



# Buenas prácticas con estilo

**Rafael Bernal**

Programador Frontend {pixel developer &  
Stencil Advocate} con amor por el diseño  
[JavaScript, TypeScript, Angular, ...]

**autentia**





## Indice

### Teoría

- Clean Code

### Práctica

- ESLint
- Prettier
- Husky
- Lint-Staged



## Uso de var, let y const

Evitar a toda costa el uso de **var**. ¿Por qué?

Al usar **var** estamos dando a esa variable un alcance global, también te permite volver a definir variables con el mismo nombre. Puede llegar a provocar confusión.

La principal diferencia de **var** con **let** y **const** es que éstas se definen para un ambito en concreto. Acotando su rango de acción.

Si sabemos que el valor de esa variable no cambiará durante la ejecución del programa optemos siempre por el uso de **const**. También es una medida que podemos tomar para evitar que esa variable cambie de valor.

Es buena práctica inicializar las variables al definirlas.



```
var foo = 1

{
  const foo = 2
  let bar = 3
  console.log(foo) // 2
}

console.log(foo) // 1
console.log(bar) // is not defined
```



Los nombres, que se entiendan

Evitar: abreviaturas, nombres impronunciables y guiones bajos / medios.

Las variables y objetos deben comenzar con minúsculas y los nombres de clases u objetos de clases con mayúsculas.

El uso del estilo CamelCase debería ser tu aliado en estos casos

Los nombres te deben decir algo, y muy importante: **¡¡Fáciles de encontrar!!**



```
let d = 1 // bad
```

```
let daysSinceCreation = 1 // good
```

```
let controllerForEfficientHandlingOfStrings // bad
```



## Comparar variables

Comparación estricta de datos. ¿Por qué?  
Esta no realiza conversión de tipos y nos asegura que las dos variables coinciden tanto en tipo como contenido.

Uso de la doble negación. La manera más elegante en comprobar un valor.

```
let foo = 0 // number
let bar = '0' // string
foo == bar // comparación no estricta
foo === bar // comparación estricta

!!0 === false // true
```



# Funciones

Las funciones representan acciones. Construimos el nombre de las funciones con verbos y sustantivos.

La sencillez es un pilar fundamental en las funciones. Cada función debe hacer exactamente lo que su nombre indica.

La primera regla en la funciones es que tienen que tener un tamaño reducido. Nuestras funciones deben tener un tamaño reducido. **¡¡5 líneas es bien!!**

No abusemos de los argumentos. En general se deberían limitar a 3 argumentos. Hay expertos en la materia que dicen que las funciones nunca deberían recibir argumentos. **Locos...**

**Sin efectos secundarios** o transparencia referencial: promete hacer una cosa pero también hace otras cosas ocultas. Esto es importante sobre todo cuando trabajamos con frameworks como por ejemplo Angular.





Comentarios

## ¡¡El código no se comenta!!

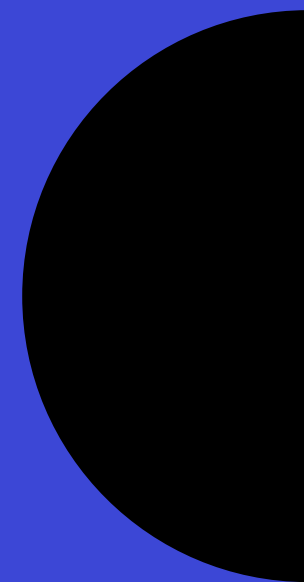
Si nos vemos en la necesidad de comentar el código, entonces posiblemente estemos en la necesidad de reescribir nuestro código mejor.

Y si no queda otra que comentar, mejor comenta el porqué lo haces y no lo que hace tu código.






**Principio DRY**



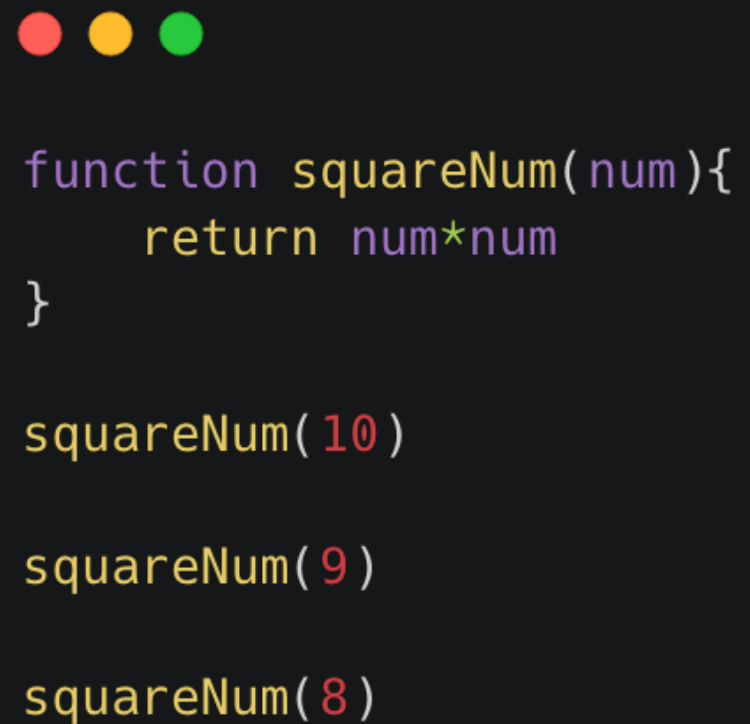
**don't repeat yourself**





Como buen programador deberías intentar no repetirte.  
La duplicación del código es una de las principales características del *smell code*. Implementar el principio **DRY** es parte fundamental en todo proyecto.

Analiza y extrae el código duplicado en una función sería una solución por ejemplo.

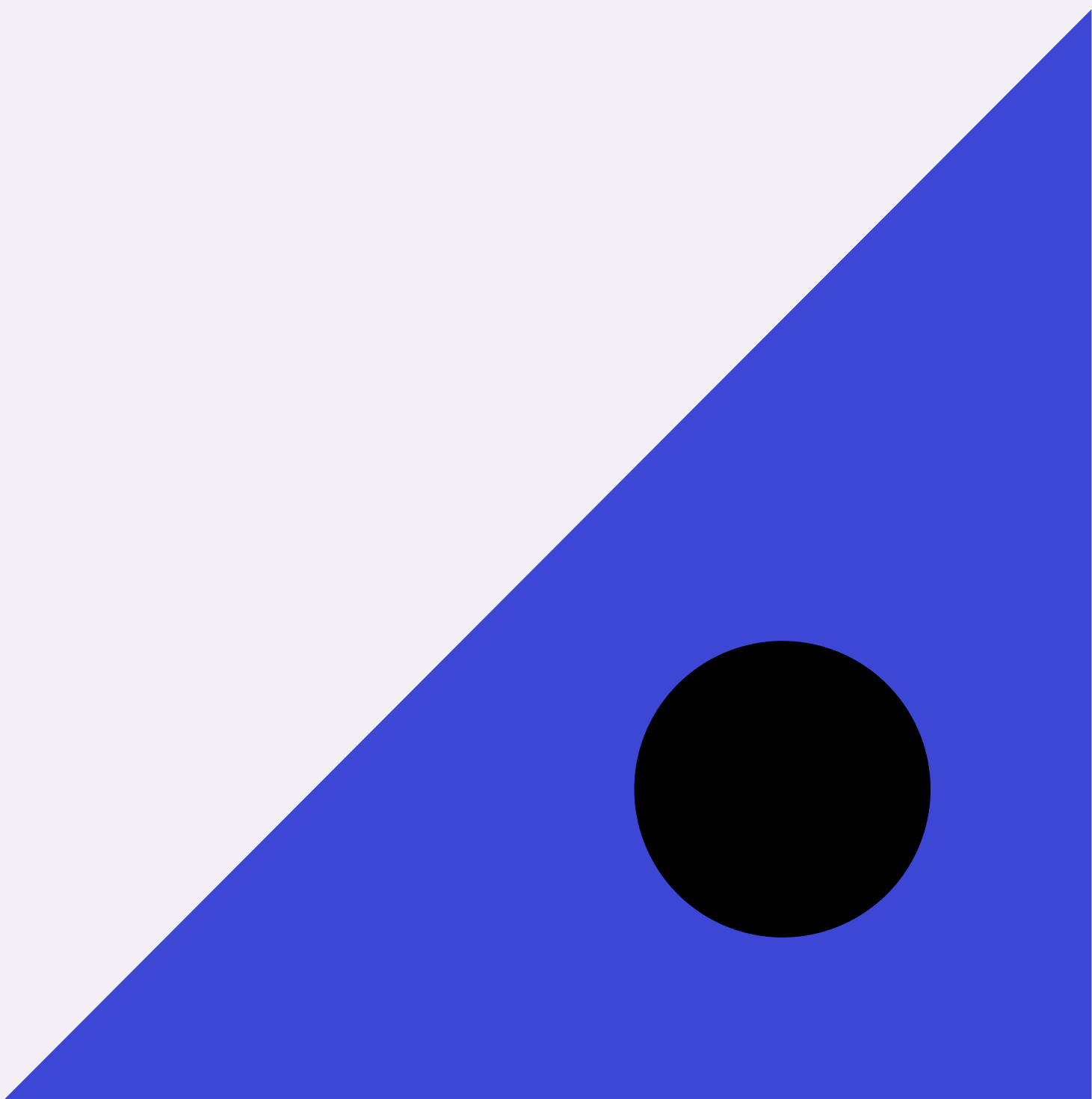


```
function squareNum(num){  
    return num*num  
}  
  
squareNum(10)  
  
squareNum(9)  
  
squareNum(8)
```

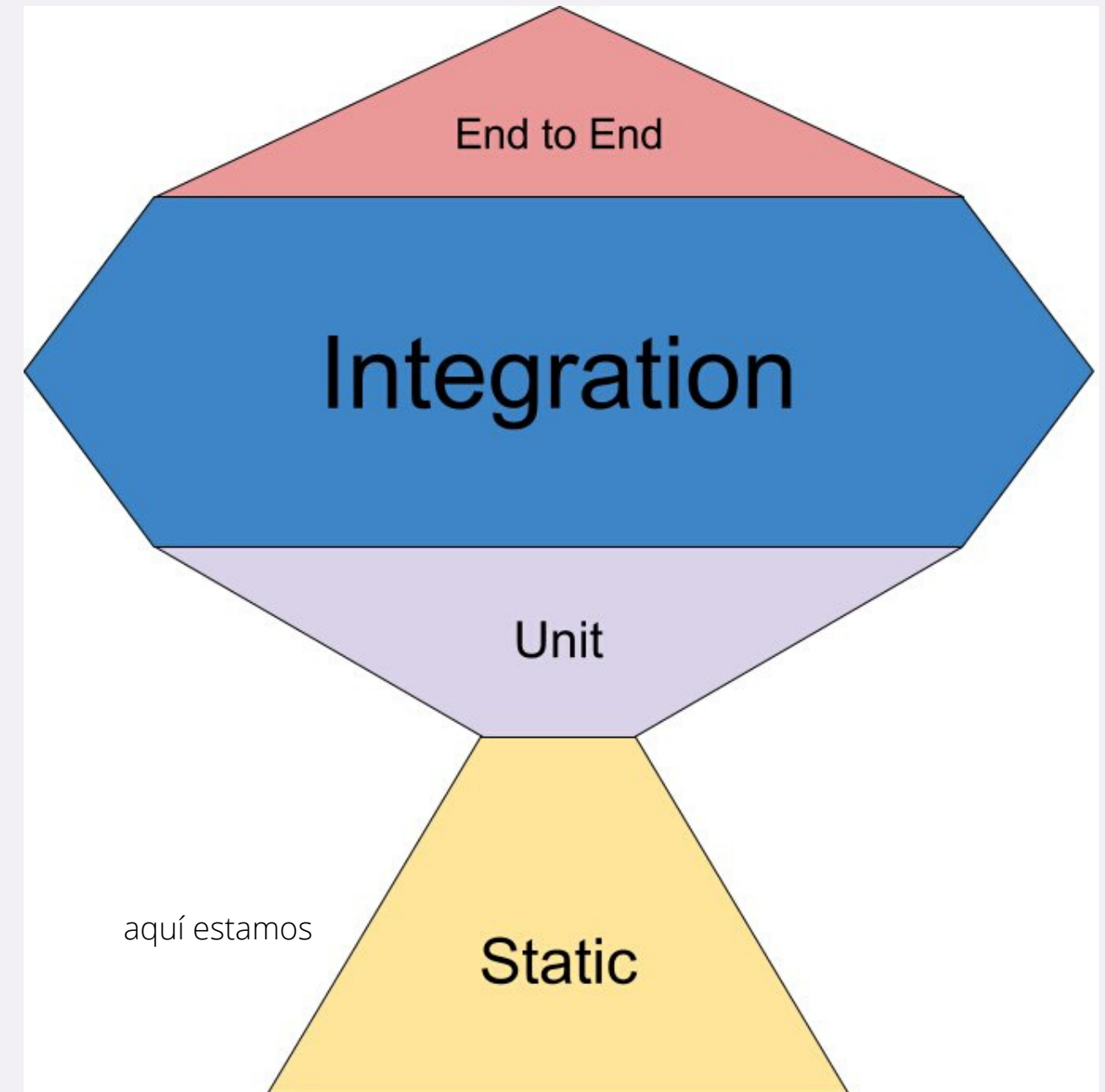


Y para terminar con la parte teórica

## Principios **S.O.L.I.D**

- **S**: Single responsibility principle o Principio de responsabilidad única
  - **O**: Open/closed principle o Principio de abierto/cerrado
  - **L**: Liskov substitution principle o Principio de sustitución de Liskov
  - **I**: Interface segregation principle o Principio de segregación de la interfaz
  - **D**: Dependency inversion principle o Principio de inversión de dependencia
- 

# Iniciar un nuevo proyecto



Kent C. Dodds

# ESlint

*linting para JavaScript*



Definiendo...

**ESlint** es un *linter* que examina nuestro código Javascript siguiendo una serie de normas y criterios previamente definidos por el usuario y nos obliga a seguir esas convenciones. Estos estilos definidos favorecerán una mayor legibilidad y obtendremos un código de mayor calidad.

### ¿Pero qué es un linter?

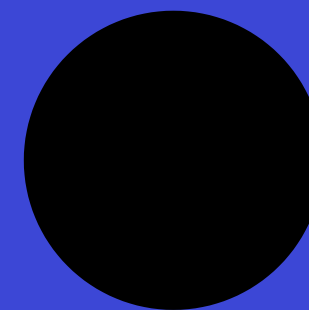
Los linters son herramientas que examinan nuestro código y nos obligan a corregir la sintaxis, nos ayuda a encontrar código incorrecto y nos impone unas buenas prácticas a la de crear nuestro código. Aquí empieza nuestra hoja de estilos propia.






**ESlint**

# Configurando y ejecutando ESLint




Para instalar **ESlint**:



```
npm install --save-dev eslint
```

Esto lo instalará como dependencia de nuestro proyecto y ya estaría listo para usarlo con el comando:



```
npx eslint .
```

Posteriormente en la raíz de nuestro proyecto crearemos el archivo **.eslintrc.js** o directamente en nuestro terminal tecleamos:



```
npx eslint --init
```

// .eslintrc.js

**rules:** es el apartado donde incluiremos las reglas que validarán nuestro código

```
{
  "env": {
    "browser": true,
    "es2020": true
  },
  "extends": "eslint:recommended",
  "parserOptions": {
    "ecmaVersion": 2019,
    "sourceType": "module",
  },
  "rules": {
    "init-declarations": ["error", "always"],
    "strict": ["error", "never"],
    "valid-typeof": "error",
    "no-unsafe-negation": "error",
    "no-unused-vars": "error",
    "no-unexpected-multiline": "error",
    "no-undef": "error"
  }
}
```




EsLint tiene muchas, cientos de reglas en su página web

**<https://eslint.org/docs/rules/>**

Pero siempre podrás usar unos paquetes ya predefinidos que incluirán las más comunes, las estándar para todo proyecto javascript

**[\\*https://docs.w3cub.com/eslint/rules/](https://docs.w3cub.com/eslint/rules/)**

```
{
  "env": {
    "browser": true,
    "es2020": true
  },
  "parserOptions": {
    "ecmaVersion": 2019,
    "sourceType": "module",
  },
  "extends": ["eslint:recommended"],
  "rules": {
    "strict": ["error", "never"],
  }
}
```



Para terminar de configurar **ESlint** y trabajar con él sea más fácil. Haremos que se ejecute con un como un script de nuestro proyecto node. Para eso incluimos en nuestro **package.json** lo siguiente:



```
"scripts": {  
  ...,  
  "lint": "eslint . --ignore-path .gitignore"  
}
```

# Prettier

*formatea tu código*



Definiendo...

**Prettier** es un formateador de código opinionado que tiene reglas predefinidas para formatear y sangrar el código.

Es decir, formatea nuestro código automáticamente según las directrices impuestas en su configuración.

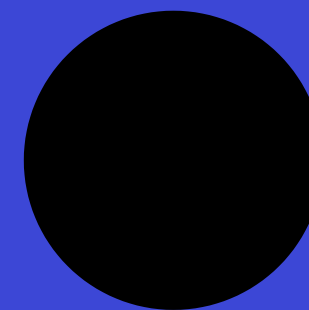
No lo veas como una herramienta a competir con **ESlint**, todo lo contrario, están hechas para trabajar juntas.





Prettier

Configurando y ejecutando Prettier



Para instalar **Prettier**:

```
npm install --dev-dependency prettier
```

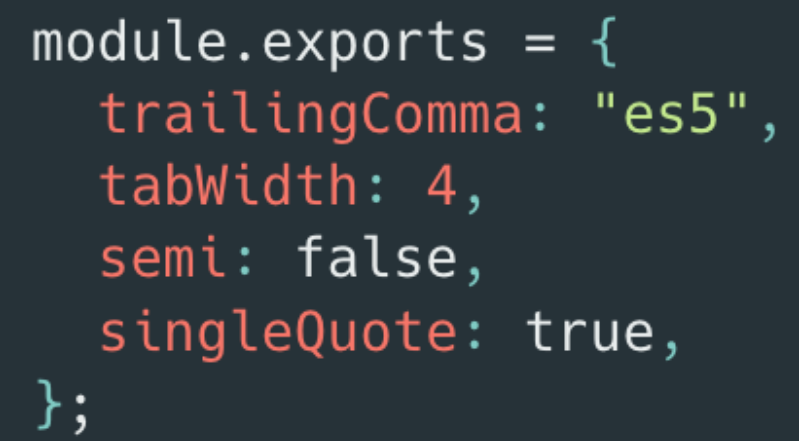
Esto lo instalará como dependencia de nuestro proyecto y ya estaría listo para usarlo con el comando:

```
npx prettier ejemplo.js
```


Al igual que hicimos con el linteador necesitamos un archivo de configuración donde indicaremos como queremos que trabaje. Para ello creamos el archivo **.prettierrc.js** en la raíz de nuestro proyecto.

---

// .prettierrc.js



```
module.exports = {  
  trailingComma: "es5",  
  tabWidth: 4,  
  semi: false,  
  singleQuote: true,  
};
```



Llegados a este punto vamos a crear un guión para que nuestro proyecto pueda ejecutar **Prettier** más fácilmente.

Para eso incluimos en nuestro **package.json** lo siguiente:



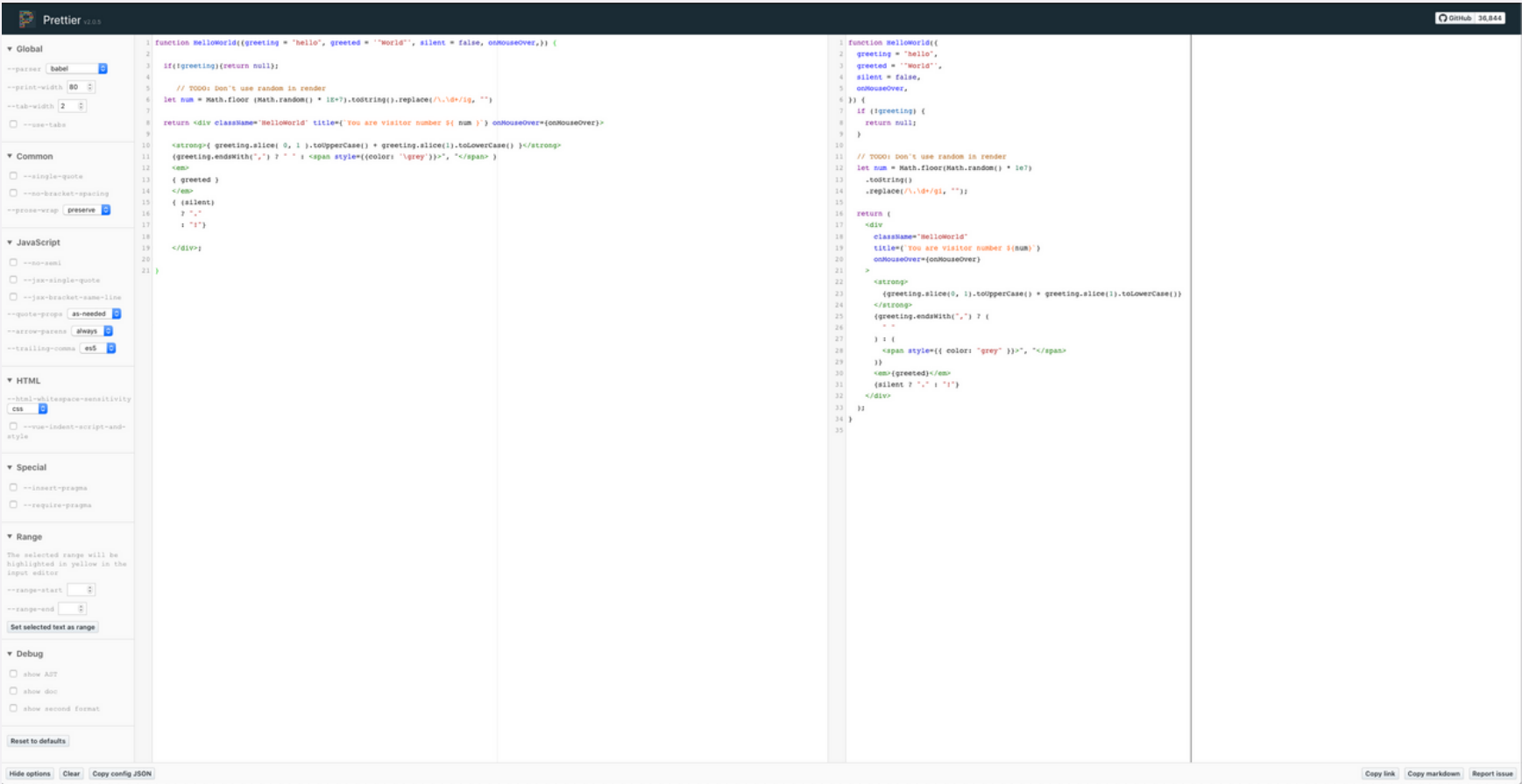
```
"scripts": {  
  ...  
  "lint": "eslint --ignore-path .gitignore .",  
  "format": "prettier --ignore-path .gitignore --write \"**/*.+(js|json)\"",  
},
```




Prettier nos ofrece desde su web una forma fácil de formar nuestro archivo de configuración

<https://prettier.io/playground/>


Donde a través de su playground nos será más fácil elegir las normas.






¿Es posible que Eslint y Prettier entre en colisión? Buena pregunta, y si no te lo habías planteado... **¡deberías!**

Cuando es una sola regla la problemática no pasa nada, lo arreglamos a mano. Pero que ocurre cuando las que entran en conflicto son reglas que **ESlint** ha incluido para trabajar con Babel, Angular, React... ahí la cosa se complica, ¿no? Instala entre tus dependencias:

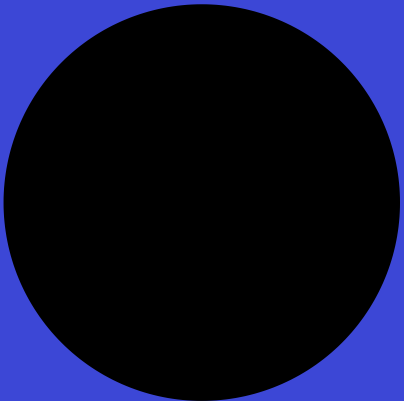


```
npm install --save-dev eslint-config-prettier eslint-plugin-prettier
```

No olvides configurar de nuevo el archivo **.eslintrc.js**,  
añadiendo el nuevo paquete instalado

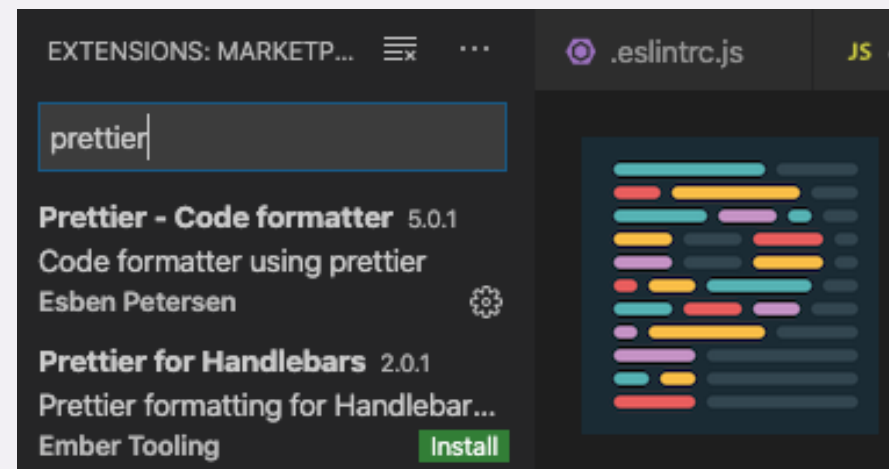


```
extends: ["eslint:recommended", "plugin:prettier/recommended"],
```



## BONUS PARA USUARIOS DE VSCODE !!

Vamos a configurar nuestro editor para que al guardar ejecute automáticamente **Prettier**. Desde el marketplace de Visual Studio Code instalamos el plugin de Prettier



Luego indicamos lo siguiente en el archivo de configuración de Visual Studio Code

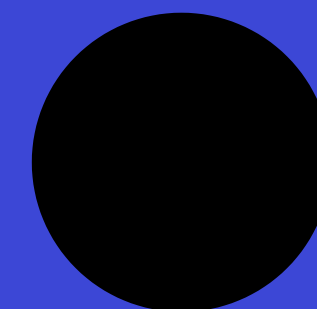
```
{  
  "editor.defaultFormatter": "esbenp.prettier-vscode",  
  "editor.formatOnSave": true  
}
```




## **ESlint & Prettier**

**Llegados a este punto...**

**¡Unamos fuerzas!**





Ya tenemos instalados y configurados tanto **ESlint** como **Prettier**.  
Configuremos nuestro package.json para que quede tal que así



```
"scripts": {  
  ...  
  "lint": "eslint --ignore-path .gitignore .",  
  "prettier": "prettier --ignore-path .gitignore \"**/*.+(js|json)\",  
  "format": "npm run prettier -- --write",  
  "check-format": "npm run prettier -- --list-different",  
  "validate": "npm run lint && npm run check-format"  
}
```



# Husky

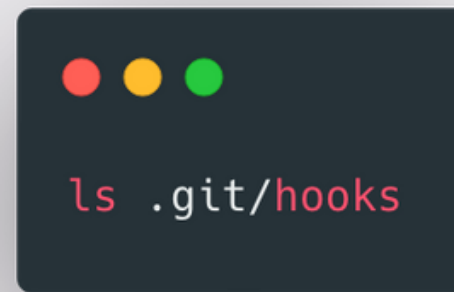
*automatiza tus commits*



## Definiendo...

**Husky** es un paquete npm muy popular que se creó para ayudarnos con los hooks de git como el commit o el push.

¿Qué es un **hook** de git? Cuando inicializas Git en un proyecto, automáticamente viene con una característica llamada hooks.



Si quisiéramos asegurarnos antes de que se ejecute una acción antes de usar el comando git commit, como que su código está correctamente alineado y formateado, podríamos escribir un hook en el pre-commit de Git.

Escribir eso manualmente probablemente no sería divertido. También sería un desafío distribuir y asegurarnos de que los hooks se instalaran en las máquinas de otros desarrolladores.

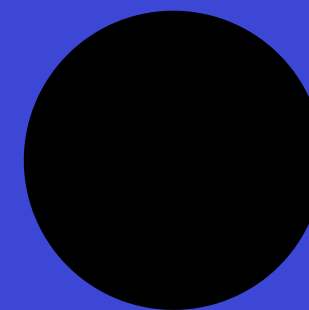
Estos son algunos de los retos que Husky pretende abordar.





**Husky**

# Configurando y ejecutando Husky





# Lint-staged

*otra vuelta de tuerca a la hora de ejecutar tus  
commits*



Definiendo...

**Lint-staged** ejecuta comandos a la hora de confirmar un commit al igual que Husky, pero con una notable diferencia.

Lint-staged sólo comprueba los archivos en la zona stage de git. Si se encuentra un error Husky detendrá el proceso. De esta forma hacemos que trabajen conjuntamente **Husky** y **Lint-staged**.



Para instalar Lint-staged:

```
npm install --save-dev lint-staged
```

En esta caso Lint-staged al igual que Husky nos permite configurarlo de dos formas diferentes. Creando un archivo en la raíz del proyecto que se llame **.lintstagedrc** o directamente dentro del archivo package.json que será nuestra elección

```
"devDependencies": {  
  ....  
},  
"husky": {  
  "hooks": {  
    "pre-commit": "lint-staged"  
  }  
},  
"lint-staged": {  
  "*.{js,html,scss,json}": [  
    "npm run format"  
  ]  
},
```

# Rafael Bernal

Programador Frontend {pixel developer & Stencil Advocate} con amor por el diseño [JavaScript, TypeScript, Angular, ...]

twitter: @rbernalber

linkedin: rafabernalber

**autentia**

“

