Blog

Written words that are often helpful

How-to Git

Posted by chathaway on Aug. 15, 2016, 2:10 a.m.

This article gives a pretty simple workflow for Git that allows you to take advantage of the distributed nature, ease of branching, and awesomesauce that is Git; for now, I assume you use Github. This can easily be adapted to other Git-systems, such as Atlassian Stash or Gerrit.

This article is a little bit... sarcastic. It's intended for people who are good with the command line (or willing to learn); although pretty, the Github UI is somewhat limited. It is much better to become familiar with the CLI interface.

Step 0: Setup your SSH key

Duh. If you're not using Linux (*BSD is fine), install it. Using Windoze and/or Mac OS is a bad idea; politics and what not (really though, Linux is much better for most kinds of development for a lot of reasons). Once that's done, setup your SSH key if you don't have one

In all seriousness though, using a SSH key instead of HTTP authentication is more secure, easier, and most importantly allows you to enable two-factor authentication on your Github account without interfering with your ability to push changes.

```
charles@Bender:~$ ssh-keygenGenerating public/private rsa key pair.Enter file in which to save the key (/home/charles/.ssh/id_
```

(FYI, this isn't my real key!)

Export your key, and copy it to Github

```
charles@Bender:~$ cat ~/.ssh/id_rsa.pubssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQDXHTHDYnud7pSm1GYV2EShFQ1znWuJwO1dgbKmuFii1EaaOZK
```

Copy the parse that looks like "ssh-rsa ...."; the entire line. Just triple click on it, then right click and copy. Go here (https://github.com/settings/ssh) and click "New SSH Key". Paste it. Add. Done.

Bonus points:

Install meld. It is a nice diff/merge tool.

```
sudo apt-get install meld
```

Step 1: Get the source code!

If you're using Github, just grab the the Git line from the webpage. Click, copy.

Then go to your terminal, enter your work directory (I use ~/Programming)

```
charles@Bender:~/Programming$ cdcharles@Bender:~$ cd Programming/charles@Bender:~/Programming$ git clone git@github.com:CSnap/
```

Step 2: Do stuff

That's right. Stuff.

Step 2.5: Commit. And commit often

If you're from SVN/CVS; forget everything you've ever done. Commit whenever you want. Use temporary commit messages like "git commit -a -m 'temp'". Don't worry about being perfect. Make mistakes.

DO NOT FALL BACK TO OLD HABITS! Don't do "cp main.cpp main.cpp.bak" when you're about to make a huge change. Do "git commit -a -m 'temp'". Make that an alias for all I care.

Step 3: Get ready to push your changes

Awesome job! You've made the code that will p@wn the planet. You cured cancer. You constructed Jarvis.

Before you push your changes back to the server, UNDO ALL THOSE TEMP COMMITS! And write a real commit message. Like a real person.

```
git reset --soft origin/mastercharles@Bender:~/Programming/rpi_csdt_community$ vim README.mdcharles@Bender:~/Programming/rpi_c
```

Sweet. But you know what? Your pal, Jim. He made a commit behind your back. And you weren't ready. So now you need to merge your code! First, grab his code and see if Git can merge it automagically.

```
charles@Bender:~/Programming/rpi_csdt_community$ git fetchcharles@Bender:~/Programming/rpi_csdt_community$ git rebase origin/m
```

Trouble? Big error message? Lots of noise! S'all good! Just run the mergetool then continue the rebase!

```
charles@Bender:~/Programming/rpi_csdt_community$ git mergetool.. Output ...charles@Bender:~/Programming/rpi_csdt_community$ gi
```

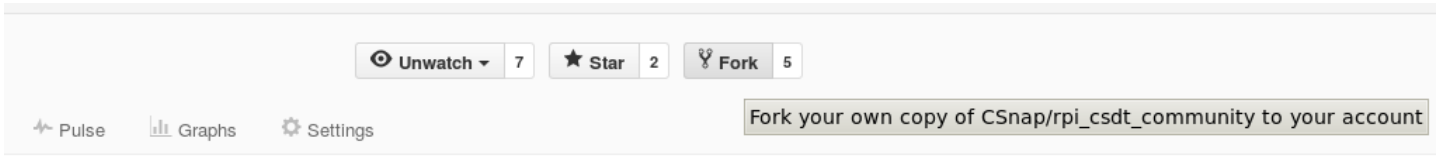
All done! Now you're ready to push your changes.

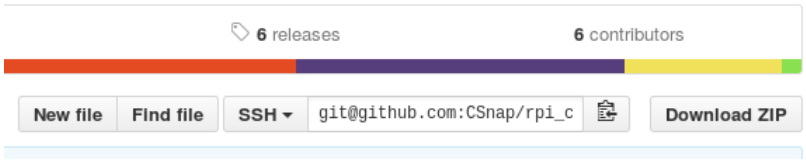
Some notes on branching patterns

I strongly encourage you to contribute to open source software; it is rewarding, and helps build better software for everyone. However, when you start contributing to real projects, people will want to look at the code you add before you add it (to make sure you're not the working for NSA, hacking our servers). To that extent, I recommend you do a pull request on Github instead of trying to directly push to the main repository.

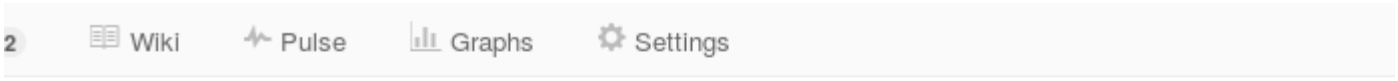
You should avoid pushing directly to your master branch; the command git push : allows you to put your commits in any branch you feel like. It doesn't matter where you do development locally (despite a bunch of online blogs that tell you otherwise), but try to keep your public-facing Git repository clean. As an added bonus, doing things this way will allow you to have multiple pull requests open at once; you can create a new pull request for each branch.

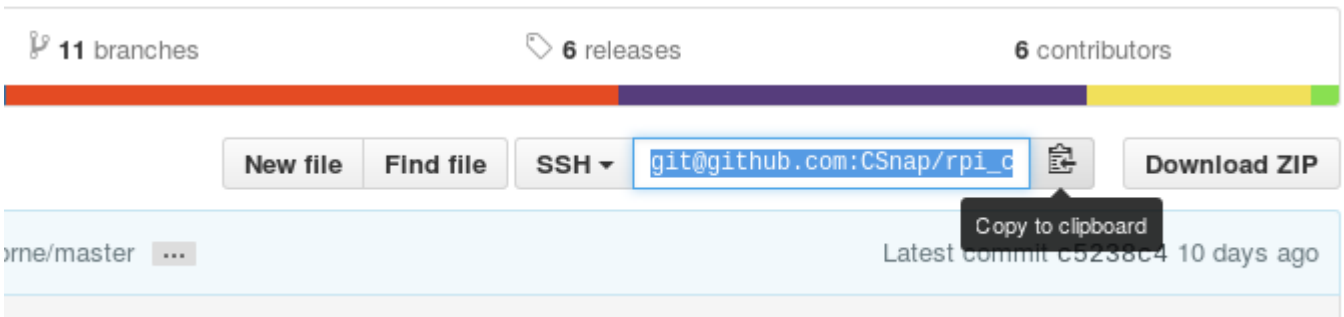
Step 1: Fork the project





Step 2: Add the forked project as a remote

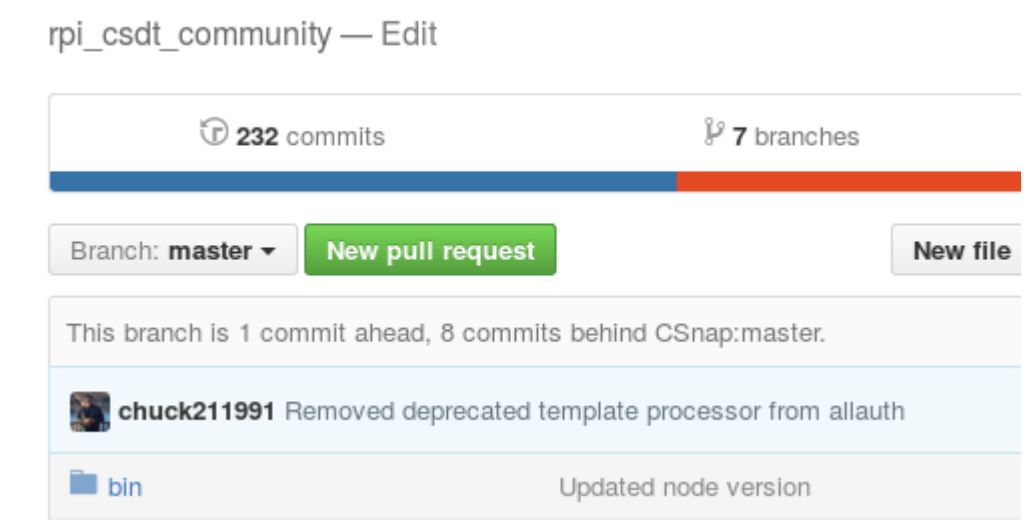


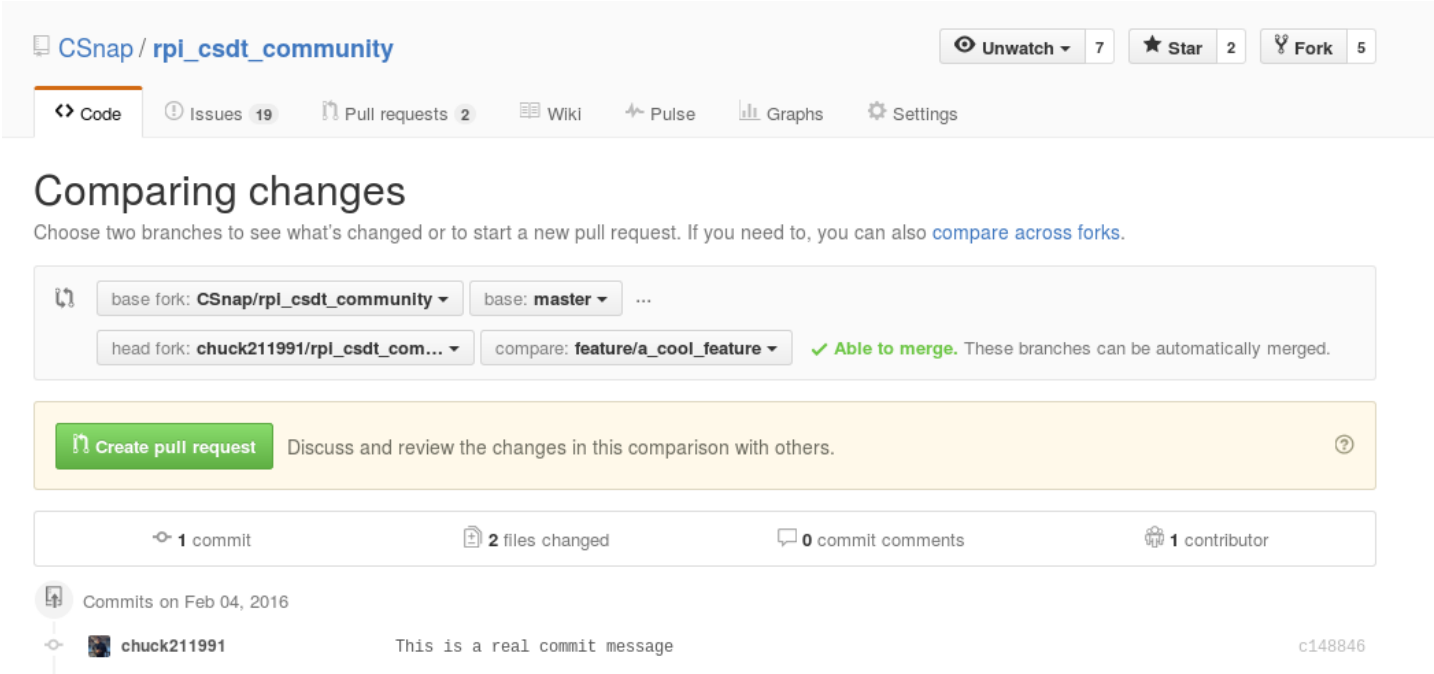



```
charles@Bender:~/Programming/rpi_csdt_community$ git remote add my_fork git@github.com:CSnap/rpi_csdt_community.gitcharles@Ben
```

Step 3: Upload your changes to a feature branch and create pull request

```
charles@Bender:~/Programming/rpi_csdt_community$ git push my_fork HEAD:feature/a_cool_featureCounting objects: 5, done.Delta c
```





Release Management

Alright, so you finished making some sick software. Now you want to give it out to people, and make an official version. Here's my advice on how this should be handled.

First, make a release branch. This branch is where you will modify anything that says "version 1.0" or things of that sort. Small changes like this won't need a review, but you may need to make some changes to get things ready for release (such as fix a few things on the punch list). You should do those changes as normal, and merge them into the release when they're committed on the main branch.

```
charles@Bender:~/Programming/rpi_csdt_community$ git checkout -b release/2016-02-09Switched to a new branch 'release/2016-02-0
```

If changes need to be done before release (such as an error found that prevented deploying), fix it your development branch then go through a shortened review process ("hotfix").

```
charles@Bender:~/Programming/rpi_csdt_community$ git checkout -b hotfix/fix_settings_file origin/masterBranch hotfix/fix_setti
```

When these changes are done, you will merge them into master following the normal process. You can then merge them into the release branch; but you can't do a rebase here. Ideally your release branch will already be on Github, and you should never rewrite history of a public repository. Instead, do a standard merge.

```
charles@Bender:~/Programming/rpi_csdt_community$ vim rpi_csdt_community/settings.py charles@Bender:~/Programming/rpi_csdt_comm
```

Why do the prepping on a separate branch?

Because merge commits are dumb and ugly. And it pollutes the history to have silly commits that just update the version; the version should always be set to "DEVELOPMENT", except when you are doing a release. YOU SHOULD DO A BUNCH OF TESTS BEFORE ACCEPTING A RELEASE.

On the topic of "code freezes"

There is no reason to freeze the code with Git. When you're getting ready to do a release, the team/person responsible for the release should create a new branch, while the development team continues working. This allows the release team to control the features that get included with a particular release, without shutting down an entire team for a day or two.

Generic Checklist

This is a generic code-review checklist. You should augment this for your specific project.

1. The commit represents a single feature-change; additional features should be made in separate commits
2. There is a commit message that makes sense and describes the change being implemented; the message should indicate which issue (Github issue, Jira issue) this commit addresses
3. The developer states (either in the commit or in a comment in the review) how the changes were verified (testing)
4. The code conforms to styling conventions for the project

This should be an absolute minimum; this doesn't even count as a code review. The reviewer should also check to make sure the change makes sense, test it themselves, identify potential issues, request explanation of non-trivial code, etc.
