git

An opinionated hotchpotch of practices

v2.0.0 | 2023.08.02 | Étienne Boisseau-Sierra

git

An opinionated hotchpotch of practices

Go global

Messages

Collaborating

Hooks

History

git

An opinionated hotchpotch of practices

Go global

Messages

Collaborating

Hooks

History

Leverage ~/.gitconfig

… to set things up once and for all

```
cat ~/.gitconfig | grep some-of-the-interesting stuff

[user]
        name = Étienne Boisseau-Sierra
        email = etienne.boisseau-sierra@biomodal.com
        signingkey = 6163437C60DE02B

[alias]
        ht = log --graph --oneline --decorate --color -20
        oops = commit --gpg-sign --amend --no-edit

[includeIf "gitdir:~/perso/"]
        path = ~/.gitconfig_perso

[core]
        editor = vim

        # import global .gitignore
        excludesfile = ~/.gitignore_global
```

git

An opinionated hotchpotch of practices

Go global

Messages

Collaborating

Hooks

History

Conventional commit

… is a good convention

```
<type>[optional scope]: <description>

[optional body]

[optional footer(s)]
```

The commit contains the following structural elements, to communicate intent to the consumers of your library:

1. **fix:** a commit of the *type* `fix` patches a bug in your codebase (this correlates with `PATCH` in Semantic Versioning).
2. **feat:** a commit of the *type* `feat` introduces a new feature to the codebase (this correlates with `MINOR` in Semantic Versioning).
3. **BREAKING CHANGE:** a commit that has a footer `BREAKING CHANGE:`, or appends a `!` after the type/scope, introduces a breaking API change (correlating with `MAJOR` in Semantic Versioning). A BREAKING CHANGE can be part of commits of any *type*.
4. *types* other than `fix:` and `feat:` are allowed, for example **@commitlint/config-conventional** (based on the **Angular convention**) recommends `build:`, `chore:`, `ci:`, `docs:`, `style:`, `refactor:`, `perf:`, `test:`, and others.
5. *footers* other than `BREAKING CHANGE: <description>` may be provided and follow a convention similar to **git trailer format**.

Additional types are not mandated by the Conventional Commits specification, and have no implicit effect in Semantic Versioning (unless they include a BREAKING CHANGE). A scope may be provided to a commit's type, to provide additional contextual information and is contained within parenthesis, e.g., `feat(parser): add ability to parse arrays`.

www.conventionalcommits.org/en/v1.0.0/

Structuring commit messages

Limit the subject line to 50 characters

Capitalize the subject line

Do not end the subject line with a period

Use the imperative mood

You can use prefixes like "feat: Foo bar"

Wrap the body at 72 characters

Use the body to explain what and why (vs. how)

Link to relevant GitHub object (or JIRA story)

```
(cool-feature) $ git commit --signoff

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: SS-1234
Signed-off-by: Jane Doe <jane.doe@unipart.io>
```

Separate subject from body with a blank line

Signoff your commits (shorthand: -s)

Adapted from chris.beams.io/posts/git-commit/

git

An opinionated hotchpotch of practices

Go global

Messages

Collaborating

Hooks

History

Once it's up there…

… it's forever!

Each commit on ***main*** should be production-ready

… squash-merge PR is the correct way

Sign your work

GPG, sign-off, and show off!

```
$ git commit --signoff
$ git log -1 | grep sign

Signed-off-by: Étienne Boisseau-Sierra <etienne.boisseau-sierra@biomodal.com>
```

Sign your work

GPG, sign-off, and show off!

```
$ git commit --signoff
$ git log -1 | grep sign

Signed-off-by: Étienne Boisseau-Sierra <etienne.boisseau-sierra@biomodal.com>



$ git \
    -c user.name="Your Nemesis" \
    -c user.email="your.nemesis@evil.corp" \
    commit
```

git

An opinionated hotchpotch of practices

Go global

Messages

Collaborating

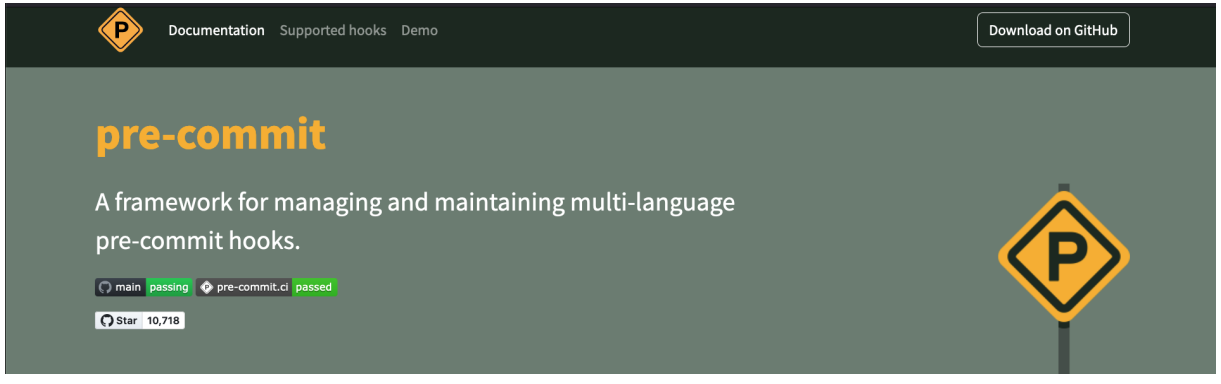
Hooks

History

Hooks

```
$ ls .git/hooks

applypatch-msg.sample
fsmonitor-watchman.sample
pre-applypatch.sample
pre-merge-commit.sample
pre-rebase.sample
prepare-commit-msg.sample
update.sample
commit-msg.sample
post-update.sample
pre-commit.sample
pre-push.sample
pre-receive.sample
push-to-checkout.sample
```



git

An opinionated hotchpotch of practices

Go global

Messages

Collaborating

Hooks

History

Rewrite history

```
(cool-feature) $ git rebase --interactive main

pick 2ade05f feat: Map filename to subdir
pick 57e1f3a feat: Define entry point to move file to subdir
pick 59ecc29 feat: Add file sorting to subdir as new step in pipeline
pick 9ab2404 refactor: Streamline code
pick c9be836 fix: Make `move_file_to_subdir` idempotent
pick 62ba741 fix: Continue even if file cannot be moved to correct subdir
pick 2c96cf0 fix: Allow overwriting FASTQs with new ones

# Rebase 6ca8c1d..2c96cf0 onto 6ca8c1d (7 commands)
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup [-C | -c] <commit> = like "squash" but keep only the previous
#                    commit's log message, unless -C is used, in which case
#                    keep only this commit's message; -c is same as -C but
#                    opens the editor
# x, exec <command> = run command (the rest of the line) using shell
# b, break = stop here (continue rebase later with 'git rebase --continue')
# d, drop <commit> = remove commit
# l, label <label> = label current HEAD with a name
# t, reset <label> = reset HEAD to a label
# m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
#         create a merge commit using the original merge commit's
#         message (or the oneline, if no original merge commit was
#         specified); use -c <commit> to reword the commit message
# These lines can be re-ordered; they are executed from top to bottom.
…
```