A Lightweight, Resilient Git Workflow

May 26, 2015

When I first started working with Git, I tried using workflows from the web. I wanted a workflow that would provide feature development isolation, gracefully support continuous deployment, and have minimal overhead when performing common operations like branching and merging. I particularly wanted the merge workflow to be simple enough to minimize risk of the codebase changing significantly by the time the merge was finished (requiring yet another merge). For rapid new feature development, I wanted master to be kept clean so anyone could branch at any time and get a known good code state. Finally, I wanted an integrated code review step since code review is one of the best practices I know for maintaining code quality.

The workflows I found all lacked something. Some were overly complex. Others lacked sufficient feature development isolation. Others lacked a code review step. Starting with some of the simpler workflows, we tweaked and modified steps until we finally arrived at something we liked. In retrospect, I realize the workflow we arrived at is quite similar to Github's git workflow, with the exception that we prefer using forks rather than having everyone work from the same repository (for reasons stated later in this post).

In short, this is the guide I wish I had when I first started using Git.

As an aside, it bears mentioning that some projects, particularly large or complex ones, may demand a more complex workflow. I'll cover such cases later in this post. Additionally, the workflow described here requires a continuous deployment and testing environment to make it fully functional. I'll also discuss this issue later in this post.

The Workflow

High Level

The essence of the workflow is this:

Do this once:

1. fork and make a local clone

Every time you begin a new feature:

1. get the latest updates from upstream
2. make a feature branch
3. make changes, commit often, rebase often
4. submit a pull request
5. once approved, merge to upstream

In Detail

Setup

Do this once.

Make a fork of the master repository on github.com by finding the 'fork' button on the main repository page. On your machine, do a git clone of the new repository you just forked.

```
git clone https://github.com/github-username/project-name.git
```

Define the main repo as your upstream repository.

```
cd project-name
```

```
git remote add upstream https://github.com/tetherpad/project-name.git
```

Check that your setup so far looks correct.

```
git remote -v
```

The output should look something like this:

```
$ git remote -v
origin    https://github.com/git-username/project-name.git (fetch)
origin    https://github.com/git-username/project-name.git (push)
upstream    https://github.com/upstream-user/project-name.git (fetch)
upstream    https://github.com/upstream-user/project-name.git (push)
```

Develop

Everytime you start working on a new feature, do the following.

Get the latest changes from upstream.

```
git checkout master
git pull upstream master:master
```

Now, make a new branch for local development.

```
git checkout -b my-branch
```

Make local changes. Stage changes you want to commit.

```
git add relevant-file
```

Check that git status is as you expect.

```
git status
```

Verify the changes are what you think they are.

```
git diff --cached
```

Commit.

```
git commit -m 'This is an informative commit message that helps others navigate my code.
Either that, or I'll squash it later.'
```

Rinse, repeat.

Push

When your changes are ready for review...

Re-sync with upstream.

```
git checkout master
git pull upstream master:master
git checkout my-branch
git rebase master
```

(or if you want to rewrite history, git rebase -i master. More on this in a subsequent post.)

Push your changes to your fork on github.

```
git push origin my-branch
```

Now, on github, navigate to your fork's main page: https://github.com/github-username/project-name.git

You should see your new branch highlighted at the top with a "Compare & pull request" button. Press the button.

Review the diff - if it's not the diff you want code reviewed, go back to the *Develop* stage and make further changes until you have the changes you want.

Does it all look good? Yay! Now add a title and comment which will help reviewers know what your changes are all about. Include relevant notes about how you tested your changes, or how someone else can log into your test server to take a look. When all is dandy, finally, at last, click the "Create pull request" button.

The pull request will now show up in the upstream repository list of pull requests.

Code Review

Ask someone nicely to review your code. You can locate your pull request in the upstream repository on github.com under "Pull requests." Converse in-line and on the pull request as a whole about the code changes. If you need to update your pull request, go back to the *Develop* stage, and complete steps again from there up to this point.

Once a trusted reviewer or two have given the "thumbs-up!" on your changes, go to the pull request page on the upstream repository and click the "Merge pull request" button. If github.com tells you that you cannot merge your changes using the button, perform a local merge on your machine, and resolve any conflicts. See Github's guide on resolving merge conflicts.Then re-review, re-test, and merge, with another review pass if necessary.

To make the code review step as little of a break in development flow as possible, we like to have git publish pull request and code review notifications into team chat. This reduces the number of places a person needs to watch to make sure they're not missing relevant code handoffs.

More on code review in a subsequent post.

Discussion

Starting work on a new feature using this workflow is fast. Frequent rebasing resolves conflicts early in small bites rather than one big merge later. Frequent small merges make conflicts clearer and bugs easier to diagnose, while ensuring results of each merge are tested frequently as development continues.

I especially appreciate the clean distinctions of labor and ownership without extra process around the code review step. Once a pull request has been submitted, ownership is explicitly transferred, allowing the developer to clear her mind and move on to developing the next feature. Once code review is complete, steps to merge are kept to a minimum, meaning less disruption of other work.

Note that pull request size can have huge effects on code quality. Pull requests should be small enough to be reviewable in under 15, or ideally under 5, minutes. I hope to discuss this topic more in depth in a subsequent post.

Repos, branches & Names

To clear up potential confusion, 'master' is the name of the source-of-truth branch on all repositories in this approach. Your fork has a 'master' branch and the upstream repository has a 'master' branch as well. This is the branch from which new feature branches are created, and from which deployments are performed against the main repository.

The upstream repository itself could be referred to as the ‘main repository,’ the ‘upstream repository,’ or the ‘master repository.’ I’ve tried to use the terms ‘main repository’ and ‘upstream repository’ here to avoid confusion with the branch named ‘master.’

Test deployments of a forked repository should be performed against whatever branch contains the features you want to test. The only way new features should get into your own fork’s master branch is by pulling changes from upstream master. This happens only after they have been pull requested, code reviewed, and merged into upstream. A feature branch should never be merged directly into the master branch of your fork. If you do this, your master branch history will be out-of-sync with upstream’s master branch, and you will no longer be able to get changes cleanly from upstream. In this case, I normally delete the fork and re-fork.

Why Forks?

Forks keep repository branches clean. Any branches on the main repository have a clear and explicit purpose. Branches on a developer’s fork can be managed at the sole discretion of the fork owner. A developer can experiment or make mistakes to their heart’s content without worrying about affecting anyone else’s work.

If all branches live in the main repository, no one person has the authority or knowledge to remove obsolete branches. When branches are instead maintained on individual developer forks, each developer then has both the knowledge and authority to manage their branches. Furthermore, there are fewer branches on each fork, making the management task more tenable.

Finally, forks provide a better sandbox. You can hire a contractor and have them start developing in your code base with little risk. There is no need to give all developers write access to the main repository if it’s not appropriate to do so. Repository write permissions are de facto managed at a user level.

Notably while write access is inherently limited, read access is more difficult to limit. The Github support pages say that they will remove a person’s access to a private fork if requested to do so by the main repository’s owner. Of course, nothing eliminates the ability of a person to make a copy of a codebase and stash it somewhere if they really want to.

Quality of Upstream, and Implied Infrastructure

One of the results of this workflow should be that the quality of the upstream master branch is maintained. However, in order for this to actually be the case, you must integrate a testing step into the process. This testing step should occur before each pull request, and after each merge.

To make this possible, you must have a test environment which can be deployed against any fork or feature branch. This allows the developer to verify features before they submit a pull request. This could be a simple deployment of code to the developer's local machine, or a deployment to test servers dedicated for testing.

Since this workflow aims to keep the master branch on the main repository deployable at all times, another implied tool is a continuous deployment system, e.g. Travis CI or Jenkins, which performs deployments whenever code is committed to the main repository. Ideally, this deployment server should run a test suite as well, and publish test results to a shared chat medium.

I'll note that continuous deployment/testing environments are huge topics in and of themselves. I won't go into these in any depth in this post.

Other concerns

Larger Projects

Teams may desire to tweak elements of this process according to local custom, especially as the size of the development team grows. For instance, your project may have certain conventions it wants to follow regarding specifying code reviewers in the pull request message, or for squashing all pull requests into a single commit to keep git log clean on upstream.

As a diversion from the above process, when a project becomes sufficiently active and complex such that the QA process takes more time to complete than the average time it takes for the next developer to merge new changes to master, it may no longer be tenable to test and deploy from the master branch. QA is either asked to test a moving target (deployed code continually changing during testing), or developers are forced to hold off committing until QA is complete.

In this case, we would suggest a modification to this process - using a release branch to deploy releases to a staging server instead of deploying from master. Then this release branch is what ships when you deploy to production. When you think you're ready to cut a new release, you update the release branch from master, and begin QA on that branch. The specifics of this approach are beyond the scope of this document, however it is similar to the GitFlow style of development and release.

This of course only matters if you decide that your product needs to be bug-free. If bugs are okay in your product, then this issue may be moot.

should I Push to Master on my Fork?

This question sometimes arises, so I'll address it here.
After updating your local cloned repository from upstream, you can choose to push the latest updates to your fork's master branch on github. However this is not actually necessary. Because you are always updating your local checkout of the master branch directly from the upstream repository - never pulling from your own fork's master branch - the status of the master branch on your fork generally won't make any difference. The exceptions to this include when you are performing repeated clone operations, or when the fork's master diverges so far from master that you begin to notice performance issues.
If you decide you want your fork to be updated, you can update your fork's master branch like so:

```
git checkout master
git push origin master:master
```

Cheat Sheet

- Setup
    - make your own fork of the main repo on github
    - `git clone https://github.com/github-username/project-name.git`
    - `cd project-name`
    - `git remote add upstream https://github.com/tetherpad/project-name.git`
    - `git remote -v`
- Develop
    - `git checkout master`

    - `git pull upstream master:master`
    - `git checkout -b my-branch`
    - make local changes
    - `git add relevant-file`
    - `git status`
    - `git diff --cached`
    - `git commit -m "This is an informative commit message that helps others navigate my code. Either that, or I'll squash it later."`
    - rinse, repeat
- Push
    - `git checkout master`
    - `git pull upstream master:master`
    - `git checkout my-branch`
    - `git rebase master`
    - `git push origin my-branch`
    - review diff on github and submit
- Code Review
    - discuss pull request on github
    - merge if good, if not, go back to *Develop*

Resources

Some further resources on this topic:

- http://blog.endpoint.com/2014/05/git-workflows-that-work.html
- http://sethrobertson.github.io/GitBestPractices/
- http://nvie.com/posts/a-successful-git-branching-model/

In Closing

What do you think of this workflow? How do you do it on your team?