ChatGPT

Node.js – The Complete Interview Guide

Node.js is a popular JavaScript runtime built on Chrome's V8 engine, enabling JavaScript to run on the server and outside web browsers [1]. It is **event-driven**, **non-blocking**, and uses an asynchronous I/O model, which makes it lightweight and efficient for scalable network applications. Key features include:

- **V8 JavaScript Engine:** Executes JS code at high speed.
- **Event-Driven, Single-Threaded Architecture:** Uses an event loop and callbacks to handle many concurrent clients without multi-threading [2].
- **Non-Blocking I/O:** File, network, and other I/O operations run asynchronously, so the main thread isn't blocked while waiting for data.
- **Rich Package Ecosystem:** Uses **npm** (Node Package Manager) to install, manage, and share libraries and tools.

These features allow Node.js to handle high-throughput and real-time applications efficiently.

Architecture and Event Loop

Node.js follows a **single-threaded event loop** model. Although there is only one main JavaScript thread, heavy I/O and CPU tasks are offloaded to worker threads in the libuv thread pool, keeping the main thread free to handle new events. The **event loop** continuously checks for pending events or callbacks and executes them. The basic phases include: timers, pending callbacks, idle/poll, check, and close callbacks [2]. For example, `setTimeout()` callbacks run in the **Timers** phase, while I/O callbacks run in **Pending Callbacks**. This architecture allows Node to handle thousands of concurrent connections with minimal overhead [2].

```
// Pseudo-diagram of event loop phases (Timers -> I/O Poll -> Check -> Close)
// Each phase processes callbacks until the queue is empty.

while (true) {
  processTimersQueue();
  processPendingCallbacks();
  waitForIOOrTimers();  // idle if nothing pending
  processCheckQueue();
  processCloseQueue();
}
```

Modules and `require`

Node.js uses the **CommonJS module system**. Each file is a module with its own scope. To use a module, the `require()` function is used. Node will:

1. **Resolve the Module:** If it's a core module (`http`, `fs`, etc.), it's loaded immediately. Otherwise, Node looks in `node_modules` folders and resolves files by name or path (e.g. `require('./utils')`).
2. **Load and Wrap:** The module code is wrapped in a function to provide `exports`, `require`, `module`, `__filename`, and `__dirname` variables.
3. **Execute Once & Cache:** The module code runs, and its `module.exports` is cached. Subsequent `require()` calls return the cached export (avoiding repeated executions).

Example:

```
// greet.js
function sayHello(name) {
  console.log(`Hello, ${name}!`);
}
module.exports = { sayHello };

// main.js
const greet = require('./greet');
greet.sayHello('Node');  // Hello, Node!
```

`module.exports` vs `exports`

- `module.exports` is the actual object returned by `require()`. You assign to it to export a function/class/object.
- `exports` is a shortcut reference to `module.exports`. You can add properties to `exports` (e.g. `exports.foo = 1`) but if you reassign `exports` itself (e.g. `exports = {}`) it breaks the reference.

Example:

```
// Correct:
module.exports = { a: 1 };
module.exports.b = 2;
// Or using exports shorthand:
exports.c = 3;

// Wrong: reassigning exports breaks the link
exports = { d: 4 };  // This does NOT change module.exports
```

It's generally safer to use `module.exports` when exporting a new object or function.

npm and Package Management

npm (Node Package Manager) is the default package manager for Node.js. It is installed automatically with Node.js. npm helps manage dependencies and scripts for your project. Key points about npm:

- **package.json:** A file that declares your project name, version, dependencies, scripts, and more. It allows reproducible installs of needed packages.
- **Install dependencies:** `npm install <pkg>` adds packages to `node_modules` and records them in `package.json`. Using `--save` saves them under `dependencies`.
- **Global vs Local:** Global installs (`npm install -g`) make tools available system-wide (e.g. `nodemon`), while local installs are project-specific.
- **Scripts:** You can define custom scripts in `package.json` (e.g. `"start": "node app.js"`) and run them with `npm run start`.

NPM greatly simplifies sharing and reusing libraries. It's the primary tool for Node package management [3].

Asynchronous Patterns: Callbacks, Promises, Async/Await

Callbacks

Node.js heavily uses **callbacks** for async work. A callback is a function passed as an argument to be invoked later. Traditional Node-style callbacks take an error-first pattern: `function(err, result)`. Example:

```
const fs = require('fs');
fs.readFile('file.txt', 'utf8', (err, data) => {
  if (err) {
    console.error('File read error:', err);
    return;
  }
  console.log('File contents:', data);
});
```

The downside of callbacks is "callback hell" – deeply nested callbacks making code hard to read.

Promises

A **Promise** represents an eventual result (or failure) of an async operation. They improve readability and chainability. With a promise, you can write:

```
const fs = require('fs').promises;  // Node 10+
fs.readFile('file.txt', 'utf8')
  .then(data => {
    console.log('File contents:', data);
  })
```

```
  .catch(err => {
    console.error('Read error:', err);
  });
```

Promises avoid deep nesting by allowing `.then()` chaining. They can also be created manually:

```
function asyncTask() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve('Done!');
    }, 1000);
  });
}
asyncTask()
  .then(result => console.log(result))  // "Done!"
  .catch(err => console.error(err));
```

Async/Await

`async` / `await` is syntactic sugar on top of promises that makes asynchronous code look synchronous. Inside an `async` function, you can `await` a promise:

```
async function readFileAsync() {
  try {
    const fs = require('fs').promises;
    const data = await fs.readFile('file.txt', 'utf8');
    console.log('Contents:', data);
  } catch (err) {
    console.error('Error:', err);
  }
}
readFileAsync();
```

This reduces boilerplate and improves clarity. In summary, JavaScript offers **callbacks, Promises, and async/await** for async operations [4], with async/await being the modern, cleaner approach.

Events and EventEmitter

Node.js is **event-driven**. The built-in `events` module provides the `EventEmitter` class, which is fundamental to Node's architecture. An `EventEmitter` can emit named events that other code can listen for. Example:

```
const EventEmitter = require('events');
class MyEmitter extends EventEmitter {}
const emitter = new MyEmitter();

emitter.on('greet', (name) => {
  console.log(`Hello, ${name}!`);
});

emitter.emit('greet', 'Alice');  // "Hello, Alice!"
```

In this code, `emitter.on('greet', ...)` registers a listener, and `emitter.emit('greet', 'Alice')` triggers it. The `EventEmitter` class underpins many Node APIs (streams, servers, etc.) [5]. Custom events allow decoupling of code and handling asynchronous events conveniently.

Buffers and Streams

Buffers

Buffers are Node.js's way to handle **binary data**. A `Buffer` is like a fixed-size array of bytes outside V8's managed heap [6]. They are useful when dealing with binary file data or streams. Example:

```
const buf = Buffer.from('Hello, world!');
console.log(buf);           // <Buffer 48 65 6c 6c 6f 2c 20 77 6f 72 6c 64 21>
console.log(buf.toString()); // "Hello, world!"
```

Buffers are used internally by streams and file operations. They allow you to work with raw bytes, convert between encodings, etc.

Streams

Streams are **objects for reading/writing data continuously**, instead of all at once [7]. There are four types: Readable, Writable, Duplex (both), and Transform (duplex that modifies data). Streams are used for efficient handling of large data (e.g. file I/O, HTTP requests).

Example – reading a file with a stream:

```
const fs = require('fs');
const readStream = fs.createReadStream('large-file.txt', 'utf8');

readStream.on('data', (chunk) => {
  console.log('Received chunk:', chunk.length);
});

readStream.on('end', () => {
```

```
  console.log('Finished reading file.');
});
```

Or piping streams:

```
const writeStream = fs.createWriteStream('copy.txt');
readStream.pipe(writeStream);
```

This reads from `large-file.txt` and writes to `copy.txt` chunk by chunk. Key stream methods and events: `.pipe()`, `.on('data')`, `.on('end')`, etc. Streams and buffers often work together: streams output data as `Buffer` chunks when in non-encoded mode.

File System (`fs` Module)

The `fs` module provides APIs to interact with the file system [8]. You can perform both asynchronous and synchronous operations:

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

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

// Synchronous read (blocking)
try {
  const data = fs.readFileSync('data.txt', 'utf8');
  console.log('Sync read:', data);
} catch (err) {
  console.error(err);
}
```

Similarly, there are `writeFile`, `appendFile`, `unlink` (delete), `readdir` (list directory), `stat` (get file info), and many more. Use asynchronous methods (`readFile`, `writeFile`, streams, or `fs.promises`) to avoid blocking the event loop. The `fs` module is essential for file manipulation, configuration, and serving static assets [8].

HTTP Servers

Node's core `http` module allows creating web servers without external libraries [9]. For example:

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

const server = http.createServer((req, res) => {
  // req: http.IncomingMessage, res: http.ServerResponse
  console.log(`${req.method} ${req.url}`);
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello from Node.js!\n');
});

server.listen(3000, () => {
  console.log('Server running at http://localhost:3000/');
});
```

This creates a basic HTTP server listening on port 3000. You can inspect `req.method`, `req.url`, headers, etc., and write to `res` to send a response. The `createServer()` callback runs for each incoming request [9]. To handle JSON, you'd collect data from `req.on('data')` and parse it. For HTTPS, Node provides the similar `https` module with TLS support. For larger apps, most developers use frameworks like Express on top of `http`.

Middleware (Express Context)

While plain Node.js core doesn't have a built-in middleware concept, frameworks like Express use **middleware** extensively. Middleware are functions that process `request` and `response` objects sequentially. For instance, in Express:

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

app.use((req, res, next) => {
  console.log('Middleware 1');
  next();  // Pass to next middleware
});

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

app.listen(3000);
```

Here, each middleware has access to `req`, `res`, and a `next()` function. Middleware can execute code, modify `req`/`res`, end the response, or pass control forward [10]. Common middleware tasks include

logging, authentication, parsing JSON (`express.json()`), and error handling. Understanding middleware is important when using Express or similar frameworks.

Environment Variables

Environment variables allow configuration separate from code. In Node.js, the `process.env` object holds these variables. For example, you might set `PORT=5000` in the environment and in code do:

```
const port = process.env.PORT || 3000;
console.log(`Server will run on port ${port}`);
```

This way, `PORT` can be configured per environment (development, production, etc.). It's common to use a `.env` file for local development and the **dotenv** package to load it. Example usage:

```
# .env file
DB_HOST=localhost
DB_USER=myuser
DB_PASS=secret
```

```
require('dotenv').config();  // Loads .env into process.env
console.log(process.env.DB_HOST);  // "localhost"
```

Node.js provides `process.env` globally [11]. Managing environment-specific settings via env vars and dotenv (or similar tools) is a best practice [11] [12].

Error Handling

In Node, errors are handled differently for synchronous and asynchronous code:

- **Callbacks:** Use the error-first callback pattern. Always check for `err` in callbacks and handle it.
- **Promises/Async:** Use `.catch()` on promises or `try/catch` inside async functions.

Example with async/await:

```
async function getData() {
  try {
    const fs = require('fs').promises;
    const data = await fs.readFile('file.txt', 'utf8');
    console.log(data);
  } catch (err) {
    console.error('Error reading file:', err);
```

```
  }
}
```

Avoid throwing errors outside callbacks (as they'll crash the process). For uncaught exceptions, Node has `process.on('uncaughtException', handler)` and `unhandledRejection` for promise rejections. However, best practice is to handle errors where they occur and let the process exit on unexpected errors (often restarting via a process manager). Use custom error classes or codes for clarity, and always validate inputs. In Express, an error-handling middleware (with signature `(err, req, res, next)`) is used to catch errors in the request chain.

Child Processes

Node's `child_process` module lets you spawn subprocesses. Common methods include `spawn`, `exec`, and `fork` [13]. For example, using `spawn` to run a system command:

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

ls.stdout.on('data', (data) => {
  console.log(`stdout: ${data}`);
});
ls.stderr.on('data', (data) => {
  console.error(`stderr: ${data}`);
});
ls.on('close', (code) => {
  console.log(`child process exited with code ${code}`);
});
```

- `spawn` launches a new process and returns streams for input/output (non-blocking).
- `exec` runs a command in a shell and buffers the output (useful for short outputs).
- `fork` is like `spawn` but specifically creates a new Node.js process and sets up an IPC channel for sending messages between parent and child.

Child processes are useful for CPU-intensive tasks, external tools, or distributing work. Remember to handle `data` and `close` events, and consider using the `stdio` or `detached` options if needed. The `child_process` API is powerful for parallelizing tasks [13].

Debugging and Logging

Node apps can be debugged in several ways:

- `console.log` / `console.error`: Simple logging to the console.
- **Node Inspector:** Run Node with `--inspect` (or `node --inspect-brk`) and open Chrome DevTools or VS Code to step through code, set breakpoints, and inspect variables.

- **Debugger:** You can also use the built-in Node debugger by running `node debug app.js` (older approach).
- **Logging Libraries:** Use modules like `debug`, `winston`, or `pino` for structured, leveled logging. For example, the `debug` package lets you enable debug logs via `NODE_DEBUG` env var.

Useful tools:

- **nodemon:** Automatically restarts your app when files change (dev time only).
- **dotenv:** Loads environment variables from a `.env` file (as mentioned above).
- **pm2:** A production process manager that runs multiple Node instances, restarts on failure, and offers monitoring.

Combine `console` or logging libs with development tools (nodemon, inspector) to efficiently diagnose issues.

Deployment Best Practices

To deploy Node.js apps to production reliably, consider these best practices [14] [15]:

- **Use Environment Variables:** Do not hard-code configs. Utilize env vars (and `.env` files locally) so behavior can change per environment [15].
- **Clustering or Process Manager:** Node runs on a single core by default. Use the built-in `cluster` module or a tool like **PM2** to spawn processes equal to CPU cores, improving throughput [14]. For example, PM2 can manage process restarting and clustering for you.
- **Ignore dev artifacts:** Add `node_modules` and logs to `.gitignore`. Use `package.json` to list all dependencies, so any clone can `npm install` to get them [16].
- **Memory and GC Tuning:** For large apps or limited-memory environments, you can set V8 flags (like `--max-old-space-size`) in your start script [17].
- **Graceful Shutdown:** Listen for termination signals (`SIGINT`, `SIGTERM`) to close servers and database connections cleanly, allowing in-flight requests to finish.
- **Monitor and Log:** Use monitoring tools (AppMetrics, New Relic, etc.) and structured logging to track performance and errors in production.

Following these guidelines helps avoid downtime and ensures your Node.js service scales and recovers well.

Core Built-in Modules

Node.js includes many **core modules**. Below are brief notes on some essential ones:

- `http` / `https`: Build HTTP/HTTPS servers and make client requests. `https` is like `http` but with TLS. (See HTTP server above.)
- `url`: Utilities for URL resolution and parsing (`new URL()`, `url.parse()`).
- `path`: Handles and transforms file paths across OSes. It normalizes separators and resolves relative paths [18]. Example: `path.join(__dirname, 'folder', 'file.txt')`.
- `os`: Provides OS-related information (CPU, memory, platform, user info) [19]. For example, `os.platform()`, `os.cpus()`, `os.totalmem()`.

- `crypto`: Cryptography utilities. Offers hash functions, encryption (symmetric/asymmetric), HMAC, random bytes, etc. This wraps OpenSSL algorithms [20]. Example: `crypto.createHash('sha256').update(data).digest('hex')`. Essential for security-related tasks.
- `zlib`: Compression (gzip, deflate, Brotli). You can compress/decompress data streams to save bandwidth or disk space [21]. Example: `zlib.gzip(buffer, callback)`.
- `timers`: Timer functions (`setTimeout`, `setInterval`, `setImmediate`). These are global but also available via `require('timers')`. They schedule callbacks in the event loop [22].
- `dns`: Domain Name System lookups. Use `dns.lookup()`, `dns.resolveMx()`, etc., to convert hostnames to IPs and query DNS records [23].
- `util`: Miscellaneous utilities. Includes `util.format`, `util.inspect`, `util.promisify`, etc. It's a toolkit of helper functions for debugging and handling common tasks [24]. Example: `const readFile = util.promisify(fs.readFile)`.
- `assert`: Simple assertion testing for invariants, usually used in tests. Provides `assert.ok()`, `assert.strictEqual()`, `assert.deepEqual()`, etc., to check conditions and throw AssertionErrors [25]. Useful for quick tests or validating assumptions.

Understanding these core modules will help you accomplish almost any server-side task with Node.js.

Useful Tools

- **nodemon:** Watches your files and restarts the Node process when changes are detected (development-time convenience).
- **dotenv:** Loads environment variables from a `.env` file into `process.env` [12].
- **pm2:** A production process manager. It can start and monitor multiple Node processes, enabling zero-downtime reloads and automatic restarts.
- **nvm (Node Version Manager):** Handy for managing multiple Node versions on a development machine.
- **TypeScript / Babel:** While not Node-specific, many Node projects use TypeScript or Babel to use newer JS features.

These tools streamline development and deployment workflows in the Node.js ecosystem.

Interview Questions & Answers

Below are common Node.js interview questions with concise answers and examples.

1. What is Node.js and what are its key features?

Answer: Node.js is an open-source, cross-platform JavaScript runtime built on Chrome's V8 engine [1]. It lets you run JavaScript on the server (outside a browser). Key features include:
- **Event-driven, Single-threaded:** Uses an event loop to handle asynchronous tasks efficiently [2].
- **Non-blocking I/O:** File and network operations are asynchronous by default, preventing blocking the main thread.
- **Rich Ecosystem:** Comes with npm for managing packages (libraries) and has a large community.

- **Fast:** V8 engine compiles JS to native code, yielding high performance.
Overall, Node.js is suited for scalable network applications and real-time services.

2. Explain the Node.js event loop. Why is Node single-threaded?

Answer: Node's event loop allows handling many concurrent operations on a single thread. Even though JavaScript execution happens on one main thread, I/O tasks (like file or network access) are offloaded to the libuv thread pool. The event loop continuously checks for new events/callbacks and dispatches them for execution [2]. This model avoids the overhead of thread creation and locking. The phases of the event loop include: timers, pending callbacks, idle/poll (waiting for I/O), check (for `setImmediate`), and closing. Because Node uses non-blocking I/O, it efficiently manages multiple clients without multi-threading, simplifying code and reducing context-switching.

3. How do CommonJS modules work? What does `require()` do?

Answer: In Node.js, each file is a CommonJS module. `require('module')` is used to import modules. When you call `require()` with:
- a **core module** (like `http`), Node loads it internally,
- a **relative path** (like `./utils`), Node resolves the file (adding `.js` if needed),
- a **package name** (like `lodash`), Node looks in `node_modules` and loads the exported API.

The required module's code is wrapped in a function providing `exports`, `module`, `__filename`, and `__dirname`. The module runs once, exports whatever was assigned to `module.exports`, and its value is cached. Further `require()` calls return the cached object. This system encourages modular code.

Example:

```
// math.js
module.exports.add = (a, b) => a + b;

// app.js
const math = require('./math');
console.log(math.add(2, 3));  // 5
```

4. What is the difference between `exports` and `module.exports`?

Answer: Both `exports` and `module.exports` are used to export values from a module, but `exports` is actually a reference to `module.exports`. If you assign `module.exports = ...`, you replace the export object itself. If you only add properties to `exports` (e.g. `exports.foo = ...`), those get added to `module.exports`. However, reassigning `exports = {...}` will break the reference, and the new value won't be exported. Therefore, it's safer to use `module.exports` when exporting a function or object directly.

```
// Correct:
module.exports = { foo: 'bar' };
exports.baz = 42;

// Wrong:
exports = function() { return 'hello'; };  // This does NOT export this function
```

5. How do you manage packages and dependencies in Node.js?

Answer: Node uses **npm** (Node Package Manager) to manage packages. To start a project, you run `npm init` to create a `package.json`. You install packages with `npm install <package>`. By default, this puts the package in `node_modules` and adds it to `package.json` under `dependencies`. You can install dev-only packages (like test libraries) with `npm install --save-dev`. The `package.json` specifies all needed packages, so others can run `npm install` to get them. npm also lets you run scripts (e.g. `npm run start`) defined in `package.json`. Yarn is an alternative package manager, but npm is more common.

6. What are callbacks and how are they used in Node.js?

Answer: A callback is a function passed as an argument to another function, to be executed later. In Node.js, asynchronous APIs typically use callbacks. A common pattern is the error-first callback: `function(err, data)`. For example, `fs.readFile(path, 'utf8', (err, data) => { ... })`. When the read completes, this callback runs. Callbacks allow non-blocking code, but can lead to deeply nested code ("callback hell") if not managed carefully. This has led to using Promises and async/await as alternatives.

7. What are Promises? Give an example in Node.js.

Answer: A Promise is an object representing the future result of an asynchronous operation. It can be in a *pending*, *fulfilled*, or *rejected* state. You use `.then()` to handle success and `.catch()` for errors. For example:

```
const fs = require('fs').promises;
fs.readFile('file.txt', 'utf8')
  .then(data => {
    console.log('File contents:', data);
  })
  .catch(err => {
    console.error('Error reading file:', err);
  });
```

This avoids nested callbacks and makes error handling cleaner. You can also create promises manually:

```
function delay(ms) {
  return new Promise(resolve => setTimeout(resolve, ms));
}

delay(1000).then(() => console.log('1 second passed'));
```

8. How does async/await improve on Promises?

Answer: `async`/`await` is syntactic sugar for Promises, making asynchronous code look synchronous. An `async` function automatically returns a promise, and you can use `await` inside it to pause execution until a promise resolves. This flattens the code and avoids chaining `.then()`. For example:

```
async function main() {
  try {
    const fs = require('fs').promises;
    const data = await fs.readFile('file.txt', 'utf8');
    console.log(data);
  } catch (err) {
    console.error('Error:', err);
  }
}
main();
```

This is often more readable than equivalent promise chains.

9. What is an EventEmitter? How do you use it?

Answer: `EventEmitter` is a class in Node's `events` module that implements the publisher/subscriber pattern. Objects can emit named events, and other parts of the code can listen for and respond to these events. Usage involves creating (or extending) an EventEmitter, registering listeners with `.on('eventName', callback)`, and emitting events with `.emit('eventName', data)`. For example:

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

emitter.on('message', (text) => {
  console.log('Message received:', text);
});

emitter.emit('message', 'Hello World');  // logs: Message received: Hello World
```

Built-in streams and servers are EventEmitters (e.g. `request.on('data', ...)`).

10. Describe Buffers in Node.js.

Answer: A `Buffer` is Node's way of handling raw binary data. It's a fixed-size chunk of memory outside the V8 heap. Use `Buffer` when dealing with binary streams (like file data, network packets, etc.). Example creation:

```
const buf = Buffer.from('Hey');
console.log(buf[0]);            // 72 (ASCII code of 'H')
console.log(buf.toString());    // "Hey"
```

Buffers are useful for converting between strings and binary, and for working with streams. They can be sliced or concatenated, and you can specify encoding (utf8, hex, base64, etc.).

11. What are Streams, and why are they useful?

Answer: Streams are objects that allow reading or writing data piece by piece, rather than all at once [7]. They are memory-efficient for handling large data. There are four types:
- **Readable:** Source you can read from (e.g. `fs.createReadStream`).
- **Writable:** Sink you can write to (e.g. `fs.createWriteStream`).
- **Duplex:** Both read and write (e.g. a network socket).
- **Transform:** Like duplex but output is computed from input (e.g. zlib compression stream).

For example, you can stream a large file to a network client with minimal memory usage:

```
const fs = require('fs');
const server = require('http').createServer((req, res) => {
  const fileStream = fs.createReadStream('large-file.mp4');
  fileStream.pipe(res);  // Streams file to response
});
```

Streams also support `.pipe()` to chain data processing (e.g. compress before sending). They are core to Node's design for efficiency.

12. How do you read and write files in Node.js?

Answer: Using the `fs` (File System) module. For example, async read:

```
const fs = require('fs');
fs.readFile('data.json', 'utf8', (err, data) => {
  if (err) throw err;
  console.log('Data:', JSON.parse(data));
});
```

Async write:

```
const content = JSON.stringify({ name: 'Node' });
fs.writeFile('output.json', content, (err) => {
  if (err) console.error(err);
  else console.log('File saved.');
});
```

There are also promise-based (`fs.promises`) and sync methods (`fs.readFileSync`). Always prefer async methods for non-blocking I/O unless in a script. To watch files, use `fs.watch` or `chokidar`.

13. How do you create a simple HTTP server in Node.js?

Answer: Use the `http` module's `createServer()` method [9]. Example:

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

const server = http.createServer((req, res) => {
  console.log(`${req.method} ${req.url}`);
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello from Node!\n');
});

server.listen(3000, () => {
  console.log('Server listening on port 3000');
});
```

This sets up a server that logs each request and responds with "Hello from Node!". You can access `req.url` to handle different paths, and set headers/status via `res.writeHead()` and `res.end()`. For HTTPS, use `https.createServer()` with TLS key/cert.

14. What is middleware in Express (Node.js web framework)?

Answer: Middleware are functions that have access to the request (`req`), response (`res`), and a `next` function. They run during the processing of requests. In Express, middleware can execute code, modify `req`/`res`, end the response, or call `next()` to pass control. Examples include body parsers (`express.json()`), authentication checks, logging, etc. Middleware is chained in the order they are registered. For instance, `app.use((req, res, next) => { ...; next(); })`. According to the Express docs, middleware functions have access to `req`, `res`, and the `next` middleware, and can either complete the cycle or call `next()` [10].

15. How do you handle environment-specific configuration in Node.js?

Answer: Use **environment variables** and possibly a `.env` file for local development. Node exposes env vars via `process.env`. For example, set `NODE_ENV=production` in production and

`NODE_ENV=development` locally. In code:

```
const isProd = process.env.NODE_ENV === 'production';
```

For DB credentials or API keys, you can set `DB_HOST`, `API_KEY`, etc. in the environment or in an ignored `.env` file and load it with dotenv. This decouples configuration from code and avoids hardcoding sensitive info [11]. You can then write code that behaves differently based on env variables (ports, debugging, logging levels, etc.).

16. What are common ways to handle errors in Node.js?

Answer: For callback-style APIs, always check the first `err` argument and handle it (returning or throwing). For promise-based code, use `.catch()` or `try/catch` with async/await. Example with async/await:

```
try {
  const data = await asyncFunc();
  // ...
} catch (err) {
  console.error('Error occurred:', err);
}
```

For uncaught exceptions or rejections, you can listen on `process.on('uncaughtException')` or `process.on('unhandledRejection')` to log or cleanup before exit. In Express, errors are often passed to `next(err)` and handled by an error-handling middleware. Always validate inputs to avoid exceptions, and consider creating custom Error subclasses for clearer error types. Use synchronous code (`try/catch`) sparingly, since it blocks the thread; prefer asynchronous patterns.

17. What is the `child_process.spawn()` method? How is it different from `exec()`?

Answer: `child_process.spawn(command, args[])` launches a new process without spawning a shell, and returns streams (`stdout`, `stderr`) for I/O. It's suitable for large outputs or continuous data. In contrast, `exec()` runs a command in a shell and buffers the whole output, returning it in a callback. Because `spawn` streams data, it's more memory-efficient for big outputs. Example of `spawn`:

```
const { spawn } = require('child_process');
const proc = spawn('ls', ['-la']);
proc.stdout.on('data', (data) => {
  console.log(data.toString());
});
```

Example of `exec`:

```
const { exec } = require('child_process');
exec('ls -la', (err, stdout, stderr) => {
  if (err) throw err;
  console.log(stdout);
});
```

Use `spawn` for long-running processes or when you need streaming I/O. Use `exec` when you need to run a quick command and get the full result.

18. How does Node.js use multiple CPU cores?

Answer: By default, a Node.js process runs on a single core. To leverage multiple cores, you can use the **Cluster** module or external tools like PM2. The `cluster` module allows you to fork the process, creating worker processes (each with their own event loop) that can share server ports. For example, using `cluster` you can spawn `numCPUs` worker processes. Alternatively, PM2 can automatically run multiple instances of your app and manage them. This effectively load-balances requests across cores. Note that each worker is a separate process; you must handle shared resources (like sessions) appropriately (e.g., sticky sessions or external stores).

19. What is `process.nextTick()` vs `setImmediate()`?

Answer: Both schedule callbacks for future execution, but in different phases of the event loop.
- `process.nextTick(callback)` queues the callback to run **before** the next event loop tick, **immediately after** the current operation completes, even before I/O. It has higher priority.
- `setImmediate(callback)` queues the callback to run on the **check** phase, after I/O events of the current loop. It's roughly equivalent to `setTimeout(callback, 0)` but more efficient.

Example:

```
process.nextTick(() => console.log('nextTick'));
setImmediate(() => console.log('setImmediate'));
console.log('in main code');
```

This will log: `in main code`, then `nextTick`, then `setImmediate`. Use `nextTick` for operations that must happen before I/O, and `setImmediate` for deferring to the next cycle.

20. How can you serve JSON data from a Node.js HTTP server?

Answer: Set the `Content-Type` header to `application/json` and send a JSON string. For example:

```
const http = require('http');
http.createServer((req, res) => {
  if (req.url === '/data' && req.method === 'GET') {
    res.writeHead(200, {'Content-Type': 'application/json'});
```

```
    const payload = { message: 'Hello', time: new Date() };
    res.end(JSON.stringify(payload));
  } else {
    res.writeHead(404);
    res.end();
  }
}).listen(3000);
```

This server responds to GET `/data` with a JSON object. When using Express, you can simply call `res.json({ ... })` to set the header and stringfy for you.

21. What's the difference between `==` and `===` in Node.js?

Answer: Same as in JavaScript generally:
- `==` is loose equality and performs type coercion before comparing.
- `===` is strict equality and checks both value and type without coercion.

Best practice is to use `===` to avoid unexpected type conversions.

22. Explain the concept of "callback hell". How do you avoid it?

Answer: "Callback hell" refers to deeply nested callbacks when performing sequential asynchronous operations. For example:

```
doA((err, resA) => {
  doB(resA, (err, resB) => {
    doC(resB, (err, resC) => {
      // ...
    });
  });
});
```

This is hard to read and maintain. Ways to avoid it:
- Use **Promises** to chain operations instead of nesting.
- Use **async/await** for linear code style.
- Modularize code into named functions instead of anonymous callbacks.
- Use control-flow libraries (async.js, though modern code prefers Promises).
Essentially, flatten the structure by returning promises or using `async` functions.

23. How do you handle JSON request bodies in Node?

Answer: When using plain Node `http`, you must collect the request data and parse it. Example:

```
let body = '';
req.on('data', chunk => { body += chunk; });
```

```
req.on('end', () => {
  try {
    const data = JSON.parse(body);
    // process data
  } catch (e) {
    res.writeHead(400);
    res.end('Invalid JSON');
  }
});
```

In Express or similar frameworks, you can use a body-parsing middleware. For example, Express provides `express.json()` which automatically parses JSON:

```
const express = require('express');
const app = express();
app.use(express.json());  // parse JSON bodies

app.post('/api', (req, res) => {
  console.log(req.body); // already parsed JSON object
  res.send('OK');
});
```

This abstracts away the manual data accumulation.

24. What are environment variables and why are they important in Node apps?

Answer: Environment variables are key-value pairs set outside the app, influencing how the app runs. In Node, they're accessed via `process.env`. They're important for:
- **Configuration:** You can set e.g. `PORT`, `DB_HOST`, `API_KEY` differently in development, testing, and production.
- **Security:** Sensitive data (passwords, tokens) shouldn't be hardcoded; instead set via env vars.
- **Flexibility:** Changing behavior (like debug modes) without code changes.

Node automatically reads environment variables from the OS. Using packages like `dotenv` helps load variables from a file. The Heroku best practices guide emphasizes being *environmentally aware* – use one `.env` file locally and real env vars in production [15].

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

Answer: There are several methods:
- **Console Logging:** Quick and simple: use `console.log()` to inspect variables.
- **Debugger Protocol:** Run Node with `node --inspect yourapp.js`, then open `chrome://inspect` in Chrome or use VSCode's debugger to set breakpoints.
- **Built-in Debugger:** You can also `debugger;` in code and run `node inspect app.js` (older approach).

- `nodemon` **with Inspect:** You can combine nodemon and the inspector for automatic restarts.
- **Debug Packages:** Use `debug` module to enable granular logging (controlled via `DEBUG=pattern` env var).
- **IDE/Editor:** Many editors (VSCode, WebStorm) have Node debug integrations.

These tools let you step through code, watch expressions, and diagnose issues interactively.

26. What is the `__dirname` variable in Node.js?

Answer: In CommonJS modules, `__dirname` is a global variable (actually a local to the module) that contains the directory name of the current module file. For example, if your script is `/home/user/app/index.js`, then `__dirname` will be `/home/user/app`. It's often used with the `path` module to construct file paths reliably, for example:

```
const path = require('path');
const fullPath = path.join(__dirname, 'data', 'file.txt');
```

In ES modules (`.mjs` or `"type": "module"`), `__dirname` is not defined and you'd use `import.meta.url` with the `url` / `path` modules to get similar info.

27. How do you handle uncaught exceptions in Node.js?

Answer: Uncaught exceptions are errors not caught by any try/catch. You can listen for them:

```
process.on('uncaughtException', (err) => {
  console.error('Uncaught Exception:', err);
  // Ideally perform cleanup then exit:
  process.exit(1);
});
```

Similarly, for unhandled promise rejections:

```
process.on('unhandledRejection', (reason, promise) => {
  console.error('Unhandled Rejection:', reason);
  process.exit(1);
});
```

However, the Node.js docs recommend that in most cases you should let the process crash after logging the error (as the app might be in an inconsistent state) and restart it under a process manager. Use these handlers for logging and graceful shutdown (e.g., closing DB connections) before exit.

28. What are streams and backpressure in Node?

Answer: (Partially covered in Q11) Backpressure is a mechanism to handle situations when a writable stream cannot accept data as fast as a readable stream is providing it. Node streams can automatically pause and resume reading to match writable speed. For example, if you have `readStream.pipe(writeStream)`, and `writeStream` is slower, `readStream` will pause until `writeStream` drains. This ensures you don't buffer infinitely. In custom code, you use `stream.pause()` and `stream.resume()` or handle the `'drain'` event on writable streams to implement backpressure manually. Understanding backpressure is important for efficiently piping streams of data.

29. Why is Node.js single-threaded and how does it handle concurrent requests?

Answer: Node.js is designed with a single-threaded event loop to simplify programming and avoid the overhead of threads. It handles concurrency by offloading work (file I/O, network calls, crypto, etc.) to the system or a thread pool in the background (libuv). The event loop continues running and can process many connections. Each incoming request's I/O is non-blocking, so Node can interleave the handling of multiple requests without threads. For CPU-bound tasks, you can still use worker threads or child processes to avoid blocking the event loop.

30. How would you optimize a Node.js application for high performance?

Answer: Key strategies include:
- **Asynchronous I/O:** Always prefer non-blocking APIs and avoid CPU-blocking operations.
- **Clustering/Scaling:** Use multiple processes to utilize all CPU cores (e.g., cluster module, PM2).
- **Caching:** Cache frequent data (in-memory, Redis, etc.) to avoid repeated expensive operations.
- **Load Balancing:** If multiple servers, use a load balancer to distribute traffic.
- **Minimize Middleware:** Only use necessary middleware and keep middleware stack short (each adds latency).
- **Use Gzip Compression:** Enable `zlib` or built-in compression in Express for HTTP responses to reduce bandwidth.
- **Database Indexing:** Off-load heavy data processing to optimized DB queries (not Node's job).
- **Code Profiling:** Use tools (clinic.js, node --prof) to find bottlenecks.

The Heroku guide also suggests optimizing garbage collection flags if running in limited-memory environments [17]. Proper logging and monitoring (like APM) also help spot issues early.

31. What is Node.js "require cache"?

Answer: When you `require()` a module, Node caches the exported result. Subsequent `require()` calls for the same module (same path) return the cached object, not re-running the module code. This improves performance. You can inspect or clear the cache via `require.cache`, though it's rarely needed. The caching behavior means that modules are effectively singletons unless you delete them from cache.

32. How do you install a specific version of Node.js?

Answer: On Unix systems, a common tool is **nvm** (Node Version Manager). With nvm installed, you can run `nvm install 18.16.0` to install that version and `nvm use 18.16.0` to switch to it. On Windows,

there's nvm-windows or you can download official binaries. You can also use Docker containers or version managers in CI/CD pipelines. Ensuring the right Node version is crucial for compatibility.

33. How do you debug performance issues in Node?

Answer: Use profiling tools. Node has `--prof` to generate a V8 CPU profile. There are libraries like `clinic.js` (Clinic) for diagnosing CPU usage, event loop delays, and memory leaks. You can analyze flame graphs to see where the code spends time. Also use monitoring solutions (APM, logs) to watch metrics like response time and memory. Ensure you're not doing blocking operations and that you're using asynchronous patterns properly.

34. What is `npm start` in Node?

Answer: If a `package.json` has a `scripts` section with a `"start": "some command"`, then running `npm start` executes that command. By convention, `"start": "node index.js"` is common. If no start script is specified, `npm start` defaults to `node server.js`. You can also define `"dev": "nodemon index.js"` and run it via `npm run dev`. Scripts in `package.json` provide convenient shortcuts for common tasks.

35. Explain `module.exports = exports` behavior.

Answer: Initially, `exports` is a reference to `module.exports`. If you set `module.exports = something`, then `exports` no longer points to it. If you do `exports.key = value`, it adds to `module.exports`. However, if you assign `exports = { foo: 'bar' }`, this only changes the local `exports` variable, not `module.exports`. In Node, only `module.exports` is returned by `require`. So make sure to use `module.exports` when assigning a function or object directly.

36. What is `process.env.NODE_ENV` typically used for?

Answer: `NODE_ENV` is an environment variable commonly used to indicate the environment mode (development, production, test). Many libraries check `process.env.NODE_ENV === 'production'` to enable production optimizations (like caching, disabling debug). You might write code like:

```
if (process.env.NODE_ENV === 'production') {
  // use prod config, turn off verbose logging, etc.
} else {
  // development settings
}
```

When deploying, you set `NODE_ENV=production` to ensure production settings (e.g. faster but less safe error handling, minified assets, etc.). It's a best practice to manage environment-specific behavior this way.

37. What are some built-in ways to format and inspect objects (for logging)?

Answer: Node's `util` module offers helpers. For example, `util.format('Hello %s', name)` is like `printf`. `util.inspect(object)` returns a string representation of an object, useful for debugging. The `console` methods use `util.format` and `util.inspect` under the hood. Additionally, `console.dir(obj, { depth: null })` can show nested object structures. These utilities help log complex objects clearly.

38. How do you schedule code to run after a delay or at intervals in Node.js?

Answer: Use the **Timers** API:
- `setTimeout(fn, delay)` schedules `fn` once after `delay` milliseconds.
- `setInterval(fn, interval)` schedules repeated executions every `interval` ms.
- `setImmediate(fn)` schedules `fn` to run on the next event loop iteration (immediate).
These functions return a timer object or id which you can cancel with `clearTimeout(id)` or `clearInterval(id)` if needed. For example:

```
setTimeout(() => {
  console.log('This runs after 2 seconds');
}, 2000);

const id = setInterval(() => {
  console.log('Tick');
}, 1000);
clearInterval(id);  // stops the interval
```

From Node 15+, there's also a Promise-based timers API in `timers/promises`.

39. What is a REPL in Node.js?

Answer: REPL stands for *Read-Eval-Print Loop*. It's an interactive shell that comes with Node. You can start it by running `node` with no arguments. In the REPL, you type JavaScript code, it executes, and returns the result. It's useful for quickly testing code snippets, debugging, or exploring APIs. The REPL supports multiline mode, context (`.exit` to quit), and can `require` modules. For example:

```
$ node
> const fs = require('fs');
> fs.readFileSync
[Function: readFileSync]
> process.pid
12345
> .exit
```

40. How do you read environment variables from a `.env` file?

Answer: Use the **dotenv** package. First install it (`npm install dotenv`). Then at the top of your entry file:

```
require('dotenv').config();
```

This loads variables from a `.env` file in the same directory into `process.env`. For example, if `.env` contains `API_KEY=abc123`, after `config()` you can access `process.env.API_KEY`. This simplifies configuring local dev environments. Remember not to commit the `.env` file if it contains secrets; typically it's added to `.gitignore` [12].

41. What is PM2 and why is it used?

Answer: PM2 is a production process manager for Node.js. It helps you keep applications alive forever, reload on code changes, and manage performance. Features include:
- Running multiple instances (cluster mode).
- Automatic restarts on crash or file changes.
- Easy logs management (`pm2 logs`).
- Metrics and monitoring (`pm2 monit`).
- Simple commands to start, stop, restart apps.

Using PM2 makes deploying Node apps easier and more reliable by handling daemonization and restarts.

42. What is cluster mode in PM2 or Node?

Answer: Cluster mode refers to running multiple instances of the Node.js app (typically one per CPU core) to utilize multi-core systems. PM2's cluster mode or Node's `cluster` module spawns child processes that all listen on the same port. This increases concurrency. For example, with PM2:

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

This command starts as many instances as there are CPU cores. PM2 will load-balance requests among them. If one instance crashes, PM2 restarts it, ensuring high availability.

43. How do you connect to a database (e.g., MySQL) in Node.js?

Answer: Use a Node.js database client library. For example, for MySQL:

```
const mysql = require('mysql');
const conn = mysql.createConnection({
  host: process.env.DB_HOST,
  user: process.env.DB_USER,
  password: process.env.DB_PASS,
```

```
  database: 'mydb'
});

conn.connect(err => {
  if (err) throw err;
  conn.query('SELECT * FROM users', (err, rows) => {
    if (err) throw err;
    console.log(rows);
  });
});
```

Always handle connections asynchronously. For production, use connection pooling (e.g. `mysql.createPool`). For PostgreSQL, MongoDB, and others, similar client libraries exist (e.g. `pg`, `mongoose`).

44. What is error-first callback?

Answer: An error-first callback is a Node convention where the first parameter of a callback function is an error object. For example:

```
fs.readFile('file', (err, data) => {
  if (err) {
    // handle error
  } else {
    // use data
  }
});
```

If the operation succeeds, `err` is `null` or `undefined`. If it fails, `err` contains the error. This convention allows consistent error handling across Node APIs.

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

Answer: In Node, the global object is `global` (similar to `window` in browsers). Global variables and functions (like `setTimeout`) are actually properties of `global`. However, avoid polluting globals; instead use modules. Example:

```
global.db = { /* some global state */ };
console.log(global.db);
```

Also, Node provides `globalThis` as a standard alias to the global object.

46. How do you import and use built-in modules in Node.js?

Answer: Use `require` with the module name. For example:

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

No installation is needed for core modules. Then you can call their methods, e.g., `os.platform()` or `http.createServer()`. In newer Node versions with ES modules enabled, you can also use `import http from 'http';`.

47. How do you install Node.js on your system?

Answer: Node.js can be installed from the official site (nodejs.org) by downloading the installer for your OS. On Unix systems, using a version manager like **nvm** is recommended: `nvm install node` for latest. On macOS, you can also use Homebrew: `brew install node`. On Windows, use the official installer or tools like `nvm-windows`. On servers, you might use package managers (apt, yum) or Docker images. Always install npm and optionally yarn for package management.

48. How does Node.js handle `require()` of JSON files?

Answer: When you `require()` a `.json` file, Node parses it and returns the JSON object. Example:

```
const config = require('./config.json');
console.log(config.property);
```

Node caches the parsed JSON like other modules. Be cautious: `require()` caches the JSON, so changes to the file at runtime won't be picked up unless you clear the cache and re-require.

49. What are some security best practices for Node.js?

Answer: Important security measures include:
- **Never eval user input.** Avoid `eval()` or `Function`.
- **Validate inputs:** Use schemas or validation libraries to avoid injection attacks.
- **Use HTTPS:** Especially for production APIs (the `https` module or a proxy with SSL).
- **Keep dependencies updated:** Use tools like `npm audit` to check for vulnerabilities.
- **Avoid exposing internals:** Don't expose stack traces or error messages to clients (leak of info).
- **Set secure headers:** Use helmet middleware to set HTTP security headers.
- **Use non-root user:** Run your app under a dedicated user, not root.
- **Sanitize data:** Especially when using databases or file paths.
- **Use TLS for external API calls:** Encrypt sensitive data in transit.

Following the OWASP Node.js guidelines is recommended.

50. How do you handle static files in Node.js?

Answer: In plain Node, you can serve static files by reading them from disk and returning their contents with appropriate headers. For example:

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

const server = http.createServer((req, res) => {
  const filePath = path.join(__dirname, req.url);
  fs.readFile(filePath, (err, content) => {
    if (err) {
      res.writeHead(404); res.end('Not found');
    } else {
      res.writeHead(200, { 'Content-Type': 'text/html' });
      res.end(content);
    }
  });
});
```

However, in practice it's easier to use middleware: with Express, use `app.use(express.static('public'));` to serve files from a directory. This automatically handles mime types and caching headers for you.

51. What is the `path` module used for? Give an example.

Answer: The `path` module provides utilities to work with file and directory paths in a cross-platform way. It helps avoid issues with path separators (Windows vs Unix). For example, `path.join(__dirname, 'data', 'file.txt')` safely constructs a path. Other methods: `path.resolve`, `path.basename`, `path.dirname`. Example:

```
const path = require('path');
const fullPath = path.resolve(__dirname, 'subdir', 'index.html');
console.log(fullPath);
```

This ensures the correct separators and absolute paths.

52. What is `process.argv` in Node?

Answer: `process.argv` is an array containing command-line arguments passed when starting a Node script. `process.argv[0]` is the Node executable path, `[1]` is the script path, and subsequent indices are additional args. Example:

```
node app.js foo bar
```

In `app.js`:

```
console.log(process.argv);
// e.g. [ '/usr/local/bin/node', '/path/to/app.js', 'foo', 'bar' ]
```

This is useful for CLI programs to read options or parameters.

53. How do you pass options to `npm` scripts?

Answer: In `package.json`, under `"scripts"`, you can define commands. To pass arguments, use `npm run`. For example:

```
"scripts": {
  "start": "node app.js"
}
```

Run: `npm start -- --port=8080`. The `--` separates npm arguments from script arguments. Inside the script, you can access `process.argv` to get those. Also, `npm` adds `node_modules/.bin` to `PATH`, so you can call locally installed binaries directly in scripts.

54. How do you write unit tests in Node.js?

Answer: Use a testing framework like **Mocha**, **Jest**, **Jasmine**, or **AVA**. Example with Jest:

```
npm install --save-dev jest
```

In `package.json`:

```
"scripts": {
  "test": "jest"
}
```

Create a test file, e.g., `sum.test.js`:

```
const sum = require('./sum');
test('adds 2 + 3 to equal 5', () => {
  expect(sum(2, 3)).toBe(5);
});
```

Run `npm test`. In code, use assertions (e.g., Jest's `expect`, or Node's `assert` for simple checks [25]). Testing ensures code correctness. Many projects separate test and source files and use tools for coverage.

55. What is the difference between `npm install` and `npm ci`?

Answer:
- `npm install` installs dependencies and updates `package-lock.json` if needed. It's more flexible for development.
- `npm ci` (introduced in npm 5.7+) installs dependencies **exactly** from `package-lock.json` and fails if `package.json` and `package-lock.json` are out of sync. It's faster and ensures reproducible builds, so it's recommended for CI/CD pipelines or production deployments.

56. How can you debug a Node.js application with VS Code?

Answer: VS Code has built-in Node.js debugging. You can create a `launch.json` with a configuration:

```
{
  "type": "node",
  "request": "launch",
  "name": "Launch Program",
  "program": "${workspaceFolder}/app.js"
}
```

Then set breakpoints in your code and run the debugger. VS Code will launch the Node process and pause on breakpoints, letting you inspect variables, step through code, and view call stacks.

57. Explain callback vs promise vs async/await using an example of reading a file.

Answer: All are ways to handle async FS read:

- **Callback:**

```
const fs = require('fs');
fs.readFile('file.txt', 'utf8', (err, data) => {
  if (err) console.error(err);
  else console.log(data);
});
```

- **Promise:**

```
const fs = require('fs').promises;
fs.readFile('file.txt', 'utf8')
```

```
  .then(data => console.log(data))
  .catch(err => console.error(err));
```

- **Async/Await:**

```
const fs = require('fs').promises;
async function printFile() {
  try {
    const data = await fs.readFile('file.txt', 'utf8');
    console.log(data);
  } catch (err) {
    console.error(err);
  }
}
printFile();
```

All achieve the same goal, but async/await often yields the cleanest, linear code.

58. What is `process.on('SIGINT')` used for?

Answer: It listens for the SIGINT signal (sent when you press Ctrl+C). You can use it to gracefully shut down your app:

```
process.on('SIGINT', () => {
  console.log('Received SIGINT. Exiting gracefully.');
  // e.g., close server or cleanup here
  process.exit(0);
});
```

Without handling, a Node process will exit immediately on SIGINT. Handling it lets you perform cleanup (closing DB connections, finishing requests) before exit.

59. How do you compress HTTP responses in Node?

Answer: Use the `zlib` module (or middleware). For example, with plain Node you could:

```
const zlib = require('zlib');
// inside request handler:
const gzip = zlib.createGzip();
res.writeHead(200, {'Content-Encoding': 'gzip'});
someReadableStream.pipe(gzip).pipe(res);
```

This compresses the data stream. In Express, you can use the built-in compression middleware:

```
const compression = require('compression');
app.use(compression());
```

This automatically gzip-compresses responses where appropriate, saving bandwidth [21].

60. Why might you use `await` over `then` in promise chains?

Answer: `await` makes asynchronous code look and read like synchronous code, which improves readability and maintainability, especially in complex flows. It also simplifies error handling with `try/catch`. While using `.then()` is equivalent, deeply nested `.then()` can become hard to follow. `async` / `await` was introduced to make writing sequential async operations more straightforward.

61. What is the difference between `crypto.randomBytes()` and `crypto.randomInt()`?

Answer: Both are from the `crypto` module. `crypto.randomBytes(size, callback)` generates a cryptographically strong random buffer of given size in bytes. `crypto.randomInt(max, callback)` (Node 14+) returns a cryptographically secure random integer in [0, max). Use `randomBytes` when you need raw random bytes (e.g. for salts), and `randomInt` when you need a random number in a range. For example:

```
crypto.randomBytes(16, (err, buf) => { console.log(buf.toString('hex')); });
crypto.randomInt(100, (err, n) => { console.log(n); });
```

62. Can you explain what `Object.freeze()` does in JavaScript/Node?

Answer: `Object.freeze(obj)` makes an object immutable: you can no longer add, remove, or change its properties (in strict mode, attempts throw errors). It's useful for constants and ensuring a config object isn't modified. It's a JS feature (not Node-specific) but often used in Node apps to protect settings or defaults. Example:

```
const config = Object.freeze({
  host: 'localhost',
  port: 3000
});
config.port = 4000;  // fails silently or throws in strict mode
```

63. How do you deal with legacy callback-based APIs in async/await code?

Answer: Use `util.promisify()` to convert callback APIs into promise-returning functions. Example:

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

async function run() {
  const data = await readFile('file.txt', 'utf8');
  console.log(data);
}
run();
```

This is a common pattern when modernizing older code.

64. What is `setImmediate()` useful for?

Answer: `setImmediate(fn)` schedules `fn` to run after the current poll phase of the event loop (i.e., on the next iteration, but before timers). It's useful for breaking up long-running operations and yielding control back to the event loop so I/O can be processed. Unlike `setTimeout(fn, 0)`, `setImmediate()` is more efficient for next-tick scheduling. For example, if you have a heavy computation, you can chunk it and use `setImmediate()` between chunks to keep the app responsive.

65. Explain `require.cache`. When might you clear it?

Answer: Node caches modules on first load. `require.cache` is an object mapping filenames to module objects. If you change a module file and want to re-require it (for example, in a long-running REPL or during testing), you can delete the cached entry:

```
delete require.cache[require.resolve('./someModule')];
const freshModule = require('./someModule');
```

This forces Node to re-load the module. Generally, in production code, you don't manipulate this (to avoid caching issues). It's more of a developer tool.

66. What are some common Node.js environment variables?

Answer: Some important ones include:
- `NODE_ENV`: application environment (development, production, test).
- `PORT`: which port the server should listen on.
- `HOME` / `USER`: user's home directory (from OS).
- `PATH`: executable search paths (OS-level).
- `DEBUG`: used by the `debug` module to enable debug logs for certain namespaces.

Additionally, custom env vars for your config (like `DB_HOST`, `API_KEY`, etc.) are common. You set them in the shell or service config (e.g. in Heroku or Kubernetes configs).

67. How does Node's module system handle circular dependencies?

Answer: When two modules require each other, Node tries to resolve as much as it can. It provides each module with an incomplete version of the other's `exports` object during loading. This can work if used carefully (e.g., if you only need part of the other's exports). However, if you try to use the other module before it has finished initializing, you might get an empty object or missing functions. Avoid circular dependencies by refactoring or extracting shared code into separate modules. If unavoidable, ensure you only use the imported values after both modules have fully loaded.

68. What is a worker thread in Node?

Answer: Worker Threads (introduced in Node 10.5+ and stable in 12+) allow running JavaScript in parallel threads. This is different from `cluster` because workers share memory via `SharedArrayBuffer` and allow true multi-threading within a single process. Use case: CPU-intensive tasks that would block the event loop. Example usage:

```
const { Worker } = require('worker_threads');
const worker = new Worker('./worker.js');
worker.on('message', msg => console.log(msg));
worker.postMessage({ payload: 'do something' });
```

The `worker.js` file would contain code to handle messages and do CPU work. Workers have a heavier overhead than cluster processes, so use them for specific tasks.

69. What is `console.dir()` in Node?

Answer: `console.dir(obj, options)` prints an interactive listing of an object's properties, similar to `util.inspect`. It's useful to log objects with options like depth. For example, `console.dir(myObject, { depth: null })` shows the full recursive object. It's a handy alternative to `console.log` when inspecting object structures.

70. Can you explain how to use `process.memoryUsage()`?

Answer: `process.memoryUsage()` returns an object describing Node's memory usage in bytes. Example:

```
console.log(process.memoryUsage());
// { rss: 4935680, heapTotal: 1826816, heapUsed: 650472, external: 49879 }
```

- `rss` is Resident Set Size (total memory allocated for the process).
- `heapTotal` and `heapUsed` are V8's memory usage (JavaScript heap).
- `external` is memory usage of C++ objects bound to JS.

You can call this periodically to monitor for leaks or high usage. In production, you might log these or integrate with a monitoring tool.

Sources: The above answers incorporate information from Node.js official documentation and tutorials [1] [2] [26] [3] [4] [5] [6] [9] [10] [11] [13] [14] [15] [18] [19] [20] [21] [22] [23] [24] [25] (W3Schools and Node.js docs).

---

[1] Node.js Introduction
https://www.w3schools.com/nodejs/nodejs_intro.asp

[2] Node.js Architecture
https://www.w3schools.com/nodejs/nodejs_architecture.asp

[3] NodeJS NPM - GeeksforGeeks
https://www.geeksforgeeks.org/node-js/node-js-npm-node-package-manager/

[4] Callbacks vs Promises vs Async/Await - GeeksforGeeks
https://www.geeksforgeeks.org/javascript/callbacks-vs-promises-vs-async-await/

[5] Node.js Events
https://www.w3schools.com/nodejs/nodejs_events.asp

[6] Node.js Buffer Module
https://www.w3schools.com/nodejs/nodejs_buffer.asp

[7] Navigating Node.js Streams - Vercaa Hosting
https://vercaa.com/index.php?rp=%2Fknowledgebase%2F642%2FNavigating-Node.js-Streams.html&language=hebrew

[8] Node.js File System Module
https://www.w3schools.com/nodejs/nodejs_filesystem.asp

[9] Node.js HTTP Module
https://www.w3schools.com/nodejs/nodejs_http.asp

[10] Using Express middleware
https://expressjs.com/en/guide/using-middleware.html

[11] [12] Node.js Environment Variables
https://www.w3schools.com/nodejs/nodejs_environment.asp

[13] Child process | Node.js v24.9.0 Documentation
https://nodejs.org/api/child_process.html

[14] [15] [16] [17] Best Practices for Node.js Development | Heroku Dev Center
https://devcenter.heroku.com/articles/node-best-practices

[18] Node.js Path Module
https://www.w3schools.com/nodejs/nodejs_path.asp

[19] Node.js OS Module
https://www.w3schools.com/nodejs/nodejs_os.asp

[20] Node.js Crypto Module
https://www.w3schools.com/nodejs/nodejs_crypto.asp

[21] Node.js Zlib Module
https://www.w3schools.com/nodejs/nodejs_zlib.asp

22 Node.js Timers Module
https://www.w3schools.com/nodejs/nodejs_timers.asp

23 Node.js DNS Module
https://www.w3schools.com/nodejs/nodejs_dns.asp

24 Node.js Util Module
https://www.w3schools.com/nodejs/nodejs_util.asp

25 Node.js Assert Module
https://www.w3schools.com/nodejs/nodejs_assert.asp

26 Modules: CommonJS modules | Node.js v24.9.0 Documentation
https://nodejs.org/api/modules.html