Node.js Interview Q&As (Part 2)

🔹 Section 1: Advanced Asynchronous Concepts

Q1. What is the difference between process.nextTick() and setImmediate()?

Answer:

- **process.nextTick()** schedules a callback **immediately after the current function finishes** but before the event loop continues.
- **setImmediate()** schedules callbacks to execute **in the check phase of the event loop**, after I/O events are processed.

Example:

```
js

console.log("Start");
process.nextTick(() => console.log("nextTick"));
setImmediate(() => console.log("setImmediate"));
console.log("End");
// Output: Start → End → nextTick → setImmediate
```

💡 **Tip:** Always mention → "nextTick has higher priority; overusing it can block I/O."

Q2. How does the Node.js event loop actually work (phases)?

Answer: The event loop runs in phases: timers (setTimeout/setInterval), pending callbacks, idle/prepare, poll (I/O), check (setImmediate), and close callbacks. Between phases, microtasks (Promises, process.nextTick) run. Understanding phases helps predict callback order and optimize non-blocking behavior.

💡 **Tip**: In interviews, sketch phases and mention where timers, I/O, and setImmediate execute.

Q3. What are Worker Threads in Node.js, and when to use them?

Answer: Worker Threads run JavaScript in parallel threads inside the same process and can share memory via SharedArrayBuffer. Use them for CPU-bound tasks (image processing, heavy computations) where blocking the event loop is unacceptable. For simple subprocess work or isolation, child processes may be preferable.

```
const { Worker } = require('worker_threads');
```

```
new Worker('./worker-task.js');
```

💡 Tip: “Workers share memory (faster) but add complexity — use for CPU-heavy loads.”

Q4. Difference between spawn, exec, and fork in child_process?

Answer: spawn streams stdout/stderr for long-running processes and is memory-efficient. exec buffers the whole output (good for small commands) and returns it in a callback. fork spawns a Node process with an IPC channel for messaging — ideal to run other Node scripts and communicate.

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

```
spawn('ls'); exec('ls', (e,o)=>{}); fork('worker.js');
```

💡 **Tip for Interviews**:

- spawn → continuous/large data (streaming).
- exec → small output (buffered),quick commands.
- fork → run another Node.js file and exchange messages,Node-to-Node IPC.

🔹 Section 2: File System & Events

Q5. How does Node.js handle file system operations (sync vs async)?
Answer: Synchronous FS calls block the event loop until complete (bad for servers). Asynchronous FS calls use libuv thread pool and callbacks/promises so the event loop stays free. Use async APIs (fs.promises or callbacks) in production; sync ops are only OK for startup scripts.

```js
// Async
const fs = require('fs/promises');
await fs.readFile('file.txt', 'utf8');
// Sync (blocks)
const data = require('fs').readFileSync('file.txt', 'utf8');
```

💡 Tip: “Always prefer async FS in request handlers to avoid blocking other requests.”

Q6. What is the EventEmitter in Node.js?
Answer: EventEmitter is the base class for event-driven programming in Node — objects emit named events and listeners respond. It’s used across Node (streams, servers). You can add/remove listeners and check listener counts to manage memory/behavior.

```js
const { EventEmitter } = require("events");
const eventBus = new EventEmitter();

// Listener
eventBus.on("order", item => console.log("Order received:", item));

// Emit event
eventBus.emit("order", "Pizza");
// Output: Order received: Pizza
```

💡 Tip: “Remove listeners and set max listeners to avoid memory leaks.”

Q7. How do you create custom events in Node.js?
Answer: Create a class extending EventEmitter or instantiate EventEmitter, then emit() events and register listeners with on()/once(). Use once() for one-time handlers and remove listeners when no longer needed.

```js
class MyBus extends require('events'){}
const bus = new MyBus();
bus.on('jobDone', msg=>console.log(msg));
bus.emit('jobDone','done');
```

💡 Tip: “Use namespaced or well-documented event names to avoid collisions.”

🔹 Section 3: Express.js (Advanced)

Q8. What is the difference between app.use() and app.all()?
Answer: app.use() mounts middleware for all HTTP methods and optionally a path prefix (it runs for matching paths). app.all() registers a route handler that matches **all HTTP methods** for a specific path. Use use() for middleware (logging, auth) and all() for route-specific catch-alls.

```
app.use('/api', authMiddleware); // middleware
app.all('/health', (req,res)=>res.send('ok')); // any method
```

💡 Tip: “Remember use doesn’t require exact path; all is route-specific.”

Q9. How do you handle file uploads in Node.js/Express?
Answer: Use middleware like multer (multipart/form-data). Configure storage (memory vs disk), validate file types/sizes, and sanitize filenames. Stream uploads to storage (S3) for large files instead of saving on local disk in production.

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

```
app.post('/upload', multer().single('file'), (req,res)=>res.send(req.file));
```

💡 Tip: “Validate file size and type; prefer streaming to cloud for scalability.”

Q10. How do you handle CORS in Express apps?
Answer: Use the cors middleware to set **Access-Control-Allow-Origin** and other CORS headers. Configure allowed origins, methods, credentials, and preflight responses securely — avoid using '*' in production unless truly public.

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

```
app.use(cors({ origin: 'https://example.com', credentials: true }));
```

💡 Tip: “Be explicit with origins and allow credentials only when necessary.”

🔹 Section 4: Testing & Debugging

Q11. How do you test a Node.js app (Mocha/Jest/Supertest)?
Answer: Unit test modules with Jest or Mocha+Chai. Use Supertest for integration tests hitting Express endpoints (in-memory). Mock external services (DB, APIs) and run tests in CI with coverage checks.

```
const request = require('supertest');
```

```
await request(app).get('/users').expect(200);
```

💡 Tip: “Write unit tests for logic + integration tests for routes; use CI to run them.”

Q12. How do you debug Node.js apps (node inspect, VSCode debugger)?
Answer: Use node --inspect and Chrome DevTools or the VSCode debugger to set breakpoints, step through code, and inspect

variables. For production, use flamegraphs (clinic), and console + logs with correlation IDs for tracing.

node --inspect-brk app.js

 # open chrome://inspect

💡 Tip: “Use source maps for transpiled code (TypeScript/Babel) to debug properly.”

🔹 Section 5: Performance & Optimization

Q13. What is middleware chaining and how does it affect performance?

Answer: In Express, multiple middleware run in sequence using next(). If you add too many or do heavy work inside middleware, requests slow down. Keep middleware light and modular.

Example:

```js
app.use((req, res, next) => { console.time("req"); next(); console.timeEnd("req"); });
app.use((req, res, next) => { console.log("Auth checked"); next(); });
```

💡 Tip: “Profile middleware order — move cheap checks earlier and expensive work later or offload.”

Q14. How does Node.js manage memory, and what is garbage collection?

Answer: V8 manages memory (heap + stack). GC (mark-and-sweep) frees objects without references. Large heaps, retained references, or global caches can prevent GC and cause OOM. Use heap snapshots and profiling to find leaks.

💡 Tip: “Know how to take heap snapshots (Chrome DevTools) and identify detached DOM or listener leaks.”

Q15. How do you profile and optimize a Node.js app?
Answer: Use profilers like clinic.js, node --inspect, or 0x to capture CPU/heap profiles. Identify hot functions, blocking calls, and memory leaks; then optimize algorithms, use streams, caching, or move CPU work to workers. Load-test with tools (k6, Artillery) before and after changes.
💡 Tip: “Measure first (profiling), fix the hotspot, then re-measure.”

🔹 Section 6: Security & Production

Q16. How do you prevent sensitive data leaks in Node.js apps?
Answer: Keep secrets out of code (.env + secret managers), use TLS in transit, encrypt sensitive data at rest, minimize logging of secrets, and apply least-privilege to services. Use SCA tools to detect credentials in repos.
💡 Tip: “Use secret managers (AWS Secrets Manager/HashiCorp Vault) in production.”

Q17. What is Helmet in Express, and why use it?
Answer: helmet is middleware that sets secure HTTP headers (HSTS, XSS protection, content security policy helpers). It’s an easy baseline to reduce common web vulnerabilities by instructing browsers how to behave.

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

app.use(helmet());
```

💡 Tip: “Helmet is a quick security win — customize CSP for complex apps.”

Q18. How do you implement rate limiting in Express?
Answer: Use middleware like express-rate-limit to throttle requests per IP or user, combine with IP-based caching/Redis for distributed limits, and apply different limits on auth endpoints vs public endpoints.

```
const rateLimit = require('express-rate-limit');
```

```
app.use(rateLimit({ windowMs: 60e3, max: 100 }));
```

💡 Tip: “Use Redis-backed store for rate limits in multi-instance setups.”

Q19. How do you secure cookies and sessions in Node.js?
Answer: Use HttpOnly, Secure, SameSite flags for cookies, store session data server-side or in Redis, rotate session IDs on login, and set short expirations. Avoid storing sensitive data in cookies.

```
res.cookie('sid', token, { httpOnly:true, secure:true, sameSite:'Lax' });
```

💡 Tip: “HttpOnly prevents XSS from reading cookies; Secure requires HTTPS.”

Q20. What are best practices for deploying Node.js apps in production?
Answer: Use process managers (PM2) or container orchestration (Docker + Kubernetes), enable logging/metrics/tracing (ELK/Prometheus/Grafana), set health/readiness probes, run multiple replicas behind a load balancer, and use CI/CD with blue/green or rolling deploys for zero downtime.
💡 Tip: “Automate CI/CD and include smoke tests and health checks before promoting releases.”