

Informe Código Limpio

Presentado por Grupo 6:

Ana María González Hernández - anagonzalezhe@unal.edu.co

Daniel Felipe Soracipa - dsoracipa@unal.edu.co

Juan José Medina Guerrero - jmedinagu@unal.edu.co

Samuel Josué Vargas Castro - samvargasca@unal.edu.co

Profesor:

Oscar Eduardo Alvarez Rodriguez

oalvarezr@unal.edu.co

Febrero 9 de 2025



Universidad Nacional de Colombia
Facultad de Ingeniería
Departamento de Ingeniería de Sistemas e Industrial
2025

Clean Code

En el desarrollo de software, mantener un código limpio y bien estructurado es fundamental para garantizar la mantenibilidad, escalabilidad y colaboración eficiente dentro de un equipo. En este informe se detallará la implementación de linters en un proyecto que consta de dos componentes principales: backend y frontend. Se explicará el proceso paso a paso para configurar y aplicar herramientas de análisis estático en cada parte del sistema, asegurando el cumplimiento de buenas prácticas de codificación, la detección temprana de errores y la mejora en la calidad del código.

Backend

Para el desarrollo del Backend se usarán los lint: ESLint y Prettier que son los más comunes y usados para frameworks que usan JavaScript y TypeScript, como para este caso en el que usaremos Nest.js es un framework de desarrollo web basado en Node.js.

La implementación de estas herramientas de análisis de código estático, es muy sencilla ya que al inicializar un nuevo proyecto de Nest.js, los archivos de ESLint y Prettier son agregados al proyecto y además de esto viene configurados.

A continuación se podrá observar como después de crear un nuevo proyecto ya podemos hacer uso de Prettier y ESLint.

Antes de crear un nuevo proyecto en Nest.js debemos tener algún gestor de paquetes como NPM. Además, se debe previamente haber instalado Nest.js (mediante la línea : `npm install -g @nestjs/cli`).

Luego de esto se ubica en la carpeta del proyecto y se usa la línea : `nest new "nombre del proyecto"` y se escoge el gestor de paquetes npm.

```
C:\Users\DANIEL\Documents\UNI\sem 6\ingesoft\proyecto\Proyecto_IngeSoft\Proyecto\Backend>nest new chazzap
✦ We will scaffold your app in a few seconds..

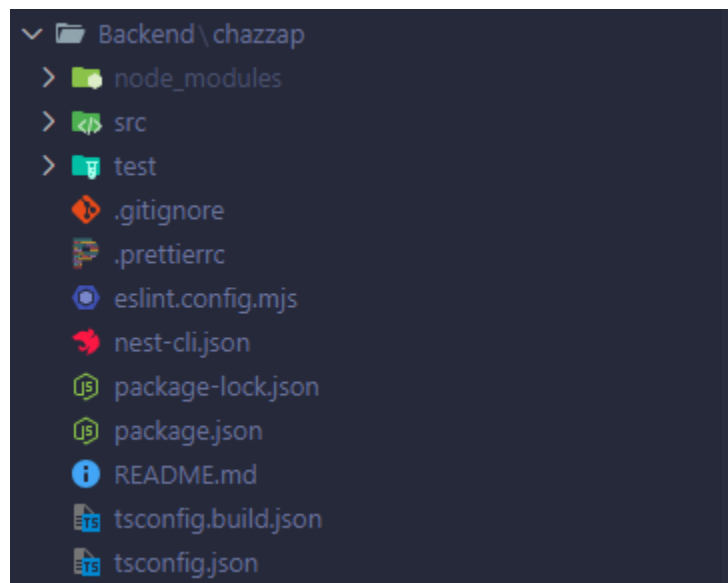
Which package manager would you ❤️ to use? npm
CREATE chazzap/.prettierrc (54 bytes)
CREATE chazzap/eslint.config.mjs (890 bytes)
CREATE chazzap/nest-cli.json (179 bytes)
CREATE chazzap/package.json (2107 bytes)
CREATE chazzap/README.md (5304 bytes)
CREATE chazzap/tsconfig.build.json (101 bytes)
CREATE chazzap/tsconfig.json (565 bytes)
CREATE chazzap/src/app.controller.ts (286 bytes)
CREATE chazzap/src/app.module.ts (259 bytes)
CREATE chazzap/src/app.service.ts (150 bytes)
CREATE chazzap/src/main.ts (236 bytes)
CREATE chazzap/src/app.controller.spec.ts (639 bytes)
CREATE chazzap/test/jest-e2e.json (192 bytes)
CREATE chazzap/test/app.e2e-spec.ts (699 bytes)
```

Esto crea un nuevo proyecto, en el que podemos ver ya se encuentran Prettier y ESLint listos para usar.

```
🚀 Successfully created project chazzap
👉 Get started with the following commands:

$ cd chazzap
$ npm run start

Thanks for installing Nest 🙏
Please consider donating to our open collective
to help us maintain this package.
```



Listo ahora podemos ejecutar ESLint mediante “npm run lint”, para que compruebe en todo el código si hay reglas que se estén rompiendo:

```
\chazzap>npm run lint

> chazzap@0.0.1 lint
> eslint "{src,apps,libs,test}/**/*.ts" --fix

C:\Users\DANIEL\Documents\UNI\sem 6\ingesoft\proyecto\Proyecto_IngeSoft\Proyecto\Backend\chazzap\src\main.ts
  8:1  warning  Promises must be awaited, end with a call to .catch, end with a call to .then with a rejection handler or be explicitly marked as ignored with the 'void' operator @typescript-eslint/no-floating-promises

✖ 1 problem (0 errors, 1 warning)
```

Al correr el lint se puede observar que no hay ninguna regla que se esté rompiendo, esta es una forma de comprobar las reglas en todo el código de forma automática, para mantener la consistencia y buenas prácticas, además de identificar y solucionar errores comunes o de sintaxis.

Y también mediante “npm run format” podemos ejecutar la herramienta de formateo automático Prettier.

```
npm run format

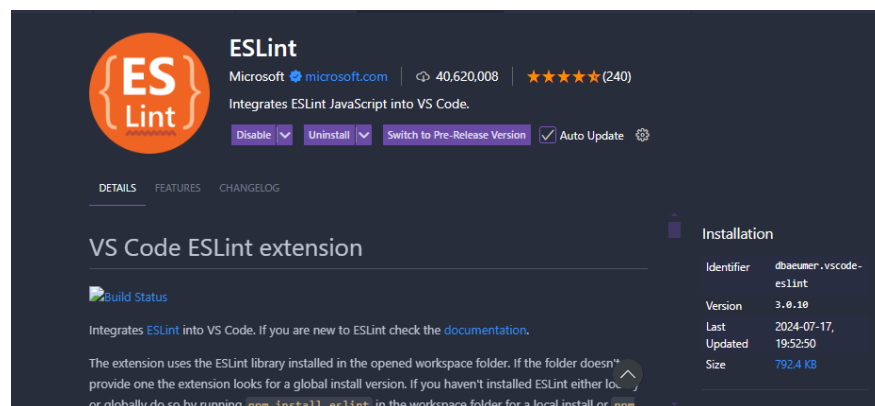
> chazzap@0.0.1 format
> prettier --write "src/**/*.ts" "test/**/*.ts"

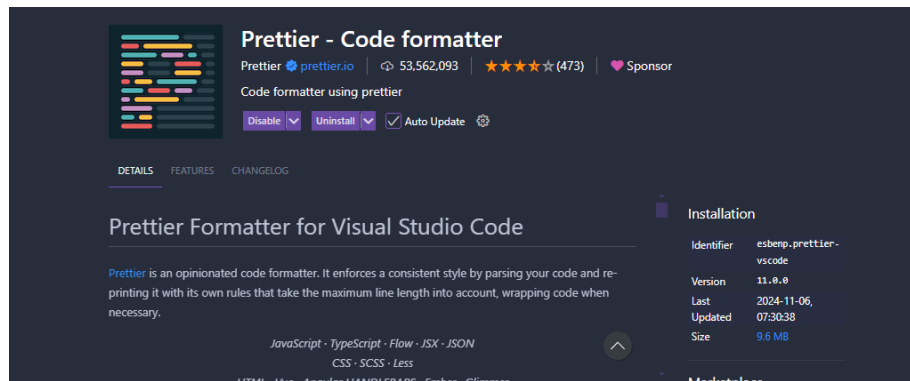
src/app.controller.spec.ts 83ms
src/app.controller.ts
[error] src/app.controller.ts: SyntaxError: Use of reserved word in strict mode (6:15)
[error]    4 | @Controller()
[error]    5 | export class AppController {
[error] > 6 |   constructor(private readonly appService: AppService) {}
[error]      |               ^^^^^^^
[error]    7 |
[error]    8 |   @Get()
[error]    9 |   getHello(): string {
src/app.module.ts 3ms
src/app.service.ts 2ms
src/main.ts 5ms
test/app.e2e-spec.ts 9ms

C:\Users\DANIEL\Documents\UNI\sem 6\ingesoft\proyecto\Proyecto_IngeSoft\Proyecto\Backend\chazzap
```

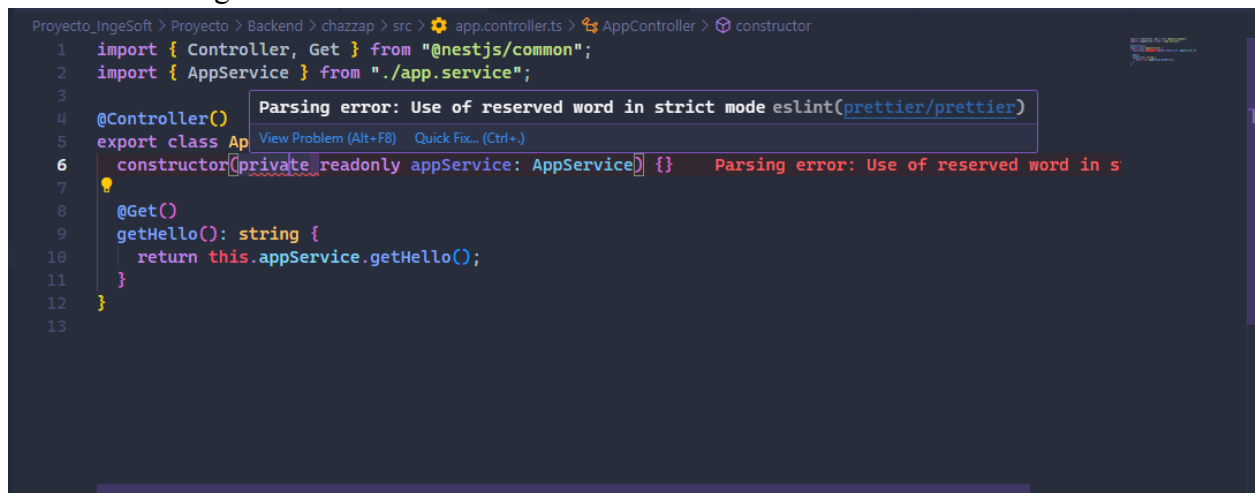
Se puede observar como Prettier encuentra un error de sintaxis en su archivo src/app.controller.ts. Específicamente, parece estar relacionado con la palabra clave privada en el constructor de su clase AppController. De esta forma se puede ir a la ubicación del error y solucionar este error de sintaxis.

Además podemos hacer uso de extensiones de Vscode que nos permitirán ver los errores directamente en el código de estas dos herramientas.





Facilitando y haciendo más intuitiva la detección y solución de errores mientras estamos escribiendo código.



Frontend

En la sección de frontend, se implementará 'flutter_lints' como herramienta principal para el análisis estático del código en Flutter, diferenciándolo de los lints propios de Dart, los cuales están más orientados a proyectos que no dependen del framework. Mientras que 'dart analyze' ofrece reglas generales para la calidad del código en Dart, 'flutter_lints' extiende estas reglas con configuraciones específicas para proyectos en Flutter, asegurando que se sigan las mejores prácticas recomendadas para el desarrollo de aplicaciones móviles y web con este framework.

En esta parte del informe, se detallará el proceso de configuración y aplicación de 'flutter_lints', incluyendo un paso a paso con capturas de pantalla que ilustren la automatización dentro del proyecto.

Los lints de flutter se definen en un paquete publicado en la [biblioteca de dart](#) y definen un conjunto de reglas que siguen el formato recomendado para la escritura de código en dart. Son utilizados por el comando 'flutter analyze' pero también se incluyen en extensiones de IDE como Visual Studio Code.

Se creó un código de prueba que incumple algunas reglas del linter como se muestra a continuación:

```
1  import 'package:flutter/material.dart';
2
3  Run | Debug | Profile
4  void main() {
5    runApp(MyApp());
6  }
7
8  class MyApp extends StatelessWidget { Constructors for public widgets should have a named 'key' parameter.⚠Try addi
9    // Incumple la regla de nombres de constantes en mayúsculas
10   final String APP_NAME = "Linter Test"; The variable name 'APP_NAME' isn't a lowerCamelCase identifier.⚠Try changi
11
12   @override
13   Widget build(BuildContext context) {
14     print("App started"); // 'print' debería reemplazarse por 'debugPrint' Don't invoke 'print' in production code.
15
16     return MaterialApp(
17       home: Scaffold(
18         appBar: AppBar(title: Text(APP_NAME)),
19         body: Center( Use 'const' with the constructor to improve performance.⚠Try adding the 'const' keyword to th
20           child: Text('Hello, Linter!'), Use 'const' with the constructor to improve performance.⚠Try adding the 'c
21         ), // Center
22       ), // Scaffold
23     ); // MaterialApp
24   }
25 }
```

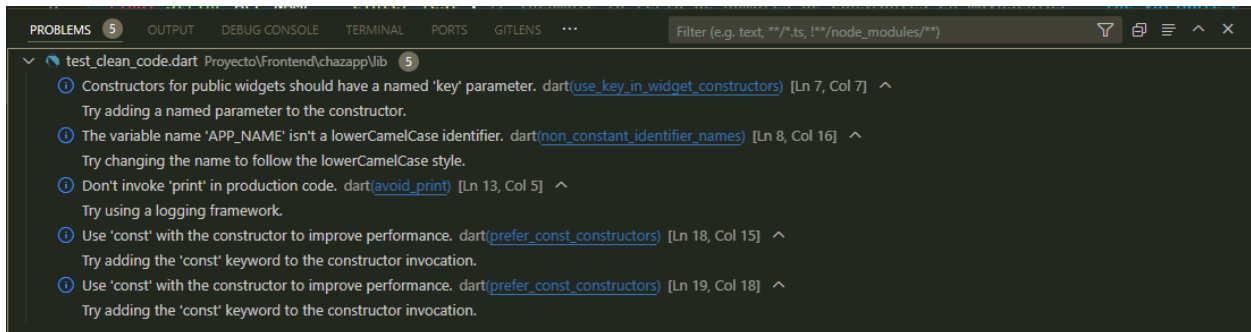
Como se puede observar, el IDE resalta los incumplimientos de las reglas utilizando el linter. Además de ello, se utiliza el comando 'flutter analyze' para ver resaltados estos errores:

```
PS C:\Users\sjvc5\Proyectos Programacion\Proyecto_IngeSoft\Proyecto\Frontend\chazapp> flutter analyze
⦿ Analyzing chazapp...

info - Constructors for public widgets should have a named 'key' parameter - lib\test_clean_code.dart:7:7 -
      use_key_in_widget_constructors
info - The variable name 'APP_NAME' isn't a lowerCamelCase identifier - lib\test_clean_code.dart:8:16 -
      non_constant_identifier_names
info - Don't invoke 'print' in production code - lib\test_clean_code.dart:13:5 - avoid_print
info - Use 'const' with the constructor to improve performance - lib\test_clean_code.dart:18:15 - prefer_const_constructors
info - Use 'const' with the constructor to improve performance - lib\test_clean_code.dart:19:18 - prefer_const_constructors

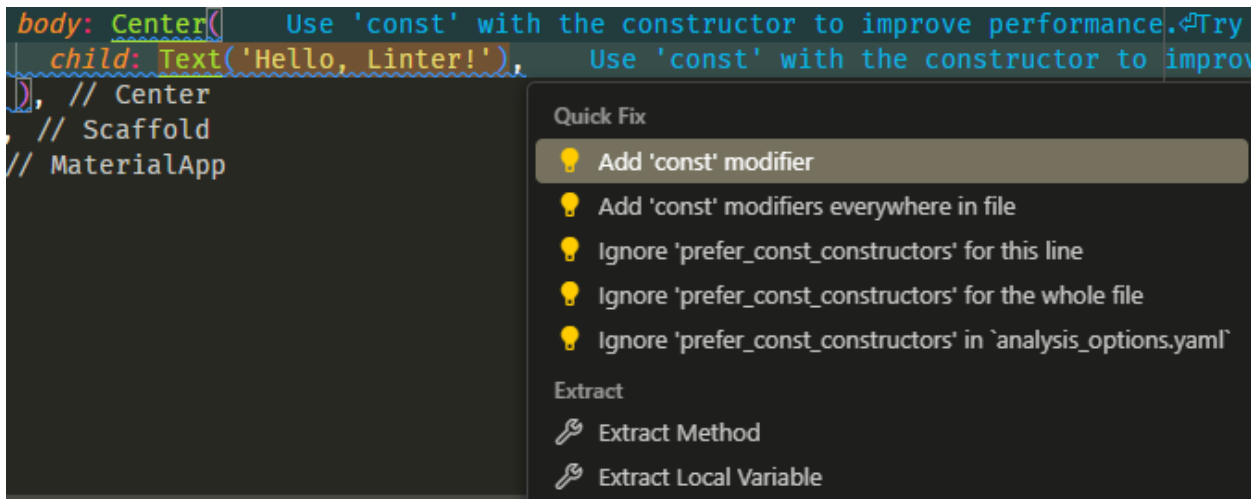
5 issues found. (ran in 4.6s)
```

Además, en la pestaña de ‘Problems’ en la barra inferior de VSCode se puede observar esta misma información.



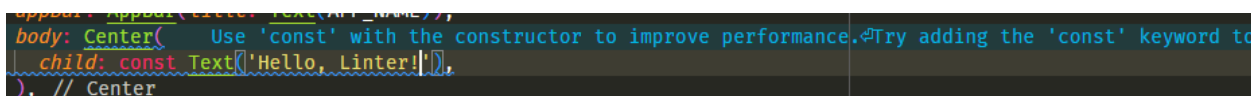
Esto demuestra la utilidad de los linters y el alta increpancia al usuario para escribir código limpio.

Ahora, algunos problemas tienen soluciones rápidas, por ejemplo, el problema de utilizar ‘const’ a la hora de llamar un constructor de un widget sin estado (prefer_const_constructors), puede corregirse mediante el uso de Ctrl+. lo cual abre una pestaña de solución de la advertencia.



Allí se puede solucionar añadiendo const en donde se intenta solucionar, pero el linter también ofrece la opción de solucionar todas las ocurrencias de esta advertencia. Además, también permite ignorar esta advertencia a nivel de la línea, a nivel de archivo o a nivel del proyecto al añadir esta excepción al archivo ‘analysis_options.yaml’.

Inicialmente se intentará cambiar solamente a nivel de línea para este problema:



Aunque se añadió a la línea del widget Text, al crear el widget Center también ocurre la misma advertencia. Por ello, sería más útil realizar la corrección en todo el archivo. Al realizar esto, el const se colocará donde corresponda mejor:

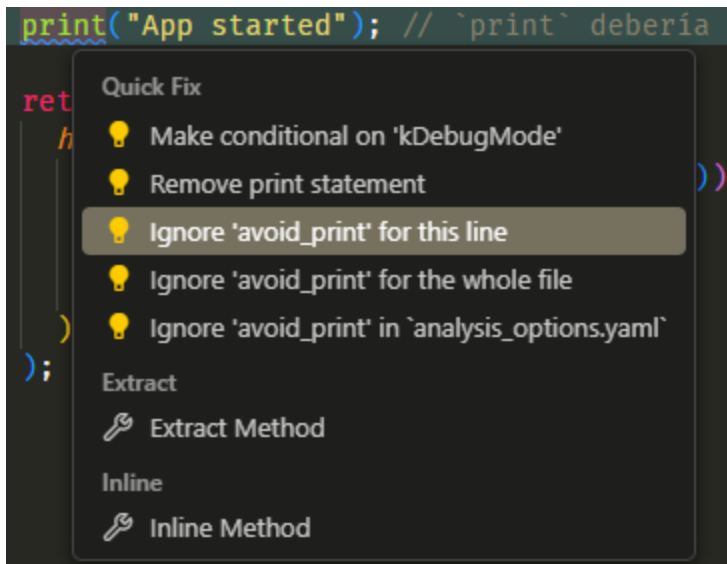
```
AppBar: AppBar(title: Text(AppName)),  
body: const Center(  
  child: Text('Hello, Linter!'),  
), // Center  
// Scaffold
```

Ahora ya no aparece ninguna advertencia. Para comprobar las opciones de ignorar las reglas del linter, vamos a observar el caso de la línea que utiliza 'print'. Esto incumple la regla 'avoid_print'.

```
print("App started"); // `print` debería reemplazarse por `debugPrint` Don't invoke 'print' in production code.
```

En caso de que se quiera ignorar, se puede utilizar nuevamente el comando Ctrl+. lo cual agrega una línea para ignorar solo esta ocurrencia de la advertencia, o escribir este comentario directamente:

```
print("App started"); // `print` debería reemplazarse por `debugPrint`
```

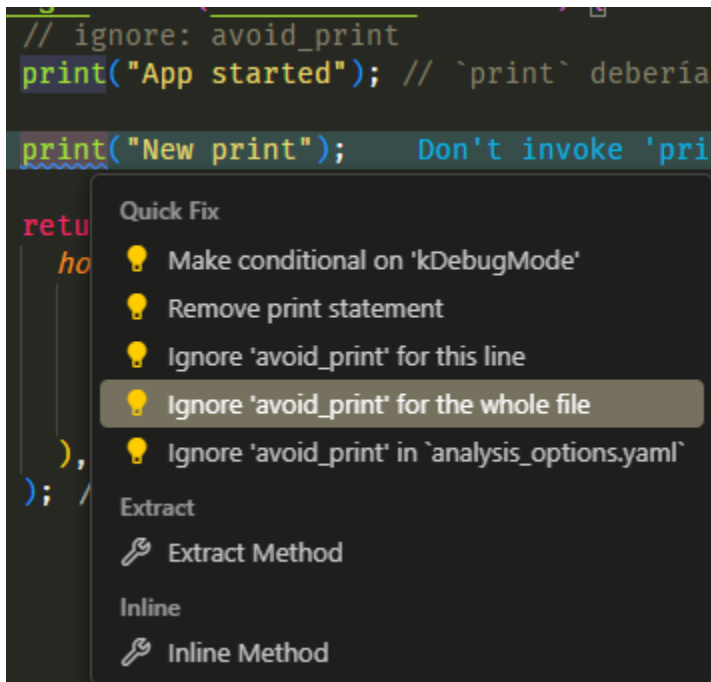


```
// ignore: avoid_print  
print("App started"); // `print` debería reemplazarse por `debugPrint`
```

Al agregar esta línea, ya no aparece la advertencia. Sin embargo, al agregar otro print, vuelve a aparecer, debido a que solo ignora la siguiente línea:


```
// ignore: avoid_print  
print("App started"); // `print` debería reemplazarse por `debugPrint`  
print("New print"); Don't invoke 'print' in production code. Try us
```

En caso de que se quisiera ignorar esta advertencia en todo el archivo, se debería escoger en su lugar la siguiente opción al llamar Ctrl+.::



De esta manera, se agrega un comentario al principio del archivo que le indica al linter no considerar la regla en este archivo:

```
// ignore_for_file: avoid_print

import 'package:flutter/material.dart';

Run | Debug | Profile
void main() {
  runApp(MyApp());
}

class MyApp extends StatelessWidget { Constructors for public widgets should have a named 'key' parameter.❗Try adding a key.
  // Incumple la regla de nombres de constantes en mayúsculas
  final String APP_NAME = "Linter Test"; The variable name 'APP_NAME' isn't a lowerCamelCase identifier.❗Try changing it to lowerCamelCase.
  @override
  Widget build(BuildContext context) {
    print("App started"); // `print` debería reemplazarse por `debugPrint`
    print("New print");

    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(title: Text(APP_NAME)),
        body: const Center(
          child: Text('Hello, Linter!'),
        ), // Center
      ), // Scaffold
    ); // MaterialApp
  }
}
```

Finalmente, para corregir los errores corregibles del proyecto se puede utilizar el comando ‘dart fix --apply’:

```
PS C:\Users\sjvc5\Proyectos Programacion\Proyecto_IngeSoft\Proyecto\Frontend\chazapp> dart fix --apply
Computing fixes in chazapp...
Applying fixes...

lib\test_clean_code.dart
  prefer_const_constructors • 1 fix
  use_key_in_widget_constructors • 1 fix

2 fixes made in 1 file.
```

En el archivo, ya se corrigieron las líneas que incumplían las reglas prefer_const_constructors y use_key_in_widget_constructors:

```

1 // ignore_for_file: avoid_print
2
3 import 'package:flutter/material.dart';
4
5 Run | Debug | Profile
6 void main() {
7   runApp(const MyApp());
8 }
9
10 class MyApp extends StatelessWidget {
11   // Incumple la regla de nombres de constantes en mayúsculas
12   final String APP_NAME = "Linter Test"; The variable name 'APP_NAME' isn't a lowerCamelCase identifier. Try changi
13
14   const MyApp({super.key});
15
16   @override
17   Widget build(BuildContext context) {
18     print("App started"); // `print` debería reemplazarse por `debugPrint`
19     print("New print");
20
21     return MaterialApp(
22       home: Scaffold(
23         appBar: AppBar(title: Text(APP_NAME)),
24         body: const Center(
25           child: Text('Hello, Linter!'),
26         ), // Center
27       ), // Scaffold
28     ); // MaterialApp
29   }
30 }

```

Sin embargo, la línea 11 no fue corregida, esto es debido a que, aunque incumple la regla “non_constant_identifiers_names”, el linter no sabe específicamente cómo se debería solucionar, ya que no sabe dónde comienza o finaliza una palabra, o si el valor debería ser una constante. Por ello, queda resaltada la advertencia para que el desarrollador aplique los cambios que considere pertinentes.

```

1 // ignore_for_file: avoid_print
2
3 import 'package:flutter/material.dart';
4
5 Run | Debug | Profile
6 void main() {
7   runApp(const MyApp());
8 }
9
10 class MyApp extends StatelessWidget {
11   // Incumple la regla de nombres de constantes en mayúsculas
12   final String appName = "Linter Test";
13
14   const MyApp({super.key});
15
16   @override
17   Widget build(BuildContext context) {
18     print("App started"); // `print` debería reemplazarse por `debugPrint`
19     print("New print");
20
21     return MaterialApp(
22       home: Scaffold(
23         appBar: AppBar(title: Text(appName)),
24         body: const Center(
25           child: Text('Hello, Linter!'),
26         ), // Center
27       ), // Scaffold
28     ); // MaterialApp
29   }
30 }

```

Al cambiar el nombre de la variable a appName, la advertencia desaparece.

Revisión Cruzada de Código

Como parte del proceso de desarrollo, realizamos una revisión cruzada del código escrito por cada integrante del equipo. El objetivo de esta evaluación es determinar si el código es fácil de leer, comprender y modificar, además de verificar si cumple con los estándares acordados. Cada estudiante ha analizado el trabajo de sus compañeros y ha compartido una breve narración sobre su experiencia, destacando fortalezas y posibles mejoras en términos de claridad, estructura y mantenibilidad. A continuación, se presentan las observaciones de cada integrante.

Ana González

Inicialmente al no estar familiarizada con NestJS y la arquitectura hexagonal, me resultó difícil comprender la estructura general del proyecto sin una guía inicial, sin embargo, al hablar con mis compañeros y revisar a profundidad el código del Backend, pude ver que la arquitectura elegida permite tener una estructura con una clara separación de las responsabilidades. Otro punto a su favor es que, por ejemplo, fue fácil entender cómo se interactúa con la API gracias a que el código escrito es bastante conciso y robusto y la elección de Typescript me pareció bastante certera para evitar errores de tipado. Conuerdo con mis compañeros en el sentido de que tener una mejor documentación podría facilitar la comprensión inicial del Back, en especial para aquellos que no estamos muy familiarizados con la tecnología.

Daniel Soracipa

El backend estructurado por idea de mi compañero me parece organizado teniendo en cuenta la arquitectura y carpetas que planteo para el proyecto, lo que facilita su mantenimiento y escalabilidad. Me pareció correcta la separación de responsabilidades y el uso de los controladores y servicios. Además, la integración con Swagger permite una interacción clara con los endpoints. Como posible mejora, se podría agregar documentación del flujo con el que se trabaja al crear una carpeta como usuario, esto también ayudaría mucho a quienes no conocen mucho del framework y la arquitectura.

Por otro lado, en el código del frontend escrito en Dart, realmente no tengo ningún conocimiento de este tipo de programación móvil, pero puedo ver que el código está organizado siguiendo la arquitectura planteada en el proyecto y hasta cierto punto fácil de entender aunque no tenga nociones de Dart, además de que tienen comentarios que pueden ayudar a entender cuál es el propósito de algunas líneas de código en los archivos.

Juan Medina

En el código del backend se puede ver que es consistente, lo que facilita su comprensión y legibilidad. Además, se adhiere al patrón de arquitectura elegido (patrón hexagonal) y cuenta con documentación detallada de los endpoints a través de Swagger. La única posible mejora sería estandarizar el nombramiento de las ramas para nuevas funcionalidades.

En cuanto al código del frontend, escrito en Dart (lenguaje con el que no estoy completamente familiarizado, pero que encuentro similar a Java), se puede apreciar una implementación clara de la conexión con el backend. Asimismo, la aplicación de programación

orientada a interfaces contribuye a la escalabilidad y al desacoplamiento del código respecto a su implementación, una posible mejora que puede surgir es la documentación manejarla en el idioma inglés.

Samuel Vargas

Al revisar el código del backend, encontré que su comprensión resultó difícil debido a que está desarrollado en una tecnología con la que no estoy familiarizado. Sin embargo, más allá de la barrera tecnológica, noté que la ausencia de comentarios y documentación interna dificulta aún más su entendimiento. En varias secciones del código, las funciones y variables no tienen descripciones claras, lo que hace que sea necesario analizar en detalle su funcionamiento en lugar de comprenderlo de manera rápida.

Como oportunidad de mejora, recomendaría agregar comentarios explicativos en puntos clave del código, utilizar nombres de variables y funciones más descriptivos, y, si es posible, contar con una breve documentación sobre la estructura general del backend. Estas acciones facilitarían la colaboración y la mantenibilidad del proyecto.