Effective pull requests and other good practices for teams using github

by **David Winterbottom** on Saturday, 20 October 2012

I work at an agency where we pay $200 a month to Github for their platinum plan. This article is a summary of an internal talk I gave on making the most of our subscription.

There's nothing original here: it's just a collection of tips that I've harvested over the last few years. I'm publishing this article mainly so I have something to refer future employees to.

Use pull requests

Pull requests are an excellent tool for fostering code review. If you're using Github for team projects, you should be using these extensively.

Many people don't realise that you can make pull requests between two branches of the same repository (the so-called "shared repository model"). For team projects, this is preferable over the "fork and pull" model because it's simpler: there are less branches and remotes to keep track of.

A good practice is for someone else to merge your code into the mainline, ensuring 2 sets of eyeballs review each feature. This is simple to organise when working in pairs, but in larger teams you may need a system for determining who reviews what.

Sample workflow

Here's a sample workflow demonstrating the use of pull requests.

Work on a story

Create a new branch for the current story you are working on:

```
(master) $ git checkout -b feature/masquerading
```

It's important to use a new branch for pull requests for several reasons:

- It allows you to submit multiple pull requests without confusion. The classical Github gotcha is to continue committing to a pull request branch after making the initial request. When these commits are pushed to the remote, they will become part of the original pull request which often ends up conflating unrelated functionality.
- When your pull request is merged into the target branch, the maintainer may decide to rebase your commits to avoid a merge commit, or to squash the commits into a single coherent commit. If your pull request was from your 'master' branch, you will encounter problems when merging the target branch back into your own 'master'. Using a temporary branch means it can be discarded once the pull request is accepted and it doesn't matter that your history was rewritten.

Make changes, run tests, commit etc.

```
(feature/masquerading) $ vim
(feature/masquerading) $ git commit
```

Ask for feedback

If it's a significant or difficult story, you may be unsure if you're on the right track. You could ask for some feedback now by pushing your commits to the remote for others to review:

```
(feature/masquerading) $ git push -u origin feature/masquerading
```

The `-u` option adds an upstream tracking reference to your local branch, meaning that you can run push subsequent commits using `git push` without having to specify the remote and branch names (and run `git pull` without additional arguments).

Now ask for feedback on your project mailing list by circulating either a link to the branch, or a compare view URL. You can use the excellent hub to easily generate compare URLs for sharing:

```
(master) $ git compare master..feature/masquerading
```

This will open your default browser on the compare URL, which you can then copy into an email.

Your fellow developers can now comment either on your commits at line-level, or make more general comments by replying to the mailing list thread.

Submit pull request

After digesting your co-workers' comments, you adjust your approach and make some further commits to your branch.

```
(feature/masquerading) $ vim
(feature/masquerading) $ git commit
```

When the story is complete, push your new commits to the remote:

```
(feature/masquerading) $ git push
```

and use the Github site to create a pull request. A couple of things to be aware of:

- Make sure the destination branch is correct, it may not always be 'master'. If you're using git-flow or similar, the appropriate destination branch may be 'develop' or a release branch.
- Use Github's preview facilties to ensure the pull request is well structured and clear. The description should explain what the pull request contains as well as the thinking behind it. For reference, have a look at this excellent pull request.

Once the pull request is created, you should find someone on your team to review it and send them a link to the request using the project mailing list so anyone else with an interest can take a look.

Code review

Others can now review your branch, make comments on individual lines or on the pull request as a whole: the same process as when you pushed some commits for review earlier.

It's also possible for others to add commits to the pull request by pushing to the same branch:

```
(master) $ git fetch origin
(master) $ git checkout feature/masquerading
(feature/masquerading) $ vim
(feature/masquerading) $ git add .
(feature/masquerading) $ git commit
(feature/masquerading) $ git push origin feature/masquerading
```

Iterate this way until the branch is ready to be merged.

> Github easter egg: add `?w=0` to diff URLs (eg a commit, compare view or pull request) to

> ignore whitespace.

Cleaning your history (optional)

When you're ready to merge, you should first clean up the feature branch.

If there are any commits on the destination branch that aren't on your feature branch then you should rebase to avoid a merge commit. You can check for such commits using:

```
(feature/masquerading) $ git log ..master
```

This shows all commits on 'master' that aren't in your current branch history. If you see any commits here, then rebase the feature branch using:

```
(feature/masquerading) $ git rebase master
```

This replays your commits on top of the new commits from the destination branch so that the merge can be a 'fast-forward'.

> Hang on! Aren't you rewriting history that has been pushed? Yes - that's true. However, wh
> the remote branch is *temporary* as for a pull request, this is ok (as far as I can tell). The pu
> request branch should be deleted once it has been merged and so it shouldn't matter that its
> history is being rewritten before merging.

Next, it may be desirable to squash your commits into larger cohesive commits. You can do this using an 'interactive' rebase:

```
(feature/masquerading) $ git rebase -i master
```

This will open `$EDITOR` with all commits since 'master' listed. You can then reorder and squash these commits, as well as rewording the commit messages. Be careful, this can become quite addictive.

One thing you can do is adjust the final commit message on your feature branch to close the pull request automatically. Simply add 'Fixes #123' (using the ID from the pull request URL) at the bottom of the message.

```
(feature/masquerading) $ git commit --amend
```

Further reading:

- Github issues 2.0: The Next Generation - An overview of Github issues with an explanation of how to close, reopen and reference pull requests from within commit messages.
- Github help: Rebasing
- Github help: Interactive rebasing

Merging

Finally, you can merge your cleaned-up feature branch with a fast-forward merge:

```
(feature/masquerading) $ git checkout master
(master) $ git merge feature/masquerading
```

Alternatively, you can force a merge commit to keep track of which commits came from the feature branch.

```
(feature/masquerading) $ git checkout master
(master) $ git merge --no-ff feature/masquerading
```

When you view the your history as a graph, you will be able to tell which commits were from the feature branch.

Now delete the local and remote feature branches:

```
(master) $ git branch -D feature/masquerading
(master) $ git push origin :feature/masquerading
```

Further reading:

- Using pull requests

Other good practices

Care about your history

Strive for a clean, coherent history. Write good commit messages, adhering to the 50 char summary followed by a longer description. Avoid unneccessary merge commits as they clutter up your history.

As we saw above, if you haven't pushed your branch to a stable remote branch, you can rewrite it:

- Use `git rebase` to rebase your feature branch against the branch you intend to merge into. This means that when you merge, it will be a so-called 'fast forward'

merge which avoids a merge commit.

- Use `git rebase -i` to rewrite your branch history, squashing related commits, rewording commit messages.

Build an audit trail

Try and build a good audit trail - your future self will be grateful. Where possible cross-reference other resources in your commit messages. These could be:

- Github pull requests or issues (eg "Related to #123")
- Mailing-list threads that discuss the work in question (try and use mailing list software that lets you link to a discussion). If you use Basecamp or something similar, link to the relevant discussion.
- Articles or blog posts relevant to your work

Basically anything that might be useful 12 months later when you're trying to work out the reasoning behind some component.

One thing, I try to do with django-oscar (a project of mine) is to maintain an audit trail from a commit all the way back to the mailing list discussion that instigated it. This works as follows:

- If you're puzzled by a particular line within a file, use `git blame` to find the commit that introduced it.
- The commit message should explain the change that led to this line and link back to a pull request.
- The pull request should be a set of related commits that together implement a new feature. The pull request description should be a functional spec for the feature in question, together with a link to the mailing list thread where this feature was discussed.

I haven't been following this process for long, but it seems to work well.

Use your prompt

Put relevant git information into your prompt - this will make your life easier. Here's a bash snippet for adding the current git branch to your prompt:

```
# ~/.bashrc
function parse_git_branch {
    git branch --no-color 2> /dev/null | sed -e '/^[^*]/d' -e 's/* \
(.*\)/(\1) /'
}
```

```
PS1="\[\e[32m\]\$(parse_git_branch)\[\e[34m\]\h:\W \$ \[\e[m\]"
export PS1
```

Use aliases for speed

Strive to be as productive as possible at the commandline. For me, that means minimising key strokes.

With git, you can define both git- and bash-aliases to make your life easier. I have lots (listed below). Two are worth highlighting:

1. I alias `git status -sb` to `g` so I can quickly check git status. This is my most frequently typed command so it makes sense to make it easy.
2. Like many others, I use a customised version of `git log` that lists one commit per line, but annotated with other useful information such as which commits other branches point to. See the definition of `git hist` below.

Selected aliases from `~/.gitconfig`:

```
[alias]
    hist = log --color --pretty=format:\"%C(yellow)%h%C(reset)
%s%C(bold red)%d%C(reset) %C(green)%ad%C(reset) %C(blue)
[%an]%C(reset)\" --relative-date --decorate
    unstage = reset HEAD --
    restore = checkout --
    cn = commit --no-verify
    co = checkout
    praise = blame
    visualise = !gitk
    graph = log --color --graph --pretty=format:\"%h | %ad | %an |
%s%d\" --date=short
```

And from `~/.bash_aliases`:

```
alias git='hub'
alias g='git status -sb'
alias gh='git hist'
alias gp='git pull'
alias gpr='git pull --rebase'
alias gpp='git pull --rebase && git push'
alias gf='git fetch'
alias gb='git branch'
alias ga='git add'
alias gc='git commit'
alias gca='git commit --amend'
```

```
alias gcv='git commit --no-verify'
alias gd='git diff --color-words'
alias gdc='git diff --cached -w'
alias gdw='git diff --no-ext-diff --word-diff'
alias gdv='git diff'
alias gl='git log --oneline --decorate'
alias gt='git tag'
alias grc='git rebase --continue'
alias grs='git rebase --skip'
alias gsl='git stash list'
alias gss='git stash save'
```

Something wrong? Suggest an improvement or add a comment (see article history)
Tagged with: git, commandlinefu
Filed in: tips

Previous: How to chroot a user in Ubuntu 12.04
Next: Mathematics and engineering

Copyright © 2005-2022 David Winterbottom
Content licensed under CC BY-NC-SA 4.0.