NodeJS Technical Questions & Answers

Top Node.js Interview Questions & Answers for 10+ Years Experienced Candidates

If you're preparing for a **senior-level interview (10+ years) in Node.js**, expect deep technical questions covering **architecture, performance optimization, security, event-driven programming, and scalability**. Here are the **most commonly asked questions with detailed answers**:

1️⃣ What is Node.js and why is it used?

✅ **Answer:**
Node.js is an open-source, cross-platform **JavaScript runtime** built on Chrome's **V8 engine**.

✔ **Why is it used?**

- **Asynchronous & Event-Driven** – Handles multiple requests efficiently.
- **Non-blocking I/O Model** – Ideal for real-time applications.
- **Scalable & Lightweight** – Uses a single-threaded event loop.
- **Cross-platform** – Runs on Windows, Linux, and macOS.

✔ **Common Use Cases:**

- **RESTful APIs & Microservices**
- **Real-time Applications (e.g., chat apps, stock trading apps)**
- **Streaming Services (e.g., Netflix, Spotify)**
- **Server-side Rendering (SSR) for React, Angular, Vue.js apps**

2️⃣ How does the Node.js event loop work?

✅ **Answer:**
The **event loop** in Node.js is a **single-threaded** loop that handles asynchronous operations using a **non-blocking, event-driven model**.

✔ **Event Loop Phases:**

1. **Timers** – Executes `setTimeout()` and `setInterval()`.
2. **Pending Callbacks** – Executes callbacks from I/O operations.
3. **Idle & Prepare** – Internal processes (used by Node.js).
4. **Poll** – Retrieves new I/O events (file, network, etc.).
5. **Check** – Executes `setImmediate()` callbacks.
6. **Close Callbacks** – Handles `close` events (e.g., socket closure).

🔹 **Why is the event loop important?**

- Avoids **blocking the main thread**.
- Handles **thousands of concurrent requests efficiently**.
- Improves **performance in I/O-heavy applications**.

3️⃣ What is the difference between process.nextTick() and setImmediate()?

✅ **Answer:**
Both are used for scheduling **asynchronous execution**, but they run in different phases of the event loop.

Feature	`process.nextTick()`	`setImmediate()`
Execution Phase	**Microtask Queue**	**Check Phase**

Priority	Runs **before** the next event loop cycle starts	Runs **after** the poll phase
Use Case	Critical tasks (e.g., error handling)	I/O-related operations

✔ **Example:**

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

process.nextTick(() => console.log("Next Tick"));

setImmediate(() => console.log("Set Immediate"));

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

Output:

pgsql
CopyEdit
```
Start
End
Next Tick
Set Immediate
```

4️⃣What is the difference between synchronous and asynchronous programming in Node.js?

✅ **Answer:**

Type	Synchronous	Asynchronous
Blocking?	Yes	No
Executes in order?	Yes	No (continues execution)
Performance	Slower	Faster

Example	File System `fs.readFileSync()`	File System `fs.readFile()`

✔ **Example:**

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

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

🔹 **Which one to use?**

- Use **synchronous code** for simple tasks.
- Use **asynchronous code** for I/O-bound operations.

5️⃣ What are streams in Node.js?

✅ **Answer:**
Streams are objects that **handle large chunks of data** in a **continuous manner**, improving performance for large file handling.

✔ **Types of Streams:**

1. **Readable Streams** → Data flows **from** the source (e.g., `fs.createReadStream()`).
2. **Writable Streams** → Data flows **to** the destination (e.g., `fs.createWriteStream()`).
3. **Duplex Streams** → Both readable & writable (e.g., TCP sockets).

4. **Transform Streams** → Data transformation (e.g., compression).

✔ Example – Reading a File using Streams:

js
CopyEdit
```
const fs = require("fs");
const readStream = fs.createReadStream("largeFile.txt");

readStream.on("data", (chunk) => {
  console.log("Received chunk:", chunk.toString());
});
```

🔹 **Why use streams?**

- **Efficient memory usage** for large files.
- Reduces **buffering issues**.

6️⃣ How do you handle errors in Node.js?

✅ **Answer:**
✔ 1. Using Try-Catch (Synchronous Code)

js
CopyEdit
```
try {
  let result = someFunction();
} catch (error) {
  console.error("Error:", error.message);
}
```

✔ 2. Handling Errors in Callbacks (Asynchronous Code)

js
CopyEdit
```
fs.readFile("file.txt", "utf8", (err, data) => {
  if (err) {
    console.error("Error:", err.message);
    return;
  }
  console.log(data);
});
```

✔ 3. Using Promises (Async/Await)

js
CopyEdit
```
const readFile = async () => {
  try {
    const data = await fs.promises.readFile("file.txt", "utf8");
    console.log(data);
  } catch (error) {
    console.error("Error:", error.message);
  }
};
```

🔹 **Best practices:**

- Always **handle errors in async functions**.
- Use **middleware for error handling in Express.js**.

7️⃣ How do you scale a Node.js application?

✅ **Answer:**

✔ 1. Clustering (Multi-Core Usage)

- Use the `cluster` module to spawn worker processes.

js
CopyEdit
```
const cluster = require("cluster");
const http = require("http");
const numCPUs = require("os").cpus().length;

if (cluster.isMaster) {
  for (let i = 0; i < numCPUs; i++) {
    cluster.fork();
  }
} else {
  http.createServer((req, res) => {
    res.writeHead(200);
    res.end("Hello World");
  }).listen(8000);
}
```

✔ 2. Load Balancing with PM2

- Use **PM2** for process management.

sh
CopyEdit
```
pm2 start app.js -i max
```

✔ 3. Using Redis for Caching

- Reduces database load by caching frequent requests.

✔ 4. Microservices Architecture

- Break down the app into smaller services.

🔹 Why scale Node.js?

- **Handles more users efficiently**.
- **Improves application reliability**.

8️⃣ What are middleware functions in Express.js?

✅ **Answer:**
Middleware functions **intercept and modify** requests and responses before reaching the final route handler.

✔ **Example Middleware:**

js
CopyEdit
```js
const express = require("express");
const app = express();

app.use((req, res, next) => {
  console.log("Middleware executed");
  next(); // Pass control to the next middleware
});

app.get("/", (req, res) => {
  res.send("Hello, World!");
});

app.listen(3000);
```

🔹 **Common Middleware Types:**

- **Application-Level Middleware** (`app.use()`)
- **Router-Level Middleware** (`router.use()`)
- **Built-in Middleware** (`express.json()`, `express.static()`)

Final Thoughts

As a **10+ years experienced developer**, focus on **Node.js internals, performance optimization, security, and real-world scenarios**.

Most Used NPM Packages in Node.js Projects and Their Use Cases

NPM Package	Use Case
express	Fast, minimal web framework for building APIs and web apps.
dotenv	Loads environment variables from a `.env` file.
mongoose	ODM (Object-Document Mapping) for MongoDB to interact with databases easily.
cors	Enables Cross-Origin Resource Sharing (CORS) in APIs.
nodemon	Automatically restarts the server during development when file changes are detected.
jsonwebtoken (JWT)	Implements JSON Web Tokens (JWT) for authentication.
bcryptjs	Hashes and verifies passwords securely.
axios	Makes HTTP requests to external APIs (alternative to Fetch API).
helmet	Enhances security by setting various HTTP headers.
morgan	Logs HTTP requests for debugging and monitoring.

body-parser	Parses incoming request bodies (now built into Express).
multer	Handles file uploads in Node.js.
express-validator	Validates and sanitizes user input in Express applications.
ws	Provides WebSocket functionality for real-time communication.
socket.io	Enables real-time, bidirectional communication between client and server.
passport	Middleware for authentication strategies like OAuth, Google, Facebook, etc.
async	Provides utilities for handling asynchronous operations more efficiently.
lodash	Offers utility functions for manipulating arrays, objects, and strings.
moment (deprecated)	Used for date/time manipulation (replaced by `date-fns` and `luxon`).
date-fns	Modern, lightweight library for date and time operations.
winston	Logging library for structured logging and debugging.
chalk	Colors terminal output for better visibility.

compression	Compresses HTTP responses using Gzip for performance optimization.
pm2	Process manager for Node.js apps, ensuring uptime and monitoring.
sequelize	ORM for SQL databases like MySQL, PostgreSQL, and SQLite.
uuid	Generates unique identifiers (UUIDs).
concurrently	Runs multiple npm scripts in parallel (useful for monorepos or microservices).
cross-env	Sets environment variables across different platforms (Windows/Linux/Mac).