to 15)**. These are slightly more advanced and show your deeper understanding of backend development with Node.js.

✅ 11. What is clustering in Node.js and why is it used?

Answer:
Node.js runs in a **single-threaded** environment, but modern servers have **multi-core CPUs**.

Clustering allows you to **create child processes** (workers) that run simultaneously and share the same server port, enabling load balancing and better CPU utilization.

Example:

```
const cluster = require('cluster');
const http = require('http');
const os = require('os');

if (cluster.isMaster) {
  const cpuCount = os.cpus().length;
  for (let i = 0; i < cpuCount; i++) {
    cluster.fork(); // create a worker
  }
} else {
  http.createServer((req, res) => {
    res.end(`Handled by process ${process.pid}`);
  }).listen(3000);
}
```

Why it matters:
It shows your knowledge of scaling Node.js applications in production environments.

✅ 12. How is error handling done in Express.js?

Answer:
In Express, error-handling middleware is defined by having **four arguments**: `(err, req, res, next)`.

Example:

```
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).send('Something broke!');
});
```

You can also use `try...catch` with `async/await` in route handlers.

Why it matters:
Interviewers test your ability to write stable and production-safe code.

✅ 13. What is the difference between `spawn()` and `fork()` in Node.js?

Answer:

Function	Description
`spawn()`	Launches a new process with a given command. It is used for non-Node scripts or binaries.
`fork()`	Specifically used to create new **Node.js processes**. It is a special case of `spawn()` with an IPC (Inter-Process Communication) channel.

Example:

```
// spawn
const { spawn } = require('child_process');
const ls = spawn('ls', ['-lh']);

// fork
const { fork } = require('child_process');
const child = fork('child.js');
```

Why it matters:
It shows you know how to manage subprocesses in Node.js effectively.

✅ 14. What is the purpose of `package.json` in a Node.js project?

Answer:
`package.json` is the **metadata file** for any Node.js project.

It includes:

- Project name, version, description
- Scripts (e.g., `start`, `test`)
- Dependencies (`dependencies`, `devDependencies`)
- Entry point (`main`)
- Author, license, etc.

Example:

```
{
  "name": "myapp",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": {
    "start": "node index.js"
  },
  "dependencies": {
    "express": "^4.18.2"
  }
}
```

Why it matters:
A good Node developer knows how to manage and understand project structure.

✅ 15. What are environment variables and how do you use them in Node.js?

Answer:
Environment variables are used to store **sensitive or configuration data** like API keys, DB URLs, etc.

You access them via `process.env` in Node.js.

Usage:

1. Create a `.env` file:

```
PORT=3000
DB_URL=mongodb://localhost:27017/mydb
```

2. Load using `dotenv`:

```
require('dotenv').config();
console.log(process.env.PORT);
```

Why it matters:
Using env variables is a best practice in deploying secure and configurable applications.

Great! Since you said **"yes"**, I'll continue with the **next 5 Node.js interview questions (16–20)** — more advanced and practical, often asked in real-world interviews.

✅ 16. What is middleware chaining in Express.js? How does `next()` work?

Answer:
Middleware chaining means **executing multiple middleware functions** in sequence for a single request.

- Each middleware uses `next()` to pass control to the next one.
- If `next()` is not called, the request **hangs**.

Example:

```
app.use((req, res, next) => {
  console.log('Middleware 1');
  next();
});

app.use((req, res, next) => {
  console.log('Middleware 2');
  next();
});

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

Output:

```
Middleware 1
Middleware 2
Final Handler
```

Why it matters:
Middleware chaining is at the heart of how Express handles requests — critical knowledge for routing and error handling.

✅ 17. What is the difference between `app.use()` and `app.get()` in Express.js?

Answer:

Method	Purpose
`app.use()`	Used to define **middleware functions** for **all HTTP methods** and **paths**, or for a specific base path.
`app.get()`	Used to handle **GET requests** to a specific route.

Example:

```
app.use('/api', middlewareFunction); // runs on all HTTP methods for routes starting with /api

app.get('/home', (req, res) => {
  res.send('Welcome to Home!');
});
```

Why it matters:
You must know when to use each based on whether you're defining middleware or route-specific logic.

✅ 18. What is `require.cache` and how can you clear a cached module in Node.js?

Answer:
Node.js **caches modules** after the first `require()` call to improve performance.

You can access the cache with `require.cache`.

To remove a module from cache:

```
delete require.cache[require.resolve('./myModule')];
```

Why it matters:
Shows you understand advanced behaviors — useful when dynamically reloading modules or testing.

✅ 19. What is the use of the `global` object in Node.js?

Answer:
`global` is the **global namespace object** in Node.js (like `window` in the browser). Variables attached to `global` are accessible everywhere.

Example:

```
global.appName = 'MyApp';

console.log(appName); // MyApp
```

However, using `global` is discouraged in large applications due to the risk of conflicts.

Why it matters:
Knowing about `global` helps you understand scope and avoid bad practices.

✅ 20. How do you handle file uploads in Node.js?

Answer:
To handle file uploads, you typically use the `multer` middleware with Express.js.

Setup Example:

```
npm install multer
```

Code Example:

```
const multer = require('multer');
const upload = multer({ dest: 'uploads/' });

app.post('/upload', upload.single('file'), (req, res) => {
  res.send('File uploaded!');
});
```

Why it matters:
Handling files is a common backend task (e.g., profile images, documents, etc.).

Perfect 👍 Let’s prepare the **first 5 interview-style questions with clear, professional answers**.

---

1. What is the difference between synchronous and asynchronous code in JavaScript?

✅ **Answer:**

- **Synchronous code** executes line by line, blocking further execution until the current line completes.
- **Asynchronous code** allows other tasks to run while waiting for a long-running operation (like fetching data) to finish.

📌 **Example:**

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

setTimeout(() => {
  console.log("Async Task");
}, 1000);

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

Output:

```
Start
End
Async Task
```

---

2. How does the JavaScript event loop work?

✅ **Answer:**
The **event loop** allows JavaScript (which is single-threaded) to handle asynchronous tasks.

- **Call Stack:** Executes synchronous code.
- **Web APIs:** Handle async tasks (e.g., `setTimeout`, `fetch`).
- **Callback Queue (Macrotasks):** Stores callbacks like `setTimeout`.
- **Microtask Queue:** Stores promise callbacks (`.then`, `await`).
- **Event Loop:** Continuously checks if the call stack is empty and pushes tasks from queues.

📌 **In short:** *"The event loop manages execution order between synchronous code, microtasks, and macrotasks."*

---

3. What are microtasks and macrotasks in JavaScript?

✅ **Answer:**

- **Macrotasks:** Executed after the current script finishes. Examples: `setTimeout`, `setInterval`, `setImmediate`.
- **Microtasks:** Executed immediately after the current task, before macrotasks. Examples: `Promise.then`, `queueMicrotask`.

📌 **Example:**

```
setTimeout(() => console.log("Macrotask"), 0);
Promise.resolve().then(() => console.log("Microtask"));
console.log("Sync");
```

Output:

```
Sync
Microtask
Macrotask
```

---

4. What are callbacks, and what is callback hell?

✅ **Answer:**

- A **callback** is a function passed as an argument to another function, executed later when the task is complete.

- **Callback hell** happens when callbacks are deeply nested, making code difficult to read and maintain.

📌 **Example of callback hell:**

```
getUser(1, function(user) {
  getPosts(user.id, function(posts) {
    getComments(posts[0], function(comments) {
      console.log(comments);
    });
  });
});
```

👉 Solution: Use **Promises** or **async/await** for cleaner code.

---

5. Difference between `setTimeout(fn, 0)` and `Promise.resolve().then(fn)`?

✅ **Answer:**

- `setTimeout(fn, 0)` → puts the callback in the **macrotask queue**, so it runs **after all microtasks are completed**.

- `Promise.resolve().then(fn)` → puts the callback in the **microtask queue**, so it runs **immediately after the current synchronous code finishes**.

📌 **Example:**

```
setTimeout(() => console.log("Timeout"), 0);
Promise.resolve().then(() => console.log("Promise"));
console.log("End");
```

Output:

```
End
Promise
Timeout
```

Great 👍 Let's continue with the **next 5 questions (6–10) with answers**.

---

6. How do Promises work internally?

✅ **Answer:**
A **Promise** represents a value that may be available now, later, or never.
It has **3 states**:

- `pending` → initial state
- `fulfilled` → operation completed successfully (`resolve`)
- `rejected` → operation failed (`reject`)

📌 **Example:**

```
const promise = new Promise((resolve, reject) => {
  setTimeout(() => resolve("Done!"), 1000);
});

promise.then(result => console.log(result));
```

Output after 1s:

```
Done!
```

---

7. What happens if you don't add a `.catch()` to a rejected Promise?

✅ **Answer:**
If a Promise is **rejected** and has no `.catch()`, JavaScript throws an **UnhandledPromiseRejectionWarning** (in Node.js) or logs an error in the browser.

📌 **Example:**

```
Promise.reject("Error occurred");
// Without .catch(), it throws an unhandled rejection error
```

Best practice → Always handle rejections with `.catch()` or `try...catch` in async/await.

---

8. Promise.all vs Promise.allSettled vs Promise.race vs Promise.any

✅ **Answer:**

- **Promise.all([p1, p2])** → Resolves when **all** promises resolve. Rejects if **any one** fails.
- **Promise.allSettled([p1, p2])** → Resolves when **all settle** (fulfilled or rejected). Never rejects.
- **Promise.race([p1, p2])** → Resolves/rejects with the **first promise** that settles.
- **Promise.any([p1, p2])** → Resolves with the **first fulfilled promise** (ignores rejections).

📌 **Example:**

```
Promise.all([Promise.resolve(1), Promise.reject("fail")])
  .then(console.log)
  .catch(console.error);  // "fail"
```

---

9. What will be the output of this code?

```
console.log("start");
setTimeout(() => console.log("timeout"), 0);
Promise.resolve().then(() => console.log("promise"));
console.log("end");
```

✅ **Answer:**
Output:

```
start
end
promise
timeout
```

📌 **Reason:**

1. `"start"` → synchronous.
2. `"end"` → synchronous.
3. `"promise"` → microtask (runs before macrotasks).

4. `"timeout"` → macrotask.

---

10. How does `async/await` work under the hood?

✅ **Answer:**

- `async/await` is **syntactic sugar** over Promises.
- An `async` function always returns a Promise.
- `await` pauses execution until the Promise resolves/rejects, then continues.

📌 **Example:**

```
async function fetchData() {
  console.log("Fetching...");
  const data = await Promise.resolve("Data received");
  console.log(data);
}
fetchData();
```

Output:

```
Fetching...
Data received
```

Perfect 👍 Let's continue with the **next set (11–15)**.

---

11. What happens if you forget to use `await` inside an async function?

✅ **Answer:**
If you forget `await`, the function will return a **Promise object** instead of the resolved value.

📌 **Example:**

```
async function test() {
  return Promise.resolve("Hello");
}

async function run() {
  const result = test();  // Forgot await
  console.log(result);    // Promise { 'Hello' }
}
run();
```

👉 To get `"Hello"`, use:

```
const result = await test();
```

---

12. How do you handle errors in `async/await`?

✅ **Answer:**
 There are **two main ways**:

1. **try...catch block**

```
async function fetchData() {
  try {
    let data = await Promise.reject("Error occurred");
    console.log(data);
  } catch (err) {
    console.error("Caught:", err);
  }
}
fetchData();
```

2. **`.catch()` on the returned Promise**

```
fetchData().catch(err => console.error(err));
```

---

13. Difference between sequential and parallel execution with `await`?

✅ **Answer:**

- **Sequential:** Each `await` waits for the previous one.

```
let a = await foo();
let b = await bar();  // Runs after foo finishes
```

- **Parallel (faster):** Run both simultaneously.

```
let [a, b] = await Promise.all([foo(), bar()]);
```

👉 **Tip:** Use `Promise.all` when tasks are independent.

---

14. Why is JavaScript single-threaded but still asynchronous?

✅ **Answer:**

- JavaScript runs in a **single thread** (only one call stack).
- Asynchronous behavior comes from **Web APIs (in browsers)** or **libuv (in Node.js)**, which handle tasks like timers, I/O, or network requests.
- The **event loop** then schedules callbacks back into the call stack when ready.

📌 **In short:**

> JS itself is single-threaded, but async features are powered by external APIs and the event loop.

---

15. What is the difference between concurrency and parallelism in JavaScript?

✅ **Answer:**

- **Concurrency:** Ability to handle multiple tasks at the same time (not literally running together, but interleaved via the event loop).
- **Parallelism:** True simultaneous execution on multiple threads/cores.

📌 Example:

- JS uses **concurrency** with async tasks.
- To achieve **parallelism**, we use **Web Workers** (browser) or **Worker Threads** (Node.js).

Perfect 👍 Let's continue with **questions 16–20**.

---

16. How would you implement your own `Promise.all()`?

✅ **Answer:**
You can create a function that takes an array of promises, waits for all to resolve, and rejects if any fail.

📌 **Example:**

```
function myPromiseAll(promises) {
  return new Promise((resolve, reject) => {
    let results = [];
    let completed = 0;

    promises.forEach((p, i) => {
      Promise.resolve(p)
        .then(value => {
          results[i] = value;
          completed++;
          if (completed === promises.length) resolve(results);
        })
        .catch(reject);
    });
  });
}


// Usage
myPromiseAll([Promise.resolve(1), Promise.resolve(2)])
  .then(console.log);  // [1, 2]
```

---

17. What happens if you `await` inside a for-loop? How to optimize it?

✅ **Answer:**

- **Problem:** `for` + `await` runs **sequentially**, which can be slow for independent tasks.

```
for (let url of urls) {
  await fetch(url);  // Waits for previous fetch to finish
}
```

- **Optimized:** Use `Promise.all` to run in parallel:

```
await Promise.all(urls.map(url => fetch(url)));
```

---

18. Explain `process.nextTick()` vs `setImmediate()` in Node.js

✅ **Answer:**

- `process.nextTick()` → runs **before any I/O or timers**, high priority, even before microtasks.
- `setImmediate()` → runs **after I/O events** in the next iteration of the event loop (macrotask).

📌 **Example:**

```
process.nextTick(() => console.log("nextTick"));
setImmediate(() => console.log("setImmediate"));
console.log("sync");
```

Output:

```
sync
nextTick
setImmediate
```

---

19. Why is `forEach` not good with async/await?

✅ **Answer:**

- `forEach` **does not wait** for async functions.
- Using `await` inside `forEach` will not pause the loop.

📌 **Example:**

```
[1,2,3].forEach(async (num) => {
  await new Promise(res => setTimeout(res, 1000));
  console.log(num);
});
```

```
// All async callbacks start simultaneously
```

Solution: Use `for...of` or `Promise.all`:

```
for (let num of [1,2,3]) {
  await new Promise(res => setTimeout(res, 1000));
  console.log(num);
}
```

---

20. Tricky Output Question

```
async function test() {
  console.log(1);
  await Promise.resolve();
  console.log(2);
}
console.log(3);
test();
console.log(4);
```

✅ **Answer:**
Output:

```
3
1
4
2
```

📌 **Reason:**

1. 3 → synchronous
2. 1 → inside async function, runs synchronously until first await
3. 4 → synchronous
4. 2 → after Promise resolves (microtask)