code.tutsplus.com

Focusing on a Team Workflow With Git

by Ian Lollar1 Dec 2014

8-10 minutes

Git provides numerous benefits for the solo developer, but it also really shines when it comes to team collaboration.

The key to establishing a great team workflow with Git is communication. Git is versatile, flexible, and can accommodate a variety of usage patterns. Deciding the workflow “rules of the road” ahead of time will help eliminate friction and confusion, and allow a team to take advantage of what Git does best: boost productivity.

That being said, this wouldn’t be much of a tutorial if it didn’t provide a tangible Git-based team workflow for you to review. The following example is based on a very popular Git workflow crafted by Vincent Driessen called Git-Flow, though it differs in certain key ways. There are several popular Git workflows floating around the

web—I'd suggest reading as many as you can so your team can pick a ruleset that feels best for it.

Let's kick off your research with the following workflow:

The Rule Above All Else

The `master` branch is always deployable. *Always.*

A deployable `master` branch is important for many reasons. First, it enables anyone new to a project to pull and build immediately without errors. Nothing is so frustrating as not being able to build an unfamiliar project.

Second, `master` shows the current state of production and/or the shipped product. If hotfixes need to be made, it is clear where to branch from.

Lastly, a deployable `master` is a safety net. If `master` is always deployable, then we can deploy without worry. Worry causes stress, and stress causes indigestion. Nobody needs that.

Branching Strategies

The `develop` branch should be the main branch of ongoing development. Feature branches are created from and merged back

into `develop`, and `develop` represents the bleeding edge of our codebase.

Since both `master` and `develop` are permanent and highly-trafficked branches, they should never be worked in directly. Instead, all work should be done in feature branches. When implementing a new feature, branch from `develop` and hack out the feature.

What's in a Name?

There are no hard-and-fast rules on branch naming, especially for feature branches. If a branch is a fix, it is probably best to prepend "fix-" to it. If a branch is a release, it is generally encouraged for the branch to follow this format: "release-X.X.X".

In general, branch names should be descriptive. And perhaps funny. The occasional and timely pun wouldn't go amiss.

You Say "Merge", I Say "Rebase"

Once your new awesome feature is coded, it's time to get it back into a shared branch (let's assume we're merging into `develop`). But before merging into `develop`, make sure that your feature

branch has the latest changes from `develop` because there may be conflicts.

All conflict resolution should happen in your feature branch. If you branched to make a small change/fix and *you have not pushed the branch to the remote*, rebase `develop` into your feature branch, and then merge your feature branch into `develop`. Push and then feel free to delete your local feature branch.

If you have pushed your branch to the remote, first merge `develop` into your branch (resolving conflicts), and then merge your branch into `develop`. Push and feel free to delete both the local and remote feature branch.

When rebasing, keep in mind that it is a *destructive* action. Meaning... be careful! Rebasing is really useful for cleaning up commit histories, but you don't want to rewrite history on anything that has been shared with someone else.

Here are a couple of rules to keep you safe when rebasing:

- Never rebase anything that has been pushed to the remote. Is the branch you're on *only* local? Then it is good to rebase. Otherwise, *no rebasing*.
- Rebase shared branches in local branches. `develop` is a shared

branch. `my-awesome-feature` is a local branch. Now I'm ready to merge `my-awesome-feature` into develop, but I want to make sure any changes that have happened in `develop` are merged into my feature branch first:

```
1 git checkout my-awesome-feature
2 git rebase develop
3 git checkout develop
4 git merge my-awesome-feature
5
6
7
```

Peer It Up

Say we've branched, we've coded, we've merged/rebased from `develop`, and now we're ready to merge our new code back into `develop`. But should we? Perhaps someone should review our changes first...

Code reviews are a good thing! They allow you to get valuable

feedback on the work you've done, and—if nothing else—increase the probability of mistakes being caught and fixed.

This is where Git's pull requests (and Bitbucket's interface) come in handy. (For a refresher on opening and managing pull requests in Bitbucket, check out part two of this series, Using Pull Requests as Code Reviews.) Pull requests can be much more than just reviewing changed code. Since pull requests are brand-based, they can become threads for discussing and collaborating on individual features. You can embed photos to share designs, comment directly on lines of code, and even use GIFs and emojis to have some fun.

When it comes to merging pull requests, it is preferable for the merge to be authored by the same person who opened the pull request, because that is likely the person who wrote the new code. To achieve this, reviewers should leave a comment approving the new code, but not actually hit the merge button. Once a teammate gives the code a "thumbs up" (either figuratively, or literally with a `:thumbsup:` emoji), the pull request opener can then go ahead and merge. Peer review plus clean logs—a wonderful thing!

I Like to Ship It, Ship It

Once `develop` is ready for a release, do a merge into `master`:

```
1 git checkout master
2 git merge --no-ff develop
3
```

Notice that `--no-ff` flag? That makes sure the merge won't be a fast-forward, so it will create a fresh commit. Why do we want that? So we can tag it! Tag that commit as the new version:

```
1 git tag -a vX.X.X -m 'Version X.X.X'
```

Then merge `master` back into `develop` so that `develop` has the version commit.

Speaking of versions, we should use semantic versioning. That breaks down to `MAJOR.MINOR.PATCH`. In general, `MAJOR` is a whole version number—it is used for massive changes and/or milestones. It is allowed to break backwards compatibility. `MINOR` is used for new features. It should not break any backwards compatibility. `PATCH` is for small changes and fixes, and should not break any backwards compatibility. We should be in a pre-release (`0.x.x`) until we launch.

Fix It Like It's Hot

We should never ship mistakes.

...but when we do, it's best to fix them fast. Since `develop` may include unfinished features, hotfixes should be branched from the current release—which is `master` (because `master` is always deployable!).

To make a hotfix, branch off `master`, make the fix, then do a *non-fast-forward* merge into `master`. Tag it, then merge `master` back into `develop` (because we'll want `develop` to have the fix too). Feel free to delete the hotfix branch.

It's Time for a Commitment

Let's talk about Git commit messages. Adhering to a common format will make it much easier to peruse our logs. Here are some good rules:

- Commit messages should be written in the present (imperative) tense: "Fix bug..." instead of "Fixed bug..." or "Fixes bug...".
- The first line (or subject line) should be a short summary of the commit (preferably 50 characters or less) with the first word

capitalized.

- If the summary line needs elaboration, you can, after a blank line, write a description. The description should be in paragraph form, with proper capitalization and punctuation.
- Commit messages should wrap at 72 columns so that logs in our terminals look pretty.

  If you want to read some more about proper Git commit message writing, take a look at Tim Pope's post.

Make It Your Own

Let me once again note that the workflow outlined above is meant to be a guideline, not hard-and-fast rules you should strictly implement for your teams. If you like them all, use ‘em! If something doesn’t work for you, tweak away!

What’s most important is that your team agrees upon a defined Git workflow, and then sticks to it. Once that happens, collaboration will follow, and you'll be taking advantage of the benefits Git has to offer when working as a team.

Check out some alternatives to the Git-Flow workflow in Atlassian's Git workflows guide.