How to Git as a Team

AUGUST 22, 2019 BY ROBIN WIERUCH - EDIT THIS POST

This tutorial is part 2 of 2 in this series.

Part 1: GitHub and Git Setup and Essentials

When I have been working with my clients over the last years, I have seen how crucial it can be to establish a **common sense git workflow for a team to become productive**. I experienced several constellations -- for instance when working in a team of thrown together developers, in an established team that just transitioned from another version control system, or as the new member of a team where no git practices were established and I wanted to get up to speed quickly -- where it made sense to align everyone on one git framework to follow a common sense and best practices.

After I have been through this struggle a couple of times, I wanted to write down what I have learned about git for teams which may help you to align your team on one workflow. If this blog post turns out too long for you, go through it with your team in a "lunch and learn"-break; and condense the most important points in a git workflow cheatsheet for your team. If you come up with the cheatsheet yourself after all, your team will own it and can iterate on it with the learnings that make sense for your particular case.

Note: Everything that follows conveys only my experience working with a team of 5 - 25 people with git as version control system. Nothing you will read here is set in stone, but I have seen productive teams once this workflow (or any other workflow) got established in an organization. If you follow a different workflow in your company, I would be curious to hear about it.

GIT TEAM WORKFLOW: BRANCHES

Basically there are three kinds of branches when working as a team in git:

master branch
staging branch
feature branch(es)

Whereas there can be more than one feature branch in your git workflow, there is only one master branch and one staging branch. The staging branch varies in its naming -- e.g. I have seen a staging branch being called *develop* and *development* branch as well. The master branch gets its name from git itself. The feature branches can be called whatever your team aligns on for the naming convention. I have seen something like:

feat/user-authentication
fix/landing-page-transition

Often people use namings like *feat/YMC-1634* for feature branches as well, to link them directly to a ticket in their scrum/kanban/... board. Note that feature branches are not only used for feature development, but also for bug fixes and other things. Feature branches are the place where all the implementation takes place, whereas staging and master branches are only used for releases of your application. In the following, I will use \<branch_name> for any of these branches.

GIT TEAM WORKFLOW: WHERE AM I?

There are several git GUI applications out there, which spare you using your command line. However, in the end I found it always the best to be familiar with the command line for git; to actually know which commands are used under the hood or to fix git problems that are nontransparent with the GUI tools. The most straightforward commands are:

```
git status
git log
```

Whereas the former command shows your changed files in staged and unstaged mode -- important: get familiar with these modes --, the latter command shows you the git history. Sometimes *git reflog* can save your ass if you screwed something up and you want to jump back in time. With the following git workflow, it's one of our goals to keep a well-arranged git history which can be seen with `git log`.

GIT TEAM WORKFLOW: BRANCH LIFECYCLES

The master and staging branches are only created once and stay as long as the project exists. In contrast, feature branches get created for the period of a feature development. They get merged into the staging branch and finally the staging branch gets merged into the master branch for a new release of your application. The staging branch in between is used for your CI/CD to prepare the next release, but also to see the staging version of your application online (e.g. staging.my-domain.com).

There are two essential commands to 1) create a new branch or to 2) check out an available branch:

```
git checkout -b <branch_name> // (1)
git checkout <branch_name> // (2)
```

At the beginning of your project's lifecycle, someone has to set up master and staging branch with the necessary configuration (e.g. no force push on master/staging branches, PR templates, ...). Also CI/CD needs to be set up especially for these two branches -- but also feature branches later on -- to find lint, test or formatting issues early on.

If you want to check out a feature branch which is only available in the remote repository, because a team mate has created and pushed it, but you don't have a local copy of it, call:

```
git fetch
```

Essentially a "fetch" keeps all the available branches in sync with your local machine in sync -- but not their commits, which needs another pull command. More about this later. If you want to delete a branch, you can either delete it 1) locally or 2) remotely:

```
git branch -d <branch_name> // (1)
git push origin -d <branch_name> // (2)
```

Be careful with the latter one, because most likely you would want to have the branch merged into staging before deleting it. Anyway, after you have finished and merged a feature branch, you are free to dispose it locally and remotely.

GIT TEAM WORKFLOW: FEATURE BRANCH

The most straightforward feature development with a feature branch looks like the following. First, you check out your new feature branch with `git checkout -b <branch_name>`. Next, you implement your code and use the following commands to make your changes available to everyone in the remote repository:

```
git add .
git commit -m "<commit_message>"
git push origin <branch_name>
```

Whereas `git add .` moves *all* changed/added/deleted files to staging for the next commit, you can use variations of git add to move only a subset of the changed files to staging. This is helpful if you follow an atomic commit strategy. For instance, I like to use `git add -u` to move all changed but not new files to staging.

If you are using `git status` in between, you will see that there are staged and unstaged files. Also git gives you instructions to move files from 1) staged to staging and from 2) staging to non changed:

```
git reset HEAD <file_path> // (1)
git checkout -- <file_path> // (2)
```

After moving your files into staging, every of these files gets committed with your commit message. There are different naming conventions for a commit message, it doesn't matter much which one you follow, but it matters that you align on one as a team. What I like to do is the following naming convention:

```
<type>(<which_file_or_domain>) <detailed_comment>
```

which can have the following types:

feat - actual feature implementation
style - code style and code clean up
test - actual test implementation
fix - bug fix
refactor - refactoring that doesn't affect the behavior of the code
chore - no production code changes, but more like configuration and setup

Thus, a commit message could look like the following:

feat(users) add authentication
fix(logout) clean up cookie
test(login) cookie set with access token
style(*) fix indentation
chore(.gitignore) add .env file

As mentioned, you don't need to follow this naming convention, but to keep everyone in your team on the same page, align on one naming convention yourself. This applies to commit messages more importantly than branch namings.

Last but not least, after you have added and committed your changes, push everything up to your remote repository with `git push origin <branch_name>`. This step is optional, because you can first accumulated commits before pushing all changes up to make it available to your team.

HOW TO KEEP A BRANCH UP-TO-DATE?

Regardless of the branch (staging/feature) you are working on, sometimes you need to update your local version of this branch with the changes from the remote branch, because someone else pushed updates to it. Before you start to update the branch, follow these optional steps:

If the branch isn't available locally for you, because someone else started it, you start with a `git fetch`. Next you navigate to the branch with `git checkout <branch_name>`.
If you have changed files, `git commit` or `git stash` them. The latter is used for storing your changes to apply them in a later stage again.

Then you can start to pull the latest changes. My recommendation would be to always use a rebase which puts your commits on top of the remote branch's commits:

```
git pull --rebase origin <branch_name>
```

If you have changed a file that has been changes remotely too, it can happen that you run into a merge conflict during the rebase. If this happens, resolve the file's conflicts and continue on the command line the following way:

```
git add .
git rebase --continue
```

In the worst case, it can happen that you run for every of your own commits that it is rebased on top of the remote branch's commits into a conflict. If that's the case, repeat the steps from before. If your pull rebase goes wrong, you can always abort it: `git rebase --abort`.

After the pull rebase finishes, all your commits should be listed on top of the remote branch's commits. If you have stashed changes away before, you can apply them again with: `git stash apply`. Your branch should be up to date with the remote branch's changes and your own changes on top.

HOW TO KEEP A FEATURE BRANCH UP-TO-DATE WITH STAGING?

Once you started to work on a feature branch, you may want to keep the branch up to date with the staging branch in case anyone else merged their feature branches into staging. So when do you want to keep your feature branch up to date with staging?

If you want to create a pull request (PR) of your feature branch to merge it into staging, but all the recent changes from staging should be included to reflect the latest changes but also to not run into merge conflicts.

If you need to include an update from staging (e.g. hotfix, library upgrade, dependent feature from someone else) to continue working on your feature branch without blocking issues.

Follow this git workflow to keep your feature branch up to date with the staging branch:

```
git checkout <branch_name>
// checks out your feature branch

follow: How to keep a branch up-to-date?

git checkout staging
// checks out staging branch

follow: How to keep a branch up-to-date?

git checkout <branch_name>
// checks out your feature branch

git rebase staging
// merges all your changes from your feature branch on top of the staging b

git push origin <branch_name>
// pushes the updated feature branch to your remote repository
```

If you have pushed your branch before to your remote repository, you have to force push your changes, because the history of your branch changed during the rebase with the staging branch:

```
git push -f origin <branch_name>
```

But be careful with a force push: If someone else made changes in between on this branch, a force push will forcefully overwrite all these changes.

HOW TO GET A FEATURE BRANCH READY FOR MERGE?

A pull request (PR) will merge all your feature branch's changes into the staging branch. Afterward, you can delete your feature branch locally and remotely. Even though a merge would be possible without a PR, a PR enables other people to review your feature on platforms like GitHub or Gitlab. That's why a best practice would be to open a PR for your feature branch once you finished the implementation of the actual feature on this branch and pushed everything to your remote repository. The git workflow looks as follows:

follow: How to keep a feature branch up-to-date with staging?
do: Open Pull Request on GitHub/Gitlab/... for your feature branch.
wait: CI/CD, discussion/review, approval of team members.
optional: Push more commits to your remote branch if changes are needed due to discussion.

If everything is fine with your feature branch, continue with one of the following ways:

1) Merge your PR directly on your GitHub/Gitlab/...
2) Or continue on the command line with the merge:

```
git checkout staging

git merge --no-ff <branch_name>
// merges your feature branch into staging

git push origin staging
```

Congratulations, you have merged your feature branch into staging and kept everything up-to-date along the way. Now, rinse and repeat this git workflow for your next feature branch. If you screwed up anything during your git workflow, it's always worth to check out this tutorial to revert almost anything with git.

BONUS: KEEPING YOUR GIT HISTORY TIDY

The previous git workflows for keeping your branches up to date and merging them into each are based on git's rebase feature. By using a rebase, you will always apply your changes on top of other team members changes. This way, you git history will always tell a linear story. The following commands will help you to keep the git history of your feature branch tidy:

```
git revert <commit_sha>
// if you want to undo a commit
// but want to keep this in history for documentation
// makes most sense if you follow atomic commits

git rebase -i HEAD~<number_of_commits>
// if you want to reorder, rename, or squash commits into each other

git commit --amend
// if you want to append changes to your last commit

git commit --amend -m "<commit_message>"
// if you want to change the commit message of your last commit
```

Some of these git commits will change your local commit history. If you have pushed your feature branch to your remote repository before, you will have to force push these changes. Be careful again that you don't overwrite a team member's changes with the force push.

I would be curious about other git workflows, if there exist any, so let me know about them in the comments. If you have any git best practices or gotchas you want to add, then let us know about them. After all, I hope this walkthrough helps you to establish a git workflow for your team working with git. In the long run, it will make you and your team more productive by aligning you on a common sense process.