Guía de Estilos para Node.js

Esta guía establece convenciones y mejores prácticas para desarrollar aplicaciones con Node.js, con el objetivo de mejorar la legibilidad, mantenibilidad y consistencia del código en proyectos de equipo. Está inspirada en estándares de la comunidad y adaptada para reflejar prácticas modernas hasta abril de 2025.

Formato del Código

- **Indentación**: Usar 2 espacios para la indentación. No usar tabulaciones.
- **Nuevas líneas**: Usar nuevas líneas estilo UNIX (`\n`), asegurando que el último carácter del archivo sea una nueva línea.
- **Espacios en blanco**: Eliminar espacios en blanco al final de las líneas.
- **Punto y coma**: Usar punto y coma al final de las sentencias.
- **Longitud de línea**: Limitar las líneas a 80 caracteres para facilitar la lectura en pantallas divididas.
- **Comillas**: Usar comillas simples, excepto en JSON donde se usan comillas dobles.
- **Llaves**: Colocar la llave de apertura en la misma línea que la declaración.
- **Declaraciones de variables**: Usar una variable por sentencia `var` , `let` o `const` . Preferir `const` para valores inmutables y `let` para variables reasignables, evitando `var` .

Ejemplo:

```
const userName = 'Juan';
let userAge = 25;

if (userAge >= 18) {
  console.log(`${userName} es mayor de edad`);
}
```

Convenciones de Nombres

- **Variables, propiedades y funciones**: Usar `lowerCamelCase` (por ejemplo, `calculateTotal` , `userProfile`).
- **Clases**: Usar `UpperCamelCase` (por ejemplo, `UserManager`).
- **Constantes**: Usar `UPPERCASE` con guiones bajos para separar palabras (por ejemplo, `MAX_USERS`).

Ejemplo:

```
const MAX_USERS = 100;

class UserManager {
  calculateTotalUsers() {
    return MAX_USERS;
  }
}
```

Variables y Estructuras de Datos

- **Creación de objetos y arrays**: Incluir comas finales, mantener declaraciones cortas en una sola línea, y citar claves solo cuando sea necesario (por ejemplo, si contienen caracteres especiales).

Ejemplo:

```
const arr = [1, 2, 3,];

const obj = {
  key: 'value',
  anotherKey: 42,
};
```

Condicionales

- **Operador de igualdad**: Usar `===` para comparaciones estrictas, evitando `==` .
- **Operador ternario**: Usar en múltiples líneas para mayor claridad.
- **Condiciones descriptivas**: Asignar condiciones complejas a variables con nombres descriptivos.

Ejemplo:

```
const isValidUser = user.age >= 18 && user.isActive;

const result = isValidUser
  ? 'Usuario válido'
  : 'Usuario no válido';
```

Funciones

- **Funciones pequeñas**: Limitar las funciones a aproximadamente 15 líneas para facilitar su comprensión.
- **Retorno temprano**: Usar retornos tempranos para reducir la complejidad del flujo.
- **Nombrar closures**: Asignar nombres a funciones anónimas para mejorar la depuración.
- **Evitar closures anidados**: Mantener estructuras planas para mayor claridad.
- **Encadenamiento de métodos**: Colocar un método por línea con indentación adecuada.

Ejemplo:

```
function processData(data) {
  if (!data) {
    return null;
  }
  return data.toUpperCase();
}

setTimeout(function timeoutHandler() {
  console.log('Ejecutado');
}, 1000);

Promise.resolve()
  .then(processData)
  .then(logResult)
  .catch(handleError);
```

Comentarios

- **Uso de comentarios**: Usar comentarios para explicar mecanismos de alto nivel o clarificar código complejo. Evitar comentarios redundantes que repitan lo obvio.
- **Formato**: Usar `//` para comentarios de una línea y `/* */` para comentarios de varias líneas.

Ejemplo:

```
// Calcula el total de usuarios activos
function getActiveUsers(users) {
  return users.filter(user => user.isActive);
}
```

Manejo de Errores

- **Verificar errores en callbacks**: Siempre comprobar errores en callbacks.
- **Patrón de error**: Usar `if (err) return callback(err);` para manejar errores.
- **Lanzar errores**: Solo lanzar errores en funciones síncronas; en asíncronas, pasar errores al callback.
- **Capturar errores**: Usar try-catch para operaciones síncronas dentro de código asíncrono.

Ejemplo:

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

fs.readFile('file.txt', (err, data) => {
  if (err) {
    return callback(err);
  }
  console.log(data);
});

async function readJson(file) {
  try {
    const data = await fs.promises.readFile(file, 'utf8');
    return JSON.parse(data);
  } catch (err) {
    throw err;
  }
}
```

Programación Asíncrona

- **APIs consistentes**: Las funciones deben ser siempre síncronas o siempre asíncronas.
- **Uso de** `process.nextTick` : Usar para resultados cacheados, asegurando consistencia en el flujo asíncrono.
- **Convención de callbacks**: Seguir el estándar de Node.js: el callback es el último argumento, y el primer argumento del callback es el error.

Ejemplo:

```
function getCachedData(key, callback) {
  if (cache[key]) {
    process.nextTick(() => callback(null, cache[key]));
    return;
  }
  fetchData(key, callback);
}
```

Módulos y Dependencias

- **Ubicación de** `require` : Colocar todas las sentencias `require` al inicio del archivo.
- **Evitar extensiones de prototipos**: No modificar prototipos de objetos nativos.
- **ES Modules vs. CommonJS**: Para nuevos proyectos, preferir ES Modules (`import` / `export`), configurando `"type": "module"` en `package.json` . CommonJS (`require` / `module.exports`) sigue siendo válido para proyectos existentes o bibliotecas específicas.

Ejemplo de ES Modules:

```
// modulo.js
export function suma(a, b) {
  return a + b;
}

// main.js
import { suma } from './modulo.js';
console.log(suma(2, 3));
```

Ejemplo de CommonJS:

```
// modulo.js
function suma(a, b) {
  return a + b;
}
module.exports = { suma };

// main.js
const { suma } = require('./modulo');
console.log(suma(2, 3));
```

Aspecto	ES Modules	CommonJS
Sintaxis	`import` / `export`	`require` / `module.exports`
Carga	Asíncrona	Síncrona
Soporte	Estándar ECMAScript, recomendado	Legado, ampliamente usado
Configuración	`"type": "module"` en `package.json`	Predeterminado en Node.js

Herramientas y Linters

- **ESLint**: Usar ESLint para garantizar consistencia y detectar errores. Configurar para entornos Node.js.
- **EditorConfig**: Implementar EditorConfig para uniformizar configuraciones de editores.

Ejemplo de `.eslintrc.json`:

```
{
  "env": {
    "node": true,
    "es2021": true
  },
  "extends": "eslint:recommended",
  "parserOptions": {
    "ecmaVersion": 12
  },
  "rules": {
    "indent": ["error", 2],
    "quotes": ["error", "single"],
    "semi": ["error", "always"]
  }
}
```

Ejemplo de `.editorconfig`:

```
root = true

[*]
indent_style = space
indent_size = 2
end_of_line = lf
charset = utf-8
trim_trailing_whitespace = true
insert_final_newline = true
```

Pruebas

- **Importancia**: Escribir pruebas unitarias y de integración para garantizar la calidad del código.
- **Frameworks**: Usar herramientas como Mocha, Jest, o similares.

Rendimiento y Seguridad

- **Mejores prácticas**: Consultar recursos especializados para optimizar el rendimiento y asegurar la seguridad, como Node.js Best Practices.
- **Consideraciones**: Evitar operaciones bloqueantes y validar entradas para prevenir vulnerabilidades.

Referencias

Esta guía se basa en estándares de la comunidad y recursos actualizados hasta abril de 2025, incluyendo:

- Node.js Style Guide by Felix Geisendörfer
- Xcidic Coding Guideline for NodeJS
- JavaScript Standard Style
- CommonJS vs. ES Modules in Node.js - LogRocket Blog
- MDN Web Docs: JavaScript