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:

```
someAsyncFunction()
  .then(result => {
      return anotherAsyncFunction(result);
  })
  .then(finalResult => {
      console.log(finalResult);
  })
  .catch(err => console.error(err));
```

// Do this:

```
async function run() {
  try {
    const result = await someAsyncFunction();
    const finalResult = await anotherAsyncFunction(result);
    console.log(finalResult);
  } catch (err) {
    console.error(err);
  }
}
run();
```

2. Destructuring Assignment

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

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

```
const path = require('path');
const filePath = path.join(__dirname, 'folderName', 'file.txt');
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.

```
const _ = require('lodash');

const fetchData = () => console.log('API call made!');
const debouncedFetchData = _.debounce(fetchData, 500);

// Call this function on every input change
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.

```
async function fetchData() {
  const [data1, data2] = await Promise.all([
    fetch('https://api.example.com/data1'),
    fetch('https://api.example.com/data2')
  ]);
  console.log(data1, data2);
}
fetchData();
```

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
```

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

7. Handle Unchaught 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

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

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.

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

```

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.

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

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.

```
const fs = require('fs');
const readableStream = fs.createReadStream('large-file.txt');

readableStream.on('data', (chunk) => {
  console.log('Read chunk: ', chunk);
});

```

10. Use os Module to Get System Information

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

```
const os = require('os');
console.log('CPU architecture: ', os.arch());
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.

```
const cluster = require('cluster');
const http = require('http');
const os = require('os');
const numCPUs = os.cpus().length;

if (cluster.isMaster) {
  // Fork workers (one per CPU core)
  for (let i = 0; i < numCPUs; i++) {
    cluster.fork();
  }

  cluster.on('exit', (worker, code, signal) => {
    console.log(`Worker ${worker.process.pid} died`);
  });
} else {
  // Worker processes have their own HTTP server
  http.createServer((req, res) => {
    // Log the worker that is handling the request
    console.log(`Worker ${process.pid} is handling a request`);

    // Define routes and logic for different requests
    if (req.url === '/') {
      res.writeHead(200, { 'Content-Type': 'text/plain' });
      res.end(`Hello from Worker ${process.pid}! You are at the home page.`);
    } else if (req.url === '/about') {
      res.writeHead(200, { 'Content-Type': 'text/plain' });
      res.end(`Hello from Worker ${process.pid}! This is the about page.`);
    } else if (req.url === '/api/data') {
      res.writeHead(200, { 'Content-Type': 'application/json' });
      res.end(JSON.stringify({ message: 'This is some API data.', workerId: process.pid }));
    } else {
      // Handle unknown routes
      res.writeHead(404, { 'Content-Type': 'text/plain' });
      res.end(`Worker ${process.pid} could not find the requested page.`);
    }
  }).listen(8000, () => {
    console.log(`Worker ${process.pid} is listening on port 8000`);
  });
}
```

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.

```
const NodeCache = require('node-cache');
const cache = new NodeCache();

const getData = async (key) => {
  try {
    const cachedData = cache.get(key); // Check the cache first
    if (cachedData) {
      console.log("Returning cached data");
      return cachedData; // Early return if cached data exists
    }

    console.log("Fetching new data from API");
    const data = await fetchDataFromAPI(); // Fetch data if not in cache
    cache.set(key, data, 60); // Cache the data for 60 seconds
    return data; // Return the fresh data
  } catch (error) {
    console.error('Error fetching data:', error);
    throw error; // Handle errors (e.g., failed API call)
  }
};

```

13. Create a Custom Logger

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

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

// Create a logs directory if it doesn't exist
const logDir = path.join(__dirname, 'logs');
if (!fs.existsSync(logDir)) {
  fs.mkdirSync(logDir);
}

// Simple log levels
const logLevels = {
  info: chalk.blue,
  warn: chalk.yellow,
  error: chalk.red,
};

// Logger function
const logger = (level, message) => {
  const timestamp = new Date().toISOString();

  // Validate log level
  if (!logLevels[level]) {
    console.log(chalk.gray(`[${timestamp}] [UNKNOWN LEVEL]: ${message}`));
    return;
  }

  // Prepare the log message with color
  const logMessage = `${timestamp} [${level.toUpperCase()}]: ${message}`;

  // Output to console with colors
  console.log(logLevels[level](logMessage));

  // Write to a log file
  const logFile = path.join(logDir, `${level}.log`);
  fs.appendFileSync(logFile, logMessage + '\n');
};

// Example usage
logger('info', 'App started');
logger('warn', 'This is a warning');
logger('error', 'Something went wrong!');
```