Documentación Técnica: Estándares de Codificación

1. Introducción

Un estándar de codificación es un conjunto de convenciones, reglas y buenas prácticas que deben seguir todos los miembros de un equipo de desarrollo para garantizar que el código fuente sea legible, mantenible, coherente y escalable. Esta documentación describe de forma detallada los aspectos a considerar al definir dichos estándares, aplicables a proyectos con stack Node.js y Angular, utilizando Prettier como herramienta de formateo.

2. Objetivos de los Estándares

- Mejorar la legibilidad del código.
- Asegurar coherencia entre múltiples desarrolladores.
- Facilitar el mantenimiento y la extensión del sistema.
- Prevenir errores comunes de programación.
- Facilitar las revisiones de código.
- Agilizar el onboarding de nuevos miembros al equipo.

3. Herramientas de Apoyo

- **Prettier**: Formateo automático de código.
- **ESLint**: Reglas adicionales de buenas prácticas y errores comunes.
- **Husky + lint-staged**: Verificación de formato antes de hacer commit.
- **TypeScript**: Tipado estático en el frontend y backend.

4. Estilo General del Código

4.1. Reglas de Prettier (configuración por defecto)

- Sangría: 2 espacios.
- Punto y coma al final de las líneas: obligatorio.
- Comillas: simples (').
- Longitud máxima de línea: 80 caracteres.
- Espacios dentro de llaves: permitidos ({ nombre: 'Sensor' }).
- Paréntesis en funciones flecha: siempre ((param) => {}).

Ejemplo:

```
const mensaje = 'Hola mundo';
```

```
const sensor = {
 nombre: 'Sensor A',
 activo: true,
};
const saludar = (nombre) => {
 console.log(` Hola, ${nombre}` );
};
```

4.2. Otros Estilos Recomendados

- Una instrucción por línea.
- Uso de llaves {} incluso para bloques de una sola línea.
- Separación lógica con líneas en blanco entre secciones.

4.3. Estándares Específicos para JavaScript

- Utilizar strict mode cuando sea necesario para mayor seguridad:

```
'use strict';
```

- Declarar variables con const o let según el uso.
- Evitar el uso de var.
- Usar === en lugar de == para comparaciones estrictas.
- No extender objetos nativos como Object.prototype.
- Utilizar funciones flecha para callbacks simples:

```
items.forEach(item => console.log(item));
```

- Manejo adecuado de promesas con async/await.

```
const obtenerDatos = async () => {
 try {
  const respuesta = await fetch('/api/data');
  const datos = await respuesta.json();
  console.log(datos);
 } catch (error) {
  console.error('Error:', error);
```

```
 }
};
```

5. Nomenclatura

Elemento	Convención	Ejemplo
Variables	camelCase	sensorActivo
Funciones	camelCase	obtenerLecturas()
Clases	PascalCase	ControladorSensor
Componentes	PascalCase	LoginComponent
Interfaces	Prefijo I	ISensor
Constantes	UPPER_SNAKE_CASE	API_BASE_URL

Ejemplo:

```
interface ISensor {
 id: number;
 nombre: string;
 activo: boolean;
}


const API_BASE_URL = 'https://api.miapp.com';
const obtenerLecturas = () => {
 // lógica de lectura
};
```

6. Organización del Código

6.1. Orden de Importaciones

1. Módulos del sistema o terceros.
2. Utilidades o helpers.
3. Servicios.
4. Componentes.

5. Archivos relativos locales.

6.2. Estructura del Archivo

- Comentario de encabezado (si aplica).
- Importaciones.
- Constantes y configuraciones.
- Definición de funciones, clases, exportaciones.

Ejemplo:

```
// src/app/sensores/sensor.service.ts


import { HttpClient } from '@angular/common/http';
import { Injectable } from '@angular/core';
import { ISensor } from './sensor.model';


@Injectable({ providedIn: 'root' })
export class SensorService {
 constructor(private http: HttpClient) {}


 obtenerSensores() {
  return this.http.get<ISensor[]>('/api/sensores');
 }
}
```

6.3. Separación de Responsabilidades

- No mezclar lógica de negocio con manipulación de interfaz.
- Dividir funcionalidades grandes en módulos reutilizables.

7. Buenas Prácticas

7.1. Variables y Tipado

- Usar const por defecto. Solo usar let si el valor cambia.
- Evitar var.

- Tipar todas las funciones y variables en TypeScript.

7.2. Funciones

- Funciones puras y con un solo propósito.
- No más de 40 líneas por función (recomendado).
- Usar funciones flecha para callbacks.

7.3. Comentarios

- Solo cuando el código no sea autoexplicativo.
- Usar comentarios en bloque /** */ para documentación.

Ejemplo:

```
/**
 * Calcula el promedio de una lista de números.
 */
const calcularPromedio = (valores: number[]): number => {
  const total = valores.reduce((acc, val) => acc + val, 0);
  return total / valores.length;
};
```

7.4. Control de Errores

- Manejar todos los errores posibles.
- Evitar silencios en bloques try/catch.

7.5. Seguridad

- Validar datos de entrada.
- No dejar contraseñas o tokens en el código fuente.
- Usar dotenv para variables sensibles.

8. Versionado y Commits

- Seguir convenciones de commits (ej. Conventional Commits).
- Ejemplo: feat: agregar nuevo componente de sensores
- Commits cortos y descriptivos.

9. Formateo y Validación Automatizada

9.1. Prettier

- Ejecutar con: npx prettier --write .
- Integrar en el editor (autoformato al guardar).

9.2. Git Hooks

- Usar husky y lint-staged para aplicar Prettier antes de cada commit.

10. Revisiones de Código

- Todo cambio debe pasar por un Pull Request.
- Validar estilo, buenas prácticas, cobertura y funcionamiento.
- Solicitar al menos una aprobación antes del merge.