https://chris.beams.io/posts/git-commit/

How to Write a Git Commit Message

31 Aug 2014 | revision history

Introduction | The Seven Rules | Tips

Introduction: Why good commit messages matter

If you browse the log of any random Git repository, you will probably find its commit messages are more or less a mess. For example, take a look at these gems from my early days committing to Spring:

```
$ git log --oneline -5 --author cbeams --before "Fri Mar 26 2009"

e5f4b49 Re-adding ConfigurationPostProcessorTests after its brief removal
2db0f12 fixed two build-breaking issues: + reverted ClassMetadataReadingV
147709f Tweaks to package-info.java files
22b25e0 Consolidated Util and MutableAnnotationUtils classes into existir
7f96f57 polishing
```

Yikes. Compare that with these more recent commits from the same repository:

```
$ git log --oneline -5 --author pwebb --before "Sat Aug 30 2014"

5ba3db6 Fix failing CompositePropertySourceTests
84564a0 Rework @PropertySource early parsing logic
e142fd1 Add tests for ImportSelector meta-data
887815f Update docbook dependency and generate epub
ac8326d Polish mockito usage
```

Which would you rather read?

The former varies in length and form; the latter is concise and consistent. The former is what happens by default; the latter never happens by accident.

While many repositories' logs look like the former, there are exceptions. The Linux kernel and Git itself are great examples. Look at Spring Boot, or any repository managed by Tim Pope.

The contributors to these repositories know that a well-crafted Git commit message is the best way to communicate *context* about a change to fellow developers (and indeed to their future selves). A diff will tell you *what* changed, but only the commit message can properly tell you *why*. Peter Hutterer makes this point well:

> Re-establishing the context of a piece of code is wasteful. We can't avoid it completely, so our efforts should go to reducing it [as much] as possible. Commit messages can do exactly that and as a result, *a commit message shows whether a developer is a good collaborator*.

If you haven't given much thought to what makes a great Git commit message, it may be the case that you haven't spent much time using `git log` and related tools. There is a vicious cycle here: because the commit history is unstructured and inconsistent, one doesn't spend much time using or taking care of it. And because it doesn't get used or taken care of, it remains unstructured and inconsistent.

But a well-cared for log is a beautiful and useful thing. `git blame`, `revert`, `rebase`, `log`, `shortlog` and other subcommands come to life. Reviewing others' commits and pull requests becomes something worth doing, and suddenly can be done independently. Understanding why something happened months or years ago becomes not only possible but efficient.

A project's long-term success rests (among other things) on its maintainability, and a maintainer has few tools more powerful than his project's log. It's worth taking the time to learn how to care for one properly. What may be a hassle at first soon becomes habit, and eventually a source of pride and productivity for all involved.

In this post, I am addressing just the most basic element of keeping a healthy commit history: how to write an individual commit message. There are other important practices like commit squashing that I am not addressing here. Perhaps I'll do that in a subsequent post.

Most programming languages have well-established conventions as to what constitutes idiomatic style, i.e. naming, formatting and so on. There are

variations on these conventions, of course, but most developers agree that picking one and sticking to it is far better than the chaos that ensues when everybody does their own thing.

A team's approach to its commit log should be no different. In order to create a useful revision history, teams should first agree on a commit message convention that defines at least the following three things:

Style. Markup syntax, wrap margins, grammar, capitalization, punctuation. Spell these things out, remove the guesswork, and make it all as simple as possible. The end result will be a remarkably consistent log that's not only a pleasure to read but that actually *does get read* on a regular basis.

Content. What kind of information should the body of the commit message (if any) contain? What should it *not* contain?

Metadata. How should issue tracking IDs, pull request numbers, etc. be referenced?

Fortunately, there are well-established conventions as to what makes an idiomatic Git commit message. Indeed, many of them are assumed in the way certain Git commands function. There's nothing you need to re-invent. Just follow the seven rules below and you're on your way to committing like a pro.

The seven rules of a great Git commit message

> *Keep in mind: This has all been said before.*

1. Separate subject from body with a blank line
2. Limit the subject line to 50 characters
3. Capitalize the subject line
4. Do not end the subject line with a period
5. Use the imperative mood in the subject line
6. Wrap the body at 72 characters
7. Use the body to explain *what* and *why* vs. *how*

For example:

```
Summarize changes in around 50 characters or less

More detailed explanatory text, if necessary. Wrap it to about 72
characters or so. In some contexts, the first line is treated as the
subject of the commit and the rest of the text as the body. The
blank line separating the summary from the body is critical (unless
you omit the body entirely); various tools like `log`, `shortlog`
and `rebase` can get confused if you run the two together.

Explain the problem that this commit is solving. Focus on why you
are making this change as opposed to how (the code explains that).
Are there side effects or other unintuitive consequences of this
```

```
change? Here's the place to explain them.

Further paragraphs come after blank lines.

 - Bullet points are okay, too

 - Typically a hyphen or asterisk is used for the bullet, preceded
   by a single space, with blank lines in between, but conventions
   vary here

If you use an issue tracker, put references to them at the bottom,
like this:

Resolves: #123
See also: #456, #789
```

1. Separate subject from body with a blank line

From the `git commit` manpage:

> Though not required, it's a good idea to begin the commit message with a single short (less than 50 character) line summarizing the change, followed by a blank line and then a more thorough description. The text up to the first blank line in a commit message is treated as the commit title, and that title is used throughout Git. For example, Git-format-patch(1) turns a commit into email, and it uses the title on the Subject line and the rest of the commit in the body.

Firstly, not every commit requires both a subject and a body. Sometimes a single line is fine, especially when the change is so simple that no further context is necessary. For example:

```
Fix typo in introduction to user guide
```

Nothing more need be said; if the reader wonders what the typo was, she can simply take a look at the change itself, i.e. use `git show` or `git diff` or `git log -p`.

If you're committing something like this at the command line, it's easy to use the `-m` option to `git commit`:

```
$ git commit -m"Fix typo in introduction to user guide"
```

However, when a commit merits a bit of explanation and context, you need to write a body. For example:

```
Derezz the master control program
```

```
MCP turned out to be evil and had become intent on world domination.
This commit throws Tron's disc into MCP (causing its deresolution)
and turns it back into a chess game.
```

Commit messages with bodies are not so easy to write with the `-m` option. You're better off writing the message in a proper text editor. If you do not already have an editor set up for use with Git at the command line, read this section of Pro Git.

In any case, the separation of subject from body pays off when browsing the log. Here's the full log entry:

```
$ git log
commit 42e769bdf4894310333942ffc5a15151222a87be
Author: Kevin Flynn <kevin@flynnsarcade.com>
Date:   Fri Jan 01 00:00:00 1982 -0200

 Derezz the master control program

 MCP turned out to be evil and had become intent on world domination.
 This commit throws Tron's disc into MCP (causing its deresolution)
 and turns it back into a chess game.
```

And now `git log --oneline`, which prints out just the subject line:

```
$ git log --oneline
42e769 Derezz the master control program
```

Or, `git shortlog`, which groups commits by user, again showing just the subject line for concision:

```
$ git shortlog
Kevin Flynn (1):
      Derezz the master control program

Alan Bradley (1):
      Introduce security program "Tron"

Ed Dillinger (3):
      Rename chess program to "MCP"
      Modify chess program
      Upgrade chess program

Walter Gibbs (1):
      Introduce protoype chess program
```

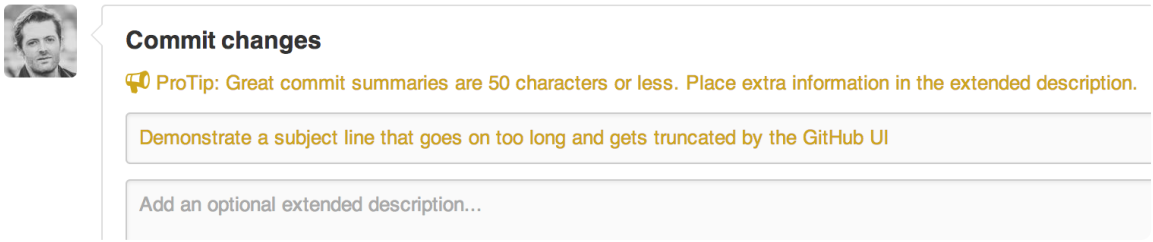
There are a number of other contexts in Git where the distinction between subject line and body kicks in—but none of them work properly without the blank line in between.

2. Limit the subject line to 50 characters

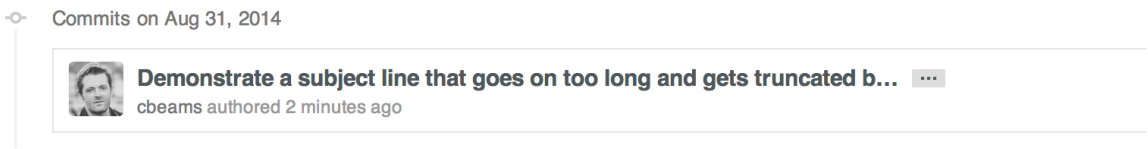
50 characters is not a hard limit, just a rule of thumb. Keeping subject lines at this length ensures that they are readable, and forces the author to think for a moment about the most concise way to explain what's going on.

> *Tip: If you're having a hard time summarizing, you might be committing too many changes at once. Strive for atomic commits (a topic for a separate post).*

GitHub's UI is fully aware of these conventions. It will warn you if you go past the 50 character limit:



And will truncate any subject line longer than 72 characters with an ellipsis:



So shoot for 50 characters, but consider 72 the hard limit.

3. Capitalize the subject line

This is as simple as it sounds. Begin all subject lines with a capital letter.

For example:

- Accelerate to 88 miles per hour

Instead of:

- accelerate to 88 miles per hour

4. Do not end the subject line with a period

Trailing punctuation is unnecessary in subject lines. Besides, space is precious when you're trying to keep them to 50 chars or less.

Example:

- Open the pod bay doors

Instead of:

- Open the pod bay doors.

5. Use the imperative mood in the subject line

Imperative mood just means “spoken or written as if giving a command or instruction”. A few examples:

- Clean your room
- Close the door
- Take out the trash

Each of the seven rules you’re reading about right now are written in the imperative (“Wrap the body at 72 characters”, etc.).

The imperative can sound a little rude; that’s why we don’t often use it. But it’s perfect for Git commit subject lines. One reason for this is that **Git itself uses the imperative whenever it creates a commit on your behalf**.

For example, the default message created when using `git merge` reads:

```
Merge branch 'myfeature'
```

And when using `git revert`:

```
Revert "Add the thing with the stuff"

This reverts commit cc87791524aedd593cff5a74532befe7ab69ce9d.
```

Or when clicking the “Merge” button on a GitHub pull request:

```
Merge pull request #123 from someuser/somebranch
```

So when you write your commit messages in the imperative, you’re following Git’s own built-in conventions. For example:

- Refactor subsystem X for readability
- Update getting started documentation
- Remove deprecated methods
- Release version 1.0.0

Writing this way can be a little awkward at first. We’re more used to speaking in the *indicative mood*, which is all about reporting facts. That’s why commit messages often end up reading like this:

- Fixed bug with Y
- Changing behavior of X

And sometimes commit messages get written as a description of their contents:

- More fixes for broken stuff
- Sweet new API methods

To remove any confusion, here's a simple rule to get it right every time.

A properly formed Git commit subject line should always be able to complete the following sentence:

- If applied, this commit will *your subject line here*

For example:

- If applied, this commit will *refactor subsystem X for readability*
- If applied, this commit will *update getting started documentation*
- If applied, this commit will *remove deprecated methods*
- If applied, this commit will *release version 1.0.0*
- If applied, this commit will *merge pull request #123 from user/branch*

Notice how this doesn't work for the other non-imperative forms:

- If applied, this commit will *fixed bug with Y*
- If applied, this commit will *changing behavior of X*
- If applied, this commit will *more fixes for broken stuff*
- If applied, this commit will *sweet new API methods*

> *Remember: Use of the imperative is important only in the subject line. You can relax this restriction when you're writing the body.*

6. Wrap the body at 72 characters

Git never wraps text automatically. When you write the body of a commit message, you must mind its right margin, and wrap text manually.

The recommendation is to do this at 72 characters, so that Git has plenty of room to indent text while still keeping everything under 80 characters overall.

A good text editor can help here. It's easy to configure Vim, for example, to wrap text at 72 characters when you're writing a Git commit. Traditionally, however, IDEs have been terrible at providing smart support for text wrapping in commit messages (although in recent versions, IntelliJ IDEA has finally gotten better about this).

7. Use the body to explain what and why vs. how

This commit from Bitcoin Core is a great example of explaining what changed and why:

```
commit eb0b56b19017ab5c16c745e6da39c53126924ed6
Author: Pieter Wuille <pieter.wuille@gmail.com>
Date:   Fri Aug 1 22:57:55 2014 +0200

   Simplify serialize.h's exception handling

   Remove the 'state' and 'exceptmask' from serialize.h's stream
   implementations, as well as related methods.

   As exceptmask always included 'failbit', and setstate was always
   called with bits = failbit, all it did was immediately raise an
   exception. Get rid of those variables, and replace the setstate
   with direct exception throwing (which also removes some dead
   code).

   As a result, good() is never reached after a failure (there are
   only 2 calls, one of which is in tests), and can just be replaced
   by !eof().

   fail(), clear(n) and exceptions() are just never called. Delete
   them.
```

Take a look at the full diff and just think how much time the author is saving fellow and future committers by taking the time to provide this context here and now. If he didn't, it would probably be lost forever.

In most cases, you can leave out details about how a change has been made. Code is generally self-explanatory in this regard (and if the code is so complex that it needs to be explained in prose, that's what source comments are for). Just focus on making clear the reasons why you made the change in the first place—the way things worked before the change (and what was wrong with that), the way they work now, and why you decided to solve it the way you did.

The future maintainer that thanks you may be yourself!

Tips

Learn to love the command line. Leave the IDE behind.

For as many reasons as there are Git subcommands, it's wise to embrace the command line. Git is insanely powerful; IDEs are too, but each in different ways. I use an IDE every day (IntelliJ IDEA) and have used others extensively (Eclipse), but I have never seen IDE integration for Git that could begin to match the ease and power of the command line (once you know it).

Certain Git-related IDE functions are invaluable, like calling `git rm` when you delete a file, and doing the right stuff with `git` when you rename one. Where everything falls apart is when you start trying to commit, merge, rebase, or do sophisticated history analysis through the IDE.

When it comes to wielding the full power of Git, it's command-line all the way.

Remember that whether you use Bash or Zsh or Powershell, there are tab completion scripts that take much of the pain out of remembering the subcommands and switches.

Read Pro Git

The Pro Git book is available online for free, and it's fantastic. Take advantage!

IMHO the CC makes good commits for machine/programs, but not for developer itself. Could you share your thoughts?

Reply • Share ›

donraj • 2 months ago • edited

It is okay advice, The **only important point** is: Use the body to explain what and why vs. how

And its extremely anti-climatic at the end when it comes down to loving the cli.

1. Life is too short for the command line!
2. Further what is this **crap about capitalization and period and N number of characters**? Imagine facebook started rejecting your informal facebook posts for not writing in caps and ending in a period. it would be funny. Software engineers are creators not some the people who sit behind the counter of DMV offices.
3. The author loves the command line; Its fine old man, there is no point dragging this tradition along!
The git log is a tree. There is no point in rendering the tree in an ASCII box when you have advanced version control TREE-VIZUALIZATIONS available.

4. Please do this:
git config --global core.editor "subl -n -w"
It will save you a lot of time - bit by bit - with every commit.

1 Reply • Share ›

Apostolis Anastasiou ➚ donraj • a month ago • edited

Life is too short to NOT use the command line.
I dont want to be the douche that tells you "you're obviously wrong, you just haven't used it enough", but if you think that command line slows you down, you probably haven't used it enough or haven't used it right. So, yes, Ill be the douche.

Anyone who has used the command line, prefers using it. Some of the reasons are:
1. You dont have to launch every single application explicitely
2. You dont have to wait these 3s (the least) until each application launches
3. You dont have to make window management, you just have all apps running on a finger snap, like Thanos with his gauntlet, except imagine how much more efficient he would be if he had a command line.
4. Ever been in a situation where you search for the right tool or action but can't find it anywhere while searching for minutes or even hours?
In terminal there's no such thing. If you know what command you want, you just write it down. Forgot it? Then

command you want, you just write it down. Forgot it? Then all you have to do is

```
man `application_name`
press `?` or `/` and search the keywords, even with
regular expressions
```

5. You want to install a CLI application? All you have to do is go to your browser, search for the application, go into their website, download the executable, install it ... Oh wait, pardon me, that is if you're NOT in the terminal. In terminal you just run

```
// Arch based
sudo pacman -S `app_name`
// OR
yay -S `app_name`
// Ubuntu based
sudo apt install `app_name`
// or any package manager you have in your system
```

6. In command line, you have COUNTLESS of options. Anything you want, you can do it. Anything. You can use multiple commands together, and combine many applications at the same command with pipes. You can print the output to a file, automate an action from a script, write the output to an ssh server, make regex actions. ANYTHING you can imagine. GUI on the other hand is just restrictive to its own tools. Understandable, but that's all you have.
7. All of the above combined, make GUI far less efficient. You spend your time more on searching things, the tools you want, the fancy GUI theme and the window managing, than actually doing the job you want to do. In terminal you dont have to do anything to use the application, you just write the right commands and boom! Magic.

If you use the command line, you're not a "cool man", you're not a "nerd", neither an "old man" as you call him. You're just doing your job efficiently and right without having to struggle with a bunch of GUI applications where you have to deal with countless different GUI designs while searching for the right tool everytime you want to do something.

If you DONT use the terminal, you're probably doing something wrong. Especially if you consider yourself an experienced man in the job.

2 ▲ ▼ • Reply • Share ›

Maxolasersquad ➚ donraj • 2 months ago

1. Life is to short to use the wrong tool. GUIs 100% have their
place, but so does the cli. Use the right tool for the right job. If you
don't think the cli has its place you are not maximizing your efficiency as a developer.

2. Writing git commits that are
formatted well goes a long way to helping your eyes skim, which is
critical when reading through a long git log. Facebook posts are usually
informal and so an informal style is acceptable. Git logs are business
and should be written like a professional wrote them. Imagine a book
publisher accepting books written with sloppy usage of punctuation, upper
and lower case, and an inconsistent style. When you are a professional it should show in all of your work.

8 · Reply · Share ›

Nastase Alex • 2 months ago

Thank you!

· Reply · Share ›

spearkkk • 2 months ago

Thank you, it is helpful for learning git. 😄

· Reply · Share ›

Sam Bronson • 3 months ago

Another reason you might have trouble fitting your summary in 50 characters: You needed a long directory name for disambiguation.

1 · Reply · Share ›

Siamak A.Motlagh • 3 months ago

Great article. Thank you

1 · Reply · Share ›

Грубый Оттуда • 4 months ago

great article, thanks! somehow i think past tense makes more sense in the commit message.

1 · Reply · Share ›

Francute ➚ **Грубый Оттуда** • 3 months ago

It depends on how do you see it. If you see it like "What I've done in this commit", it makes more sense to use the past tense. (Your focus is in Who).
But if you see it like "What is this commit doing if I apply it",

or "What is going to do GIT if I apply this commit", makes more sense to use the imperative one. (Your focus is in What)
So, just choose an style with your team and stick to it, or follow the repository style convention. If you can't find a convention in some repository, use the imperative as a default.

1 ^ v • Reply • Share ›

Andras Racz ➚ Грубый Оттуда • 4 months ago

it is always longer than the imperative, without being more informative

2 ^ v • Reply • Share ›

Joseph ➚ Грубый Оттуда • 4 months ago

Same. It doesn't actually give a single reason as to why to use imperative other than that git itself does.
If anything, when it's talking about how it should follow the structure "If applied, this commit will release version 1.0.0", I feel it would make sense for it to just have it in past tense so you don't have to follow such an arbitrary convention.

^ v • Reply • Share ›

Dev Verma ➚ Грубый Оттуда • 4 months ago

I feel so too

^ v • Reply • Share ›

Ellen • 4 months ago

Thank you. I'm teaching this to my students.

2 ^ v • Reply • Share ›

Marcus Eisele • 5 months ago

Coming back here from time to time and now thought that I could leave a comment.
Often read, Often forwarded. Just a really nice article. Probably one of the best/straight forward articles about this topic.

Fun fact: Today I realized what kind of codebase you must be working on having a look at the the commit messages in detail and also I recognized pwebb :)

2 ^ v • Reply • Share ›

Greg Aburime • 5 months ago

Beautiful article, nicely written. This has helped me a lot.

^ v • Reply • Share ›

Gabriel Reynoso • 6 months ago

Chris, what do you think about GitKraken?

^ v • Reply • Share ›

Chema Olea • 6 months ago

Just to clarify, what you call imperative, I would say is actually infinitive.

"If applied, this commit will INFINITIVE FORM"
If applied this commit will fix the bug.

Imperative would be:
"Please Chris, IMPERATIVE FORM"
Please Chris, fix the bug.

Though they are written in the same way. Correct me if I am wrong, english is not my native language.

3 ^ v • Reply • Share ›

s47 ➚ **Chema Olea** • 5 months ago

"this commit will fix the bug." is the future tense. The future tense and imperative are not written the same way, because the conjugate in the future tense for regular verbs results in saying or writing "will" before the infinitive: I will go, you will eat, he will leave, we will celebrate, etc. Does that help?

^ v • Reply • Share ›

Christian R. Conrad ➚ **s47** • 2 months ago

The future tense consists of "will" (or "shall" etc) plus the infinitive of the actual verb in question. So "fix the bug" IS infinitive; "this will" is what makes the whole thing future tense. Does that help?

^ v • Reply • Share ›

Nik • 6 months ago

Very well written article. I could use your article, to change the opinions of my team to use the proper / verbose commenting method.

^ v • Reply • Share ›

Frederico Canoeira • 6 months ago

Very good.

But I didn't understood this:

"However, when a commit merits a bit of explanation and context, you need to write a body. For example:"

Could you give me a realistic example (with command) where we need to write a body?

^ v • Reply • Share ›

Omar Suliman • 7 months ago

we can summarize it with be clear and descriptive, and we can add this rules to it (8) Write commit messages in present tense, not in past tense ,and thank you its helpful article

Reply • Share ›

Bimochan Shrestha • 7 months ago

For advanced commit messages you may wanna use this plugin. https://github.com/sbimocha...

Reply • Share ›

Greg • 7 months ago

Thank you for the article. If you are interested in a very short introduction into Git and its comparison to other version control systems, I recommend this blog post.

Reply • Share ›

Brent Engelbrecht • 8 months ago

Thanks for this post Chris. I've never really thought about commit messages in this way until now!

Reply • Share ›

GOWRI SANKAR • 9 months ago

Thanks for the tip on "writing imperative commit messages." and Git Pro book is indeed great!

Reply • Share ›

Sebastian Wilgosz • 9 months ago

Thanks for the article, it really covers the basic assumptions really well. I also like adding a tags for commit titles to easily filter them out and search through the git history (like: fix, refactor, feat). It makes the commits even more readable.

Inspired on this I took it a one step further on my blog and extended this information by explaining how to make well-defined commit messages in the productive way using git commit templates.

https://driggl.com/blog/a/h...

If you care about your commits, that might be interesting for you as well.

Reply • Share ›

Ryoma • 10 months ago

Can I share this in my blog by chinese?

Reply • Share ›

Chris Beams Mod ➚ Ryoma • 10 months ago

Yes, and please share the link here when the post is up

Thanks!

26 ▲ ▼ • Reply • Share ›

Ryoma ➚ Chris Beams • 7 months ago

Thank you for your license, this is my post in chinese: https://blog.ryoma.top/post...

1 ▲ ▼ • Reply • Share ›

Paul • 10 months ago

I'm a bit confused about *why* to use the imperative. Reading the example commit made me feel like the author wanted me to do these things.

1 ▲ ▼ • Reply • Share ›

Mikko Rantalainen ➚ Paul • 4 months ago

The imperative style is because git handles patches, not revisions. Using imperative style is a lot about mindset where you may end up re-ordering patches, dropping some patch, inserting a new patch between some existing pathes, etc. If you label your commits as history you'll automatically limit the way you think about git.

Using imperative style also makes it easier to understand what will happen if you revert (instead of dropping) any given patch.

Note that if you never bother to really learn about `rebase -i`, all of the above may seem just a chore for you.

3 ▲ ▼ • Reply • Share ›

Jeff Byrnes ➚ Paul • 10 months ago

The reasoning for imperative is that it matches the grammatical mood of messages that Git generates automatically, e.g., merges, reversions, and the like.

Additionally, it matches the tense of how the log indicates what's happening: each commit *does* the thing their diff indicates. I.e., as you look at a particular commit, it is not a reflection of what *happened*, but what *is happening*.

▲ ▼ • Reply • Share ›

DeProgrammer ➚ Jeff Byrnes • 10 months ago

That's really not a good reason, though--that boils down to "tradition." A good reason to use the imperative form is that it generally allows you to compress information into fewer characters--in English, at least.

19 ▲ ▼ • Reply • Share ›

Jeff Byrnes ↗ DeProgrammer
• 10 months ago

I view is as being consistent w/ Git, not “tradition”, though I can understand why you’d read it that way.

Git, like much of the programming world, is English-centric, for good or ill.

20 ^ v • Reply • Share ›

Ido Ran • 10 months ago

What do you think about writing ticket/PR number in the commit