How I Use Git

When it comes to talking about git best practices, a lot of people seem to point toward the git-flow branching model. Personally, there are things I like about it (hotfix branches) and things I can't get past (a preference for merge commits..ewww). I think with some tweaks, it could be a reasonable model for medium to large development teams. But I tend to work with small teams and I don't have actual experience with git-flow, so I'm going to stop talking about that.

Here's what I do have experience with: *How I Use Git*

Contents:

Seven Rules I (Almost Always) Follow

1. **Master is Sacred.** The `master` branch is like subversion's `trunk`. Successful projects tend to keep these in a state where releases can be made with minimal effort. Therefore, only code that has been fully tested is allowed in.
2. **Code in Branches.** See #1. Branches are extremely easy to create in git, and they don't go to the origin repository unless you explicitly push them . Use a relevant issue id as the branch name whenever possible. Also, convention in git-land seems to be that tags and branches are named all-lowercase, so if I have an issue named `MYPROJ-42`, name the branch `myproj-42`.
3. **No Merge Commits.** A merge commit is technically a commit with two "parent" commits in git. The problem is, if you're not careful, you'll create a lot of these unintentionally just because of the nature of distributed version control. Merge commits may reflect "the truth" about what happened in the repository, but they can be extremely difficult for people to understand. Linear commit histories are much less likely to make your fellow committers grumble when trying to review your changes. See How to Avoid Merge Commits below.
4. **Squash Commits and Use Good Commit Messages in Master** Commit as many times and with whatever level of log message detail you want while your branch is in development. But when merging it to master, keep in mind that the granularity of the commits and the content of the messages need to be useful to people besides you. See How to Squash and Re-word Commits and How to Write a Good Commit Message below.
5. **No Old Branches.** When you're done with a branch, whether it's been merged to master or not, delete it. It's just clutter. The history of the master branch is what matters. To remove a local branch, `git branch -d branchname`. To remove a remote branch, `git push origin :branchname`.
6. **Consistent Release Tags.** In the git world, people tend to name release tags like "`v1.0.0`". I like this convention (including the trailing zero) because the Major.Feature.Bugfix convention is compatible with semantic versioning and is commonly seen with Maven artifacts.
7. **Never Rebase Public Commits or Force a Push.** Rebase (described in the sections below) is a useful tool for doing surgery on private branches, but it has potential to cause serious problems if you use it with commits that have made it outside your local repository. Therefore:
    - If people *pull* a branch from you regularly, you should consider all commits on that branch to be immediately public and never rebase on it at all.
    - If you attempt to *push* history-modifying changes to a remote branch, git will notify you of the conflict and tell you how to force it. Don't force it.

How to Avoid Merge Commits

Merge commits are ugly. Nobody likes them and they smell funny. Here are some tips for avoiding them:

- When you do a `git pull` to get remote changes, if you have any local commits that haven't been pushed yet (*wait, you're not working directly on master, are you?*), specify `--rebase` (you can also make this the default option as explained in this article). This will modify the sequence of commits to your local repository so that your locally-committed changes appear *after* those you just pulled. Now when you push your changes, your fellow committers will just see an incremental set of changes with yours at the end, and it will make sense to them.
- Always *fast-forward* your branch commits to master. Fast-forward just means putting your commits at the end of the latest existing commits on a branch. When you run `git merge yourbranch`, by default, git will attempt to do a fast-forward commit if possible. It's not possible if there are any changes in master that aren't already in your branch. In this case, before attempting to merge your changes into master, switch to your branch and type `git rebase master`. This will put all the new commits from master in your branch, *behind* your changes. Now it should be possible to fast-forward merge your branch into master by switching to master and typing `git merge yourbranch`
- Note: It's possible, though not common, that a `git pull --rebase` or a `git rebase master` will fail due to a conflict. In that case, see this excellent article, which covers resolving rebase conflicts.

How to Squash and Re-word Commits

Before pushing the changes from an issue branch to the origin master branch, it's useful to be able to do a little surgery on them so they make more sense to others. Squashing commits allows you to turn a high number of commits into a low number, with the same resulting code. The sequence of commits get "squashed" into a smaller number of commits. Re-wording lets you completely change the commit message for any number of commits.

You can use an interactive rebase to do one or both of these. Let's say you're on a branch and you want to squash the last 3 commits into one and reword the commit message. First, run:

```
git rebase -i HEAD~3
```

The editor will come up and basically guide you through the rest of the process. In this case, you'll want to change the first line to begin with "r" (reword) and the following two lines to begin with "s" (squash into previous commit).

How to Write a Good Commit Message

If there's a tracked issue associated with your commit (ideal):

```
MYPROJ-42: Feature/bug description under 50 chars if possible

Longer description, if needed. Don't exceed 72 lines here. A blank line
between sections is ok if multiple sections of text are needed.

https://example.org/path/to/tracker/MYPROJ-42
```

Otherwise (it's a small change/typo fix, etc.):

```
Fix testing errors introduced by last commit
```

Notice the following conventions:

- The first line does not end with a period
- The first line is in the present tense
- If it's more than a one-line commit message, the second line is *always* blank

Git Tools I Can't Live Without

After using git for a little over a year, here are a few (Unix-based) tools I've settled on.

- vim fugitive - Shows your current branch in the vim statusline and lets you do a bunch of other git-related things from directly within the editor.
- git bash completion - Lets you use tab completion with a lot of git command-line things and lets you configure your prompt to automatically show the current branch when inside a git working directory.
- tig - A fast, text-based, clean looking history/diff browser.