Scott Chacon

GitHub Flow

August 31, 2011

Issues with git-flow

I travel all over the place teaching Git to people and nearly every class and workshop I've done recently has asked me what I think about git-flow. I always answer that I think that it's great – it has taken a system (Git) that has a million possible workflows and documented a well tested, flexible workflow that works for lots of developers in a fairly straightforward manner. It has become something of a standard so that developers can move between projects or companies and be familiar with this standardized workflow.

However, it does have its issues. I have heard a number of opinions from people along the lines of not liking that new feature branches are started off of `develop` rather than `master`, or the way it handles hotfixes, but those are fairly minor.

One of the bigger issues for me is that it's more complicated than I think most developers and development teams actually require. It's complicated enough that a big helper script was developed to help enforce the flow. Though this is cool, the issue is that it cannot be enforced in a Git GUI, only on the command line, so the only people who have to learn the complex workflow really well, because they have to do all the steps manually, are the same people who aren't comfortable with the system enough to use it from the command line. This can be a huge problem.

Both of these issues can be solved easily just by having a much more simplified process. At GitHub, we do not use git-flow. We use, and always have used, a much simpler Git workflow.

Its simplicity gives it a number of advantages. One is that it's easy for people to understand, which means they can pick it up quickly and they rarely if ever mess it up or have to undo steps they did wrong. Another is that we don't need a wrapper script to help enforce it or follow it, so using GUIs and such are not a problem.

GitHub Flow

So, why don't we use git-flow at GitHub? Well, the main issue is that we deploy all the time. The git-flow process is designed largely around the "release". We don't really have "releases" because we deploy to production every day – often several times a day. We can do so through our chat room robot, which is the same place our CI results are displayed. We try to make the process of testing and shipping as simple as possible so that every employee feels comfortable doing it.

There are a number of advantages to deploying so regularly. If you deploy every few hours, it's almost impossible to introduce large numbers of big bugs. Little issues can be introduced, but then they can be fixed and redeployed very quickly. Normally you would have to do a 'hotfix' or something outside of the normal process, but it's simply part of our normal process – there is no difference in the GitHub flow between a hotfix and a very small feature.

Another advantage of deploying all the time is the ability to quickly address issues of all kinds. We can respond to security issues that are brought to our attention or implement small but interesting feature requests incredibly quickly, yet we can use the exact same process to address those changes as we do to handle normal or even large feature development. It's all the same process and it's all very simple.

How We Do It

So, what is GitHub Flow?

- Anything in the `master` branch is deployable
- To work on something new, create a descriptively named branch off of `master` (ie: `new-oauth2-scopes`)
- Commit to that branch locally and regularly push your work to the same named branch on the server
- When you need feedback or help, or you think the branch is ready for merging, open a pull request
- After someone else has reviewed and signed off on the feature, you can merge it into master
- Once it is merged and pushed to 'master', you can and *should* deploy immediately

That is the entire flow. It is very simple, very effective and works for fairly large teams – GitHub is 35 employees now, maybe 15-20 of whom work on the same project (github.com) at the same time. I think that most development teams – groups that work on the same logical code at the same time which could produce conflicts – are around this size or smaller. Especially those that are progressive enough to be doing rapid and consistent deployments.

So, let's look at each of these steps in turn.

#1 – anything in the master branch is deployable

This is basically the only hard *rule* of the system. There is only one branch that has any specific and consistent meaning and we named it `master`. To us, this means that it has been deployed or at the worst will be deployed within hours. It's incredibly rare that this gets rewound (the branch is moved back to an older commit to revert work) – if there is an issue, commits will be reverted or new commits will be introduced that fixes the issue, but the branch itself is almost never rolled back.

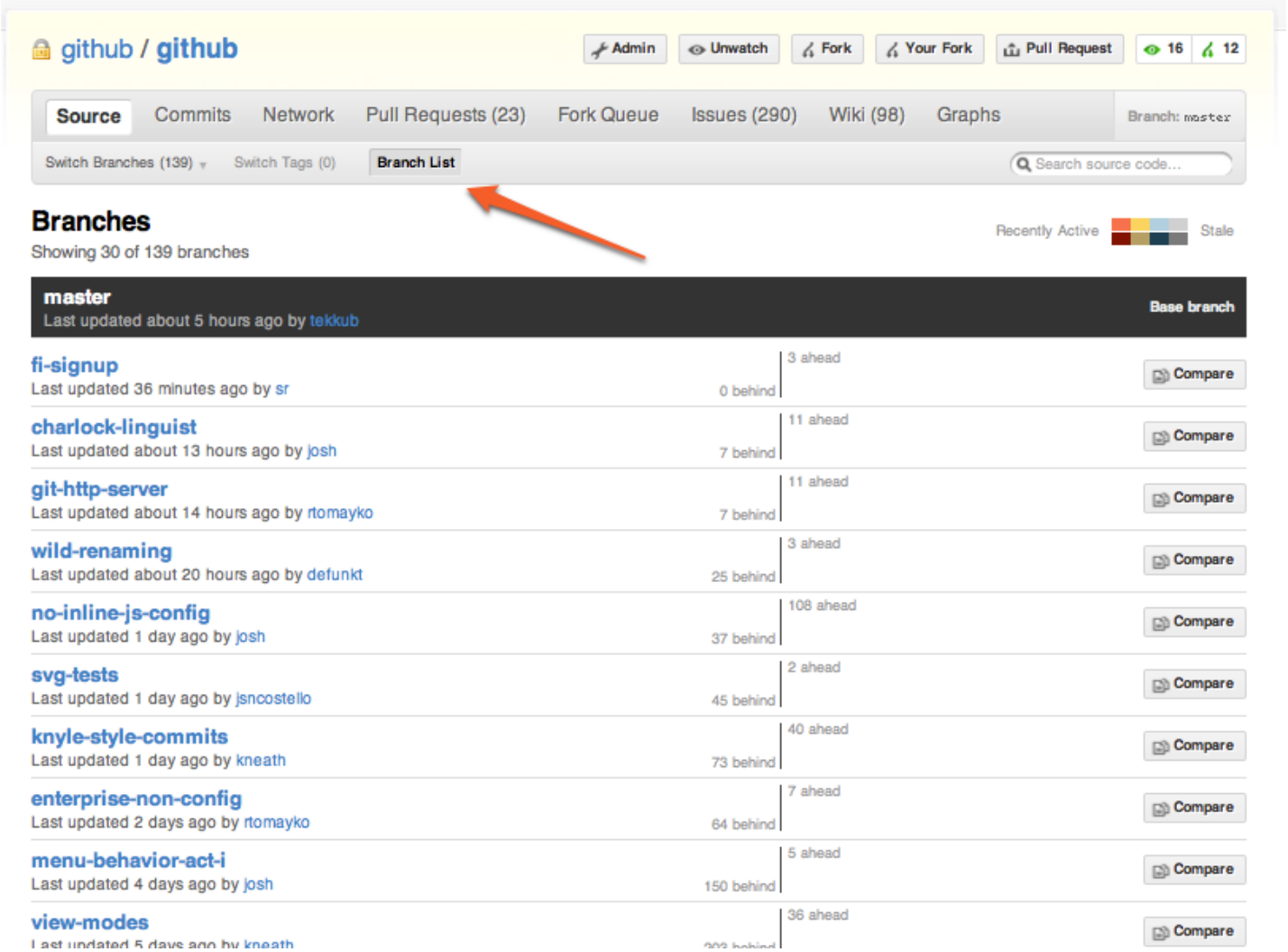
The `master` branch is stable and it is always, always safe to deploy from it or create new branches off of it. If you push something to master that is not tested or breaks the build, you break the social contract of the development team and you normally feel pretty bad about it. Every branch we push has tests run on it and reported into the chat room, so if you haven't run them locally, you can simply push to a topic branch (even a branch with a single commit) on the server and wait for Jenkins to tell you if it passes everything.

You could have a `deployed` branch that is updated only when you deploy, but we don't do that. We simply expose the currently deployed SHA through the webapp itself and `curl` it if we need a comparison made.

#2 – create descriptive branches off of master

When you want to start work on anything, you create a descriptively named branch off of the stable `master` branch. Some examples in the GitHub codebase right now would be `user-content-cache-key`, `submodules-init-task` or `redis2-transition`. This has several advantages – one is that when you fetch, you can see the topics that everyone else has been working on. Another is that if you abandon a branch for a while and go back to it later, it's fairly easy to remember what it was.

This is nice because when we go to the GitHub branch list page we can easily see what branches have been worked on recently and roughly how much work they have on them.



It's almost like a list of upcoming features with current rough status. This page is awesome if you're not using it – it only shows you branches that have unique work on them relative to your currently selected branch and it sorts them so that the ones most recently worked on are at the top. If I get really curious, I can click on the 'Compare' button to see what the actual unified diff and commit list is that is unique to that branch.

So, as of this writing, we have 44 branches in our repository with unmerged work in them, but I can also see that only about 9 or 10 of them have been pushed to in the last week.

#3 – push to named branches constantly

Another big difference from git-flow is that we push to named branches on the server constantly. Since the only thing we really have to worry about is `master` from a deployment standpoint, pushing to the server doesn't mess anyone up or confuse things – everything that is not `master` is simply something being worked on.

It also make sure that our work is always backed up in case of laptop loss or hard drive failure. More importantly, it puts everyone in constant communication. A simple 'git fetch' will basically give you a TODO list of what every is currently working on.

```
$ git fetch
remote: Counting objects: 3032, done.
remote: Compressing objects: 100% (947/947), done.
remote: Total 2672 (delta 1993), reused 2328 (delta 1689)
Receiving objects: 100% (2672/2672), 16.45 MiB | 1.04 MiB/s, done.
Resolving deltas: 100% (1993/1993), completed with 213 local objects.
From github.com:github/github
 * [new branch]      charlock-linguist -> origin/charlock-linguist
 * [new branch]      enterprise-non-config -> origin/enterprise-non-config
 * [new branch]      fi-signup  -> origin/fi-signup
   2647a42..4d6d2c2  git-http-server -> origin/git-http-server
 * [new branch]      knyle-style-commits -> origin/knyle-style-commits
   157d2b0..d33e00d  master     -> origin/master
 * [new branch]      menu-behavior-act-i -> origin/menu-behavior-act-i
   ea1c5e2..dfd315a  no-inline-js-config -> origin/no-inline-js-config
```

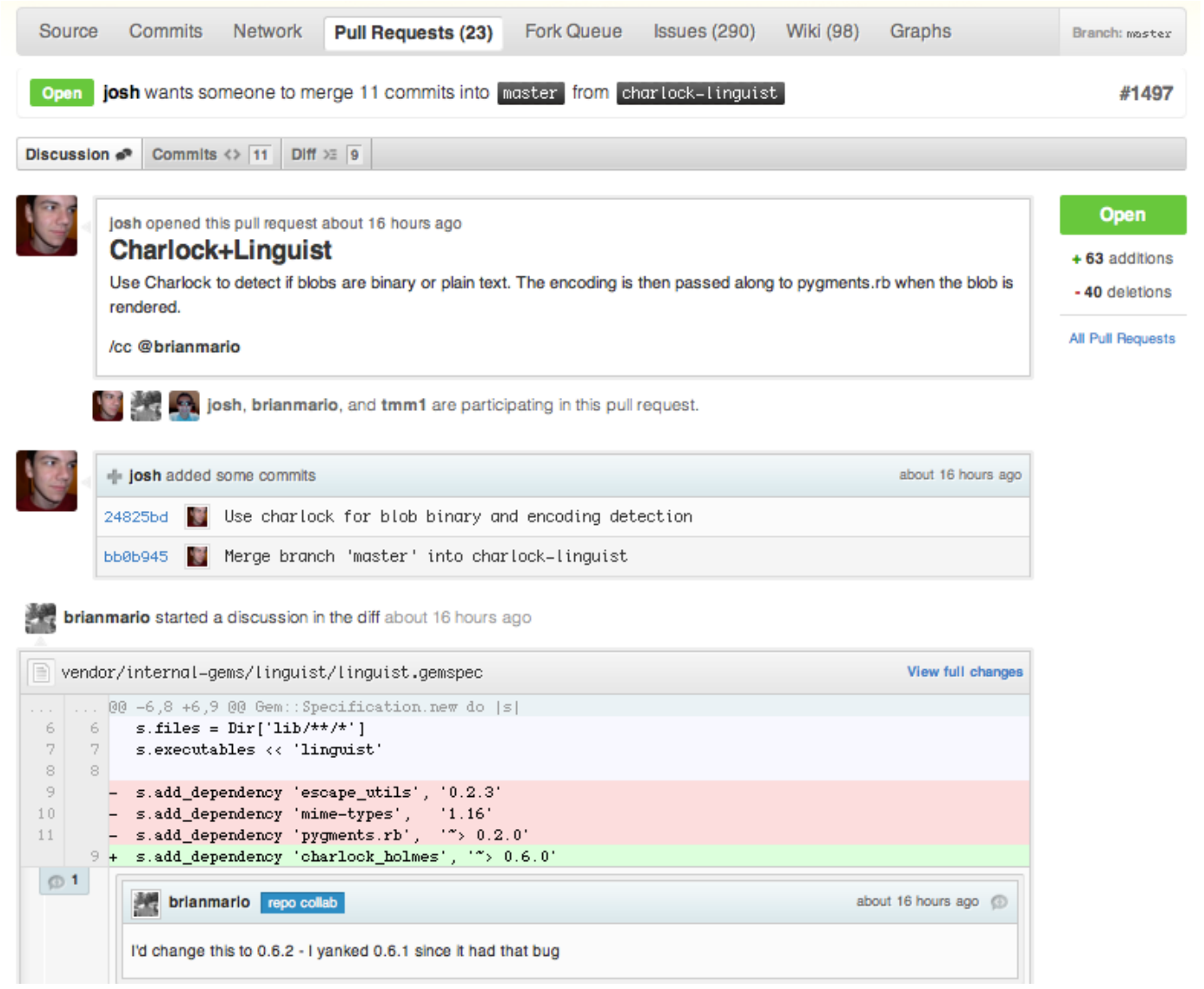
```
 * [new branch]      svg-tests  -> origin/svg-tests
   87bb870..9da23f3  view-modes -> origin/view-modes
 * [new branch]      wild-renaming -> origin/wild-renaming
```

It also lets everyone see, by looking at the GitHub Branch List page, what everyone else is working on so they can inspect them and see if they want to help with something.

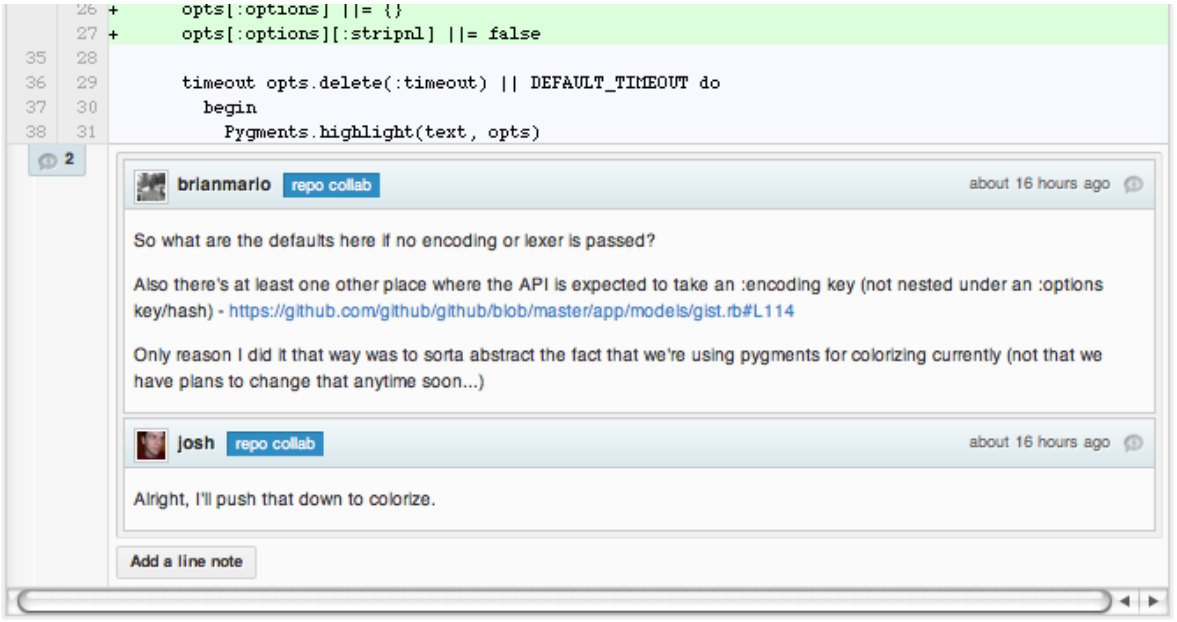
#4 – open a pull request at any time

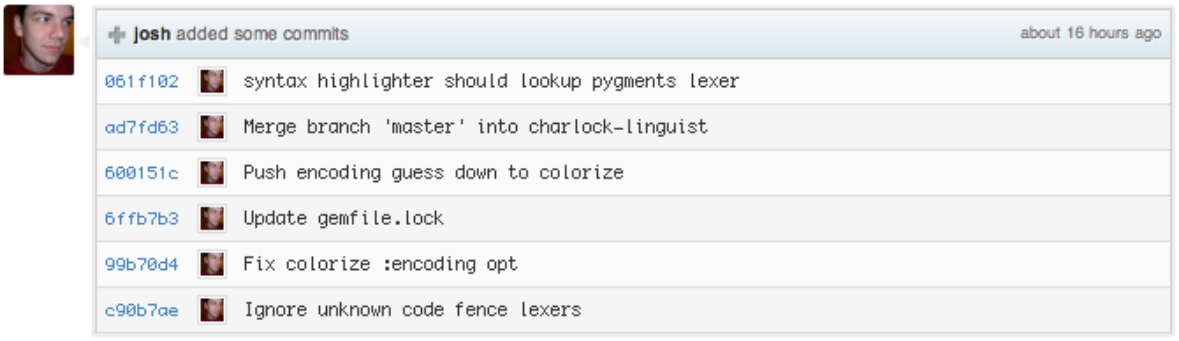
GitHub has an amazing code review system called Pull Requests that I fear not enough people know about. Many people use it for open source work – fork a project, update the project, send a pull request to the maintainer. However, it can also easily be used as an internal code review system, which is what we do.

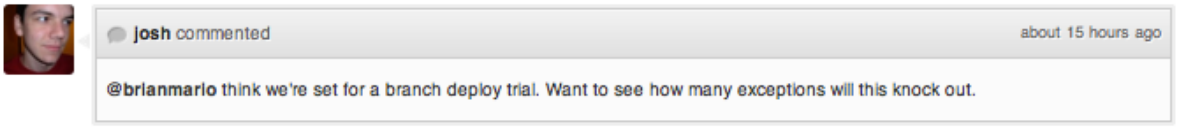
Actually, we use it more as a branch conversation view more than a pull request. You can send pull requests from one branch to another in a single project (public or private) in GitHub, so you can use them to say "I need help or review on this" in addition to "Please merge this in".



Here you can see Josh cc'ing Brian for review and Brian coming in with some advice on one of the lines of code. Further down we can see Josh acknowledging Brian's concerns and pushing more code to address them.





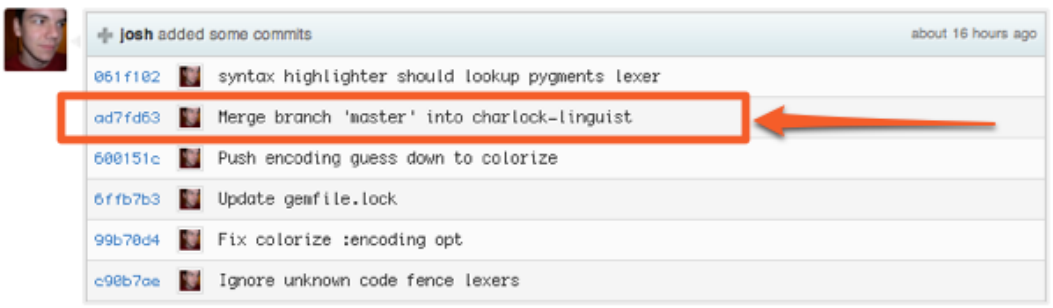


Finally you can see that we're still in the trial phase – this is not a deployment ready branch yet, we use the Pull Requests to review the code long before we actually want to merge it into `master` for deployment.

If you are stuck in the progress of your feature or branch and need help or advice, or if you are a developer and need a designer to review your work (or vice versa), or even if you have little or no code but some screenshot comps or general ideas, you open a pull request. You can cc people in the GitHub system by adding in a @username, so if you want the review or feedback of specific people, you simply cc them in the PR message (as you saw Josh do above).

This is cool because the Pull Request feature let's you comment on individual lines in the unified diff, on single commits or on the pull request itself and pulls everything inline to a single conversation view. It also let you continue to push to the branch, so if someone comments that you forgot to do something or there is a bug in the code, you can fix it and push to the branch, GitHub will show the new commits in the conversation view and you can keep iterating on a branch like that.

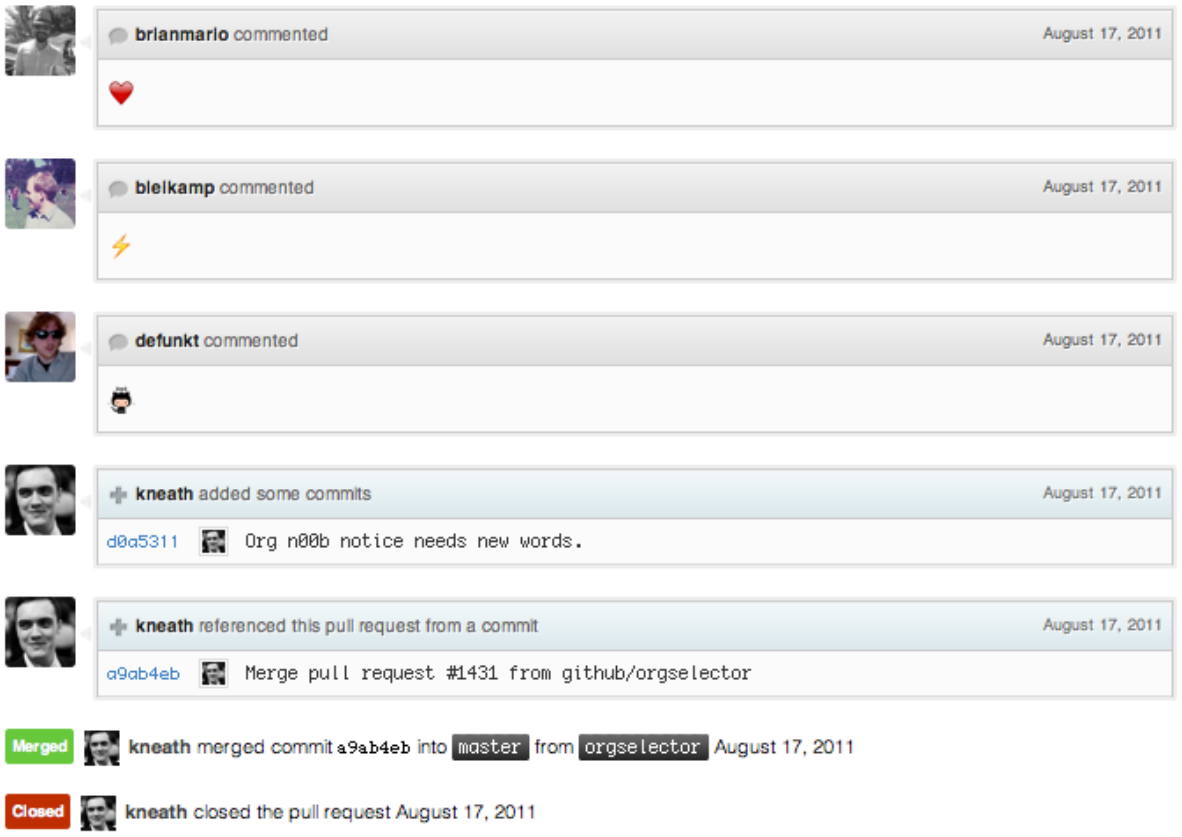
If the branch has been open for too long and you feel it's getting out of sync with the master branch, you can merge master into your topic branch and keep going. You can easily see in the pull request discussion or commit list when the branch was last brought up to date with the 'master'.



When everything is really and truly done on the branch and you feel it's ready to deploy, you can move on to the next step.

#5 – merge only after pull request review

We don't simply do work directly on `master` or work on a topic branch and merge it in when we think it's done – we try to get signoff from someone else in the company. This is generally a +1 or emoji or ":shipit:" comment, but we try to get someone else to look at it.



Once we get that, and the branch passes CI, we can merge it into master for deployment, which will automatically close the Pull Request when we push it.

#6 – deploy immediately after review

Finally, your work is done and merged into the `master` branch. This means that even if you don't deploy it now, people will base new work off of it and the next deploy, which will likely happen in a few hours, will push it out. So since you really don't want someone else to push something that you wrote that breaks things, people tend to make sure that it really is stable when it's merged and people also tend to push their own changes.

Our campfire bot, hubot, can do deployments for any of the employees, so a simple:

```
hubot depoy github to production
```

will deploy the code and zero-downtime restart all the necessary processes. You can see how common this is at GitHub:

Rick	hubot deploy github/oauth_cors to production	Tue, Aug 2
Rick	hubot lock deploy github migration db	Tue, Aug 2
Rick	hubot deploy github/oauth_cors to staging	Tue, Aug 2
tmm1	hubot deploy github to the cloud	Tue, Aug 2
tmm1	hubot deploy github to the cloud	Tue, Aug 2
atmos	hubot deploy logs for github	Tue, Aug 2
atmos	hubot deploy github to production	Tue, Aug 2
Rick	hubot deploy github to production/fs	Tue, Aug 2
Rick	hubot deploy github to production/arch1	Tue, Aug 2
Rick	hubot deploy github to production/fs	Tue, Aug 2
Rick	hubot deploy github to production/fs	Tue, Aug 2
sr	hubot deploy github to production/fe	Tue, Aug 2
sr	hubot deploy github to production	Tue, Aug 2
sr	hubot deploy github to production	Tue, Aug 2
sr	hubot deploy github/ghost-town to production	Tue, Aug 2
ekkub	hubot deploy github to production	Tue, Aug 2
sr	hubot deploy github/ghost-town to staging	Tue, Aug 2
sr	hubot deploy github to production	Tue, Aug 2
atmos	hubot deploy github to production	Tue, Aug 2
sr	hubot deploy github to production	Tue, Aug 2
atmos	hubot deploy github to production	Tue, Aug 2
atmos	hubot deploy github to production	Tue, Aug 2
atmos	hubot deploy github to production	Tue, Aug 2
meron	hubot deploy github to production	Tue, Aug 2
atmos	hubot deploy github to staging	Tue, Aug 2
atmos	hubot deploy github to production/smtp1	Tue, Aug 2
atmos	hubot deploy github to production/fe,fs,aux1	Tue, Aug 2
atmos	hubot deploy github to production/fe,fs,aux1	Tue, Aug 2
atmos	hubot deploy github to production	Tue, Aug 2

You can see 6 different people (including a support guy and a designer) deploying about 24 times in one day.

I have done this for branches with one commit containing a one line change. The process is simple, straightforward, scalable and powerful. You can do it with feature branches with 50 commits on them that took 2 weeks, or 1 commit that took 10 minutes. It is such a simple and frictionless process that you are not annoyed that you have to do it even for 1 commit, which means people rarely try to skip or bypass the process unless the change is so small or insignificant that it just doesn't matter.

This is an incredibly simple and powerful process. I think most people would agree that GitHub has a very stable platform, that issues are addressed quickly if they ever come up at all, and that new features are introduced at a rapid pace. There is no compromise of quality or stability so that we can get more speed or simplicity or less process.

Conclusion

Git itself is fairly complex to understand, making the workflow that you use with it more complex than necessary is simply adding more mental overhead to everybody's day. I would always advocate using the simplest possible system that will work for your team and doing so until it doesn't work anymore and then adding complexity only as absolutely needed.

For teams that have to do formal releases on a longer term interval (a few weeks to a few months between releases), and be able to do hot-fixes and maintenance branches and other things that arise from shipping so infrequently, git-flow makes sense and I would highly advocate it's use.

For teams that have set up a culture of shipping, who push to production every day, who are constantly testing and deploying, I would advocate picking something simpler like GitHub Flow.

Discussion, links, and tweets

I'm a developer at GitHub. Follow me on Twitter; you'll enjoy my tweets. I take care to carefully craft each one. Or at least aim to make you giggle. Or offended. One of those two— I haven't decided which yet.

Tweet Follow @chacon