

# Node.js


## Essential **Tips & Tricks** for Developers!



# 1. Async/Await for Cleaner Code


Simplify asynchronous code with `async/await` instead of nested callbacks or `.then()/.catch()` chains. It's more readable and reduces the callback hell.

// Instead of this:



```
1 someAsyncFunction()
2   .then(result => {
3     return anotherAsyncFunction(result);
4   })
5   .then(finalResult => {
6     console.log(finalResult);
7   })
8   .catch(err => console.error(err));
```

// Do this:



```
1 async function run() {
2   try {
3     const result = await someAsyncFunction();
4     const finalResult = await anotherAsyncFunction(result);
5     console.log(finalResult);
6   } catch (err) {
7     console.error(err);
8   }
9 }
10 run();
```

## 2. Destructuring Assignment

Destructure objects and arrays to access specific properties or elements in a cleaner way.



```
1  const user = { name: 'John', age: 30 };
2  const { name, age } = user; // Easier to access properties
3  console.log(name); // 'John'
4  console.log(age); // 30
```

## 3. Using path Module for File Paths

Avoid issues with file paths across different operating systems by using Node's path module.



```
1  const path = require('path');
2  const filePath = path.join(__dirname, 'folderName', 'file.txt');
3  console.log(filePath); // Properly resolves file paths
```

## 4. Debouncing API Calls with **Lodash**

Prevent too many requests in a short period by debouncing API calls. If you're handling user input, such as typing in a search bar or filtering a list, you don't want to make an API call every time the user types a character. This can result in excessive requests.



```
1  const _ = require('lodash');
2
3  const fetchData = () => console.log('API call made!');
4  const debouncedFetchData = _.debounce(fetchData, 500);
5
6  // Call this function on every input change
7  debouncedFetchData();
```

The **500** milliseconds is the amount of time the code will wait after the last call to `debouncedFetchData` before calling `fetchData`.

## 5. Using `async/await` with Promises in Parallel

You can run multiple `promises` concurrently using `Promise.all` inside an `async` function to improve performance.



```
1  async function fetchData() {
2      const [data1, data2] = await Promise.all([
3          fetch('https://api.example.com/data1'),
4          fetch('https://api.example.com/data2')
5      ]);
6      console.log(data1, data2);
7  }
8  fetchData();
9
```

## 6. Environment Variables with `.env` for Configs

Store configuration settings (like API keys or database URLs) in a `.env` file using the `dotenv` package to keep sensitive data secure and portable.

```
# .env file  
DB_HOST=localhost  
DB_USER=root  
DB_PASS=password
```




```
1 // Load environment variables  
2 require('dotenv').config();  
3 console.log(process.env.DB_HOST); // 'localhost'
```

## 7. Handle Uncaught Exceptions and Unhandled Rejections


Avoid app crashes by handling uncaught exceptions and unhandled promise rejections.

*This code listens for uncaught exceptions in your Node.js application. An uncaught exception occurs when an error is thrown and not caught by any try-catch block*



```
1 process.on('uncaughtException', (err) => {
2   console.error('Unhandled exception: ', err);
3   process.exit(1); // Exit the process
4 });
```

*This part of the code listens for unhandled promise rejections. A promise rejection occurs when a promise is rejected (usually because of an error), but there's no .catch() method or try-catch block to handle the rejection.*




```
1 process.on('unhandledRejection', (reason, promise) => {
2   console.error('Unhandled rejection: ', reason);
3 });
4
```

## 8. Avoid Blocking the Event Loop with `setImmediate`

Ensure that long-running tasks don't block the **event loop**, use `setImmediate` to break tasks into smaller- chunks.

The **event loop** is the mechanism that allows Node.js to handle multiple operations, like I/O (file reads, HTTP requests, etc.), without blocking the execution of other tasks. It's one of the key reasons Node.js is so efficient and scalable for handling asynchronous tasks.



```
1  setImmediate(() => {  
2    // Non-blocking code execution  
3    console.log('Executed asynchronously without blocking the event loop');  
4  });
```

`setImmediate()` schedules the provided callback function to run in the next iteration of the event loop.

The callback won't execute immediately, but it ensures that it will be executed as soon as the current phase of the event loop finishes (right after the I/O events). This makes it non-blocking because it won't stop other operations from executing while waiting for it to run.



## 9. Use **stream** for Handling Large Files

Instead of reading large files into memory all at once, use streams to handle large data efficiently.



```
1  const fs = require('fs');
2  const readableStream = fs.createReadStream('large-file.txt');
3
4  readableStream.on('data', (chunk) => {
5    console.log('Read chunk: ', chunk);
6  });
7
```

## 10. Use **os** Module to Get System Information

Quickly access system information such as available CPUs, memory, and platform.



```
1  const os = require('os');
2  console.log('CPU architecture: ', os.arch());
3  console.log('Free memory: ', os.freemem());
```

# 11. Use **cluster** for Load Balancing


Leverage Node.js's **cluster** module to utilize multiple CPU cores for better performance and load balancing in production.

```
1  const cluster = require('cluster');
2  const http = require('http');
3  const os = require('os');
4  const numCPUs = os.cpus().length;
5
6  if (cluster.isMaster) {
7    // Fork workers (one per CPU core)
8    for (let i = 0; i < numCPUs; i++) {
9      cluster.fork();
10   }
11
12   cluster.on('exit', (worker, code, signal) => {
13     console.log(`Worker ${worker.process.pid} died`);
14   });
15 } else {
16   // Worker processes have their own HTTP server
17   http.createServer((req, res) => {
18     // Log the worker that is handling the request
19     console.log(`Worker ${process.pid} is handling a request`);
20
21     // Define routes and logic for different requests
22     if (req.url === '/') {
23       res.writeHead(200, { 'Content-Type': 'text/plain' });
24       res.end(`Hello from Worker ${process.pid}! You are at the home page.`);
25     } else if (req.url === '/about') {
26       res.writeHead(200, { 'Content-Type': 'text/plain' });
27       res.end(`Hello from Worker ${process.pid}! This is the about page.`);
28     } else if (req.url === '/api/data') {
29       res.writeHead(200, { 'Content-Type': 'application/json' });
30       res.end(JSON.stringify({ message: 'This is some API data.', workerId: process.pid }));
31     } else {
32       // Handle unknown routes
33       res.writeHead(404, { 'Content-Type': 'text/plain' });
34       res.end(`Worker ${process.pid} could not find the requested page.`);
35     }
36   }).listen(8000, () => {
37     console.log(`Worker ${process.pid} is listening on port 8000`);
38   });
39 }
```

## 12. Caching API Responses with **node-cache**

Implement simple in-memory caching for your API responses to reduce load times and server requests.

**NodeCache** is a simple in-memory cache module for Node.js. It provides a way to store and retrieve key-value pairs in memory. It's often used to cache results from expensive operations (like database queries or external API requests) to improve performance by avoiding repeated calls for the same data.



```
1  const NodeCache = require('node-cache');
2  const cache = new NodeCache();
3
4  const getData = async (key) => {
5    try {
6      const cachedData = cache.get(key); // Check the cache first
7      if (cachedData) {
8        console.log("Returning cached data");
9        return cachedData; // Early return if cached data exists
10   }
11
12   console.log("Fetching new data from API");
13   const data = await fetchDataFromAPI(); // Fetch data if not in cache
14   cache.set(key, data, 60); // Cache the data for 60 seconds
15   return data; // Return the fresh data
16 } catch (error) {
17   console.error('Error fetching data:', error);
18   throw error; // Handle errors (e.g., failed API call)
19 }
20 };
21
```

## 13. Create a Custom Logger

Create a custom **logger** to capture different levels of logging (e.g., info, warn, error) in your application.

```
1  const fs = require('fs');
2  const path = require('path');
3  const chalk = require('chalk');
4
5  // Create a logs directory if it doesn't exist
6  const logDir = path.join(__dirname, 'logs');
7  if (!fs.existsSync(logDir)) {
8    fs.mkdirSync(logDir);
9  }
10
11 // Simple log levels
12 const logLevels = {
13   info: chalk.blue,
14   warn: chalk.yellow,
15   error: chalk.red,
16 };
17
18 // Logger function
19 const logger = (level, message) => {
20   const timestamp = new Date().toISOString();
21
22   // Validate log level
23   if (!logLevels[level]) {
24     console.log(chalk.gray(`[${timestamp}] [UNKNOWN LEVEL]: ${message}`));
25     return;
26   }
27
28   // Prepare the log message with color
29   const logMessage = `${timestamp} [${level.toUpperCase()}]: ${message}`;
30
31   // Output to console with colors
32   console.log(logLevels[level](logMessage));
33
34   // Write to a log file
35   const logFile = path.join(logDir, `${level}.log`);
36   fs.appendFileSync(logFile, logMessage + '\n');
37 };
38
39 // Example usage
40 logger('info', 'App started');
41 logger('warn', 'This is a warning');
42 logger('error', 'Something went wrong!');
```