Buenas prácticas para escribir commits en Git

midu.dev/buenas-practicas-escribir-commits-git

Escribir buenos mensajes de commit es importante para que el histórico de tu proyecto sea legible, fácilmente escaneable y, claro, entendible por cualquier persona que participe en el proyecto.

Por ello voy a darte 6 reglas para escribir un buen mensaje de commit:

1. Usa el verbo imperativo (`Add`, `Change`, `Fix`, `Remove`, ...)

Aunque el mensaje puede sonar un poco borde, el verbo presente es una forma de expresar la acción que se realiza en el commit. Por ejemplo, `Add` significa que se añade un nuevo archivo, `Change` significa que se modifica un archivo existente y `Fix` significa que se arregla un bug.

¿Quién no se ha encontrado con un historial de commits así? Fuente: xkcd.com

Sé que muchas veces estamos tentados a escribirlo en pasado *"Added…"*, *"Fixed…"* o *"Removed…"* pero **cada commit hay que entenderlo como una instrucción para cambiar el estado del proyecto**. Dicho de otro modo, el verbo presente nos permite saber qué estado queremos que el proyecto se encuentre en el momento de añadir el commit.

Sólo hay que ver también los mensajes de commit que el propio Git nos añade (al hacer merge de una rama usa `Merge branch`).

Lo mejor es que el mensaje del commit complete esta frase: *"Si aplico este commit, entonces este commit…"*

- …add a new search feature
- …fix a problem with the topbar

- …change the default system color
- …remove a random notification

2. No uses punto final ni puntos suspensivos en tus mensajes

Usar puntuación, más allá de las comas, es innecesario a la hora de crear un buen mensaje de commit. Cada carácter cuenta a la hora de crear un buen mensaje de commit así que no lo desperdicies con puntos innecesarios.

```
git commit -m "Add new search feature." # ❌
git commit -m "Fix a problem with the topbar..." # ❌
git commit -m "Change the default system color" # ✔️
```

¿Por qué? **El primer mensaje de commit es el título del commit.** Y según las reglas de puntuación para títulos, tanto en castellano como en inglés, estos no llevan puntuación final. Sobre los puntos suspensivos… ¡Nuestros commits no deberían tener ningún suspense! Deben ser una instrucción clara y concisa.

Si te fijas, los commits que genera GitHub no tienen puntos suspensivos ni punto final en ningún caso.

3. Usa como máximo 50 carácteres para tu mensaje de commit

Sé corto y conciso. Si tienes mucho que explicar… seguramente es que tu commit hace demasiadas cosas. ¿Puedes separarlo en diferentes commits? Pues hazlo.

Haz que el mensaje sea claro, directo y que realmente refleje los cambios que lleva.

```
git commit -m "Add new search feature and change typography to improve
performance" # ❌
git commit -m "Add new search feature" # ✔️
git commit -m "Change typography to improve performance" # ✔️
```

4. Añade todo el contexto que sea necesario en el cuerpo del mensaje de commit

A veces necesitas proveer de más contexto a tu commit. Para ello, en lugar de saturar el sumario del commit, añade información que sea necesaria en el cuerpo del mensaje.

Puedes lograrlo usando `git commit -m "Add summary of commit" -m "This is a message to add more context."` pero en estos casos lo mejor es que uses directamente `git commit` de esta forma:

```
git commit
```

Y con el editor, podrás añadir un mensaje de commit con saltos de línea fácilmente.

> Como hemos comentado antes, el primer mensaje de commit es el título del commit. Pero el resto de mensajes son el cuerpo y, por lo tanto, sí debes usar todas las reglas de puntuación que tendrían un texto normal.

5. Usa un prefijo para tus commits para hacerlos más semánticos

Cuando un proyecto crece, es necesario que existan ciertas reglas para que el historial sea legible. Para ello, puedes añadir un prefijo para darle más significado a los commits que realizas. A esto se le llama *commits semánticos* y se haría de la siguiente manera:

```
<tipo-de-commit>[scope]: <descripcion>
```

Por ejemplo:

```
feat: add new search feature
^--^  ^--------------------^
|     |
|     └--> # Descripción de los cambios
|
└------> # Tipo del cambio
```

En monorepositorios multipaquete, puedes añadir también la información del paquete que es afectado por el commit. Se le conoce como `scope` y sería de la siguiente forma:

```
feat(backend): add filter for cars
fix(web): remove wrong color
```

Sobre el tipo de cambio, lo mínimo es diferenciar entre `fix` y `feat` pero existen diferentes convenciones. Una de ellas, bastante famosa, es la convención que sigue el framework Angular.

Estos serían los prefijos:

- **feat**: Una nueva característica para el usuario.
- **fix**: Arregla un bug que afecta al usuario.
- **perf**: Cambios que mejoran el rendimiento del sitio.
- **build**: Cambios en el sistema de build, tareas de despliegue o instalación.
- **ci**: Cambios en la integración continua.
- **docs**: Cambios en la documentación.
- **refactor**: Refactorización del código como cambios de nombre de variables o funciones.
- **style**: Cambios de formato, tabulaciones, espacios o puntos y coma, etc; no afectan al usuario.
- **test**: Añade tests o refactoriza uno existente.

Otra ventaja muy importante de utilizar commits semánticos es que podrás leer el historial de commits para publicar nuevas versiones de un paquete, desplegar nuevas versiones de una aplicación o generar un CHANGELOG con todos los cambios.

6. Considera usar utilidades para hacer commit

Puedes usar `husky` para ejecutar scripts o comandos antes de realizar diferentes acciones sobre el repositorio, gracias a los hooks de git. Por ejemplo, puedes ejecutar los tests antes de subir los cambios al repositorio remoto.

```
# instalamos las dependencias
npm install husky --save-dev
npx husky install

# iniciamos husky en nuestro repositorio
npm set-script prepare "husky install"
npm run prepare

# creamos un hook para que pase los tests antes de hacer commit
npx husky add .husky/pre-push "npm test"
git add .husky/pre-push

git commit -m "Keep calm and commit"
git push -u origin master # los tests se ejecutarán antes de realizar el push
```

Con `commitlint` puedes asegurarte de que los commits sean semánticos, legibles y sigan una convención que elijas.

Para instalar `commitlint`:

```
# Install commitlint cli and conventional config
npm install --save-dev @commitlint/{config-conventional,cli}
# For Windows:
npm install --save-dev @commitlint/config-conventional @commitlint/cli

# Añadir hook para revisar el mensaje de commit
npx husky add .husky/commit-msg 'npx --no-install commitlint --edit "$1"'
```

Puedes usar sistemas como `conventional-changelog` para leer el CHANGELOG y generar nuevas versiones o publicar paquetes. También con `commitizen` puedes usar una línea de comandas para que te haga elegir el tipo de commit y así no tengas que depender de realizar esto manualmente en el mensaje de commit.