Git Training Plan

A mostly painless introduction to Git and Git Flow

Opin Software

June, 2015

Andrew Trebble
Senior Developer
andrew.trebble@opin.ca

Table of Contents

Git Introduction

> "Git is a distributed revision control system with an emphasis on speed, data integrity, and support for distributed, non-linear workflows. Git was initially designed and developed by Linus Torvalds for Linux kernel development in 2005, and has since become the most widely adopted version control system for software development." - Git (Software) In *Wikipedia*. Retrieved June 23, 2015, from https://en.wikipedia.org/wiki/Git_(software)

Every copy of a Git repository is as fully featured and capable of performing operations as every other copy of the code. Each repo can operate completely independently of any other. Code operations (commits, diffs, branching, tagging, etc.) can be performed without any outside connections. Specific clones are given special roles only through convention and agreement of the development team.

This model allows for cheap (easy) branching and merges; and flexible team structures.

Basic repo management

Init

A git repo can be created at any place on a computer through a simple command:

```
$ git init my_repo
```

This will create a new directory called my_repo that has a full (but empty) git repository. From this point on any work done in the my_repo directory can be managed using git.

Bare Repos

A bare repo is a special type of Git repo. It is a repo that has no working copy. A working copy is a copy of all the files for a developer to work on. A bare repo only has the git files for managing file history and such. It is intended for copies of the repo that will not be worked on directly instead will be used only on a server for a purpose such as pushing deployments or as a central point for a team to integrate their work / interface with other teams. A bare repo can be inited using:

```
$ git init --bare my_repo
```

Cloning

Cloning a repo is the act of creating a copy of a repo from one place to another. The repo that was cloned from is then set as the "origin" for the new clone. A clone is an exact copy of the repo that was cloned from with the exception of the remotes.

A remote is another copy of the Git repo. It could be on a server through SSH, another team member's computer, a website, or even another directory in on the same computer. A repo can be cloned using one of the following sample commands.

To clone via SSH

```
$ git clone ssh://username@remote.server.com:/path/to/repo
  my_repo
```

To clone from a website:

```
$ git clone http://remote.website.com/path/to/repo my_repo
```

To clone from another directory on the same computer:

```
$ git clone /path/to/repo my_repo
```

Remotes

Git being distributed means that you are not limited to working with a single master server that manages the state of the code. Instead you can add any number of other clones of the repo to your own copy so that you can sync your work with other developers or teams. A remote can be added to your repo using the following command:

```
$ git remote add [address to the remote repo] server_name
```

The remote address is the same as can be used for git clone. Once a remote is added then you can pull from and push to the remote repo.

By default git creates a remote called origin when you clone a repo. If you created a new repo with git init then there is no default configured remote.

Ignoring

Before starting work on a repo you should first make a decision about what files you do not want included in the repo. Likely candidates are any metadata files that your OS or IDE creates (.DS_Store, Thumbs.db), configuration files that contain sensitive things such as passwords, encryption keys, large media files that are not required for the functioning of the code and potentially stored elsewhere.

To mark a file as being ignored you will add it to the .gitignore file in the root directory of the repo. If the file does not exist you can create it. The format for the file is as such:

```
# Ignore configuration files that may contain sensitive information.
sites/*/*settings*.php
# Ignore paths that contain generated content.
files/
sites/*/files
sites/*/private
# Metadata files
.DS_Store
```

Lines that begin with a # are comments.

If a file has already been added to the repo but then later added to the .gitignore then the file will not be ignored. It will be tracked as a normal file.

Commits and the stage

Before a file can be committed to the repo it needs to be added to the stage. The stage can be thought of as a prepping area for the commit that is about to be made. A commit will only take files that are on the stage when it is made. Files that have changed but are not on the stage will not be included in the commit. Individual files can be added to the stage or whole directory trees. Adding a directory will add any files under that directory that have been changed. Adding to the stage is also used when you are adding a new file to the repo. An important note here is that Git only cares about files it does not track directories. An empty directory cannot be added to the stage and thus cannot be committed.

To see what files have been changed and can be added to the stage:

```
$ git status
```

To add a file to the stage:

```
$ git add path/to/file
```

To add a directory sub tree to the stage:

```
$ git add path/to/directory/
```

After all the files that will make up a commit are added to the stage then a commit can be made. A commit will take all the files that are in the stage and bundle them up as a commit. Once the commit is completed and saved the stage is cleared. To create a commit:

```
$ git commit -m "A useful commit message"
```

If the commit message is to complex or long to add to the commit command then the -m option can be left out and git will launch a text editor for you to enter your message. Usually this will be vi or nano. Using these editors is outside the scope of this document.

If you have a lot of changes or you know for certain that all files that have been changed you can skip the step of adding files to the stage and add the -a flag to the commit command:

```
$ git commit -a -m "A useful commit message"
```

This will not add any new / untracked files to the commit.

Commits are always made to the current branch.

Log

The log will tell you the history of the current branch.

```
$ git log
```

You can quit from the log listing by pressing q. The log is the reason you want to provide useful commit messages. The messages will be listed here.

Branches

Creating

A branch is technically just a list of commits that when replayed from the begining will result in the working copy being put in a certain state. There are two common ways of creating branches.

```
$ git branch my_new_branch
```

This will create a new branch using the current commit as its starting point. This command will not switch the current working branch to the new one.

```
$ git checkout -b my_newer_branch
```

This will create a new branch using the current commit as its starting point. This command will then checkout the new branch for you to use.

Checkouts

A checkout is when you change the current working tree to match the state from a particular branch, commit, or tag. Checking out a branch will set the working copy to the state at the last commit on that branch. Checking out a tag or commit will set the working copy to the state of the specified commit. Checkout can be done to match the state of a remote repo's branch if you have already pulled that remote's refs to your repo. This is useful if you are want to test or integrate another team member's work.

To checkout a branch or tag:

```
$ git checkout master
```

To checkout a commit, first you need to find the id for the commit you want to checkout. This can be found in the log.

```
$ git log
$ git checkout aa8858d1a519fa459b484b803cbe49629aa38b32
```

To checkout a remote branch:

```
$ git checkout remote_name/branch_name
```

Sometimes you may receive an error message stating that you cannot do the checkout because it would overwrite changes in your working copy. This occurs when you have modified a file but not committed it yet and the checkout will result in your changes being overwritten. There are several ways to resolve this situation: committing the changes, stashing the changes, or reverting the changes. These options will be discussed later in this document.

HEAD

You will see the term HEAD quite often when reading about Git. HEAD is a special pointer that points to the last commit in the current branch. If you checkout a tag or a commit you will eand up in a "headless state". This means that git does not know what branch you are on and thus if you were to make a commit git would not be able to put that commit into a branch. From a headless state however you can create a new branch that uses the current commit as it's starting point.

Pulling (fetch and merge)

Pulling is the act of bringing the latest changes available on a remote to your local copy of the repo. A pull is actually two operations: fetch and merge.

A fetch is what git does to bring all the latest changes down from the remote. It does not make any changes to any of your local branches.

A merge is when you meld two or more branches together (legend has it that Linus once merged 12 branches together in one operation).

So the following two code blocks are equivalent.

```
$ git fetch origin
$ git merge origin/master
```

```
$ git pull origin master
```

You may be asking why would you need the first method. The answer is that you don't always want to do both operations at the same time. Some times you may only want to see what someone else has done. In that case you can do something like:

```
$ git fetch team_server
$ git checkout team_server/awesome_new_feature
```

Or you may want to merge two local branches

```
$ git checkout master
$ git merge my_awesome_feature
```

Remote branches can be merged as well

```
$ git checkout my_new_feature
$ git merge team_server/bills_latest_branch
```

Pushing

Once you have some commits ready and want to make your changes available for others to use you can push your work to a remote. You can push one branch or tag at a time. Before pushing to the remote you should make sure that your branch is up to date with the version on the remote by pulling the latest version on the remote. This is usually done with a pull.

```
$ git pull origin awesome_new_feature
$ git push origin awesome_new_feature
```

Upstream

A branch needs to be configured to know what branch on the remote is the same as itself. This is called the upstream tracking branch. This is useful for when you want to make frequent pushes to a remote for a branch. The upstream can be set when you do do a push to the remote.

```
$ git push origin my_new_branch --set-upstream
```

Or it can be set without a push

```
$ git branch --set-upstream my_new_branch origin/my_new_branch
```

Once the upstream is set for the branch then when you have the branch checked out you can do pushes and pulls without having to specify the remote and branch name.

Merging

Merging is the act of bringing two branches together into one. Usually you merge a feature branch into a mainline branch. Always remember that you want to be in the branch that is receiving the merge before you issue the command.

Assuming all your work has been committed to your feature branch, a merge can be initiated using the command:

```
$ git checkout my_mainline_branch
$ git merge my_feature_branch
```

Git was designed to make merging easy but that doesn't mean that there will not be problems. Which brings us to the next topic.

Conflicts

A merge conflict arises when the two branches made changes to the same file in such a way that Git

can't figure out how to bring the two together safely. In this case Git plays it safe and halts the process. The repo is left in a state where it needs user intervention to resolve issues and tell git how to continue.

Mergetool

There are several graphical diff tools that git can interface with to help you resolve conflicts. An excellent and free tool for resolving merge conflicts is kdiff3. Documenting how to use kdiff3 is beyond the scope of this document but I encourage you to learn how to use the software. To configure git to use kdiff3 to resolve conflicts here are two links:

Windows: http://jebaird.com/2013/07/08/setting-up-kdiff3-as-the-default-merge-tool-for-git-on-windows.html

MacOS: http://naleid.com/blog/2012/01/12/how-to-use-kdiff3-as-a-3-way-merge-tool-with-mercurial-git-and-tower-app (also has information on how to use kdiff3. you can ignore the sections on how to integrate with Mercurial).

When Git reports a conflict you can begin the process of resolving it by

```
$ git mergetool
```

This will launch kdiff3 for each file that has a conflict. You will need to go over each difference in each file and select the correct version of the merged file. When you are done with a file you will save the result and quit kdiff3. Git will ask you if you have resolved the conflict for the file or not. If not then git will stop all processing and you will need to restart the process when you are ready. If yes then Git will then relaunch kdiff3 for the next file in conflict.

After you have resolved all the conflicts you will then need to tell git to resume the merge.

```
$ git commit
```

After the merge is complete you will notice that the merge tool has left some extra files in the code. These files are named the same as the file that had the conflict but with a .orig extension added to the name. These files can be safely deleted.

Manual resolving

If you know you want to use your version of the file, or the other version of the file or if you feel like editing the file by hand to resolve the conflict you can do that too. To use your (current branch) or their (branch being merged) version of the file you can use one of the two following commands

```
$ git checkout --ours file_name.php
$ git checkout --theirs file_name.php
```

Be careful in doing this as any changes in the other version of the file will be lost.

To manually edit the file you ca open it in your editor of choice and look for the following pattern:

```
I'm a little teapot short and stout
here is my handle
<<<<<<< HEAD
here is my spout
=======
Whoops! I'm a sugar bowl
>>>>>>> master
```

The version being merged is on top and the version in the current version is on the bottom. You can edit the file to replace this structure with the version you want and save the file. Be sure to remove all the conflict structure. Anything left in the file will be considered the correct version and that includes any thing accidentally left behind.

After you have resolved the conflict you can commit you merged files.

```
$ git add file_name.php
$ git commit -m "Merged branch correct_lyrics into master"
```

Rebase

Rebasing is a special situation where you change the parent commit for your branch to another commit. This is useful when you have been working on a feature branch based off another branch and in the time you have been working on your branch the parent branch has moved forward with other commits and merges. A rebase lets you bring your branch in line with all the changes that have happened since you created your branch.

In a technical sense what is happening is git is rewinding your branch to it's start and then replaying each commit on the end of the branch specified as the new parent. This creates a series of new commits. The difficult part of this process is that each time Git replays a commit it could potentially create a conflict. Also it is best to do this on a branch that has not been pushed to a remote yet as it causes all the commit ids to change which will create problems for the remote repo.

To start a rebase

```
$ git fetch origin
$ git rebase origin/master
```

If everything worked without problem you are free to continue working. If there was a conflict then you will need to resolve the conflict. You can use a similar process to resolve merge conflicts as shown above. With a small change. After you finish resolving the conflict you will need to run

```
$ git rebase –continue
```

This will continue replaying the commits. Be prepared to do more conflict resolution as each commit

that is bein replayed could end up being in conflict.

If for whatever reason in the middle of attempting to resolve the conflicts you want to stop the entire process and revert back to where you started from you can issue the command

```
$ git rebase --abort
```

Rollback

If you see that there was a problem with a commit that was merged in to your branch and want to go back to the state of your branch before the bad commit you have two options in front of you. Revert or Reset. The two will have the same effect of your files but differ in the behind the scenes stuff like the project history. Lastly you can also use checkout to revert individual files.

Revert

A revert will attempt to create an "undo" commit that will completely undo any changes that the commit in question introduced. It will then add this new commit to the branch. This has the bonus of keeping the project history intact and does not interfere with other copies of the branch on other repo clones. The downside of revert is that it works on only one commit at a time. The upside of that is that it can be applied to any commit.

To use git revert to revert a specific commit:

```
$ git revert aa8858d1a519fa459b484b803cbe49629aa38b32
```

or to use the relative from HEAD format to revert the commit that is two back from current HEAD:

```
$ git revert HEAD~2
```

Reset

A reset instead repoints the current HEAD back to a previous commit. This is a "risky" option because if you reset a branch back to a previous state then all commits that came after the point you reset back to are permanently lost. There is no way to take a mulligan from a git reset. Git reset is only able to reset back to a commit in the current branch and only as a relative number of jumps back from the current branch HEAD. Reset does have it's place though and if you know how to use it properly it will become a powerful tool in your git toolbox.

To go back to the commit two back from the current HEAD

```
$ git reset HEAD~2
```

To get rid of all uncommitted changes in the working copy

```
$ git reset --hard
```

To remove a file that was accidentally added to the stage but before you commit

```
$ git reset – file_name.php
```

Checkout

Checkout can be used to bring a single file back from a previous commit into the current working copy

```
$ git checkout HEAD~3 file_name.php
```

Or if you want to checkout from HEAD

```
$ git checkout -- file_name.php
```

Stashing

If you have a bunch of uncommitted changes to your files that you want to keep but are potentially in conflict of a pull, merge, checkout, etc. Git will give you a warning and refuse to do the action. You can use the stash to put your modified files in a safe place, put the working copy back into the state of the HEAD so that it is safe to perform the operation that then use stash to bring your changes back.

```
$ git stash
```

Then when you are ready to bring your changes back

```
$ git stash apply
```

If you make a stash then forget to apply it and continue working it becomes difficult to impossible to apply your stash. In this case you can create a branch from your stash and perform a merge into your branch.

```
$ git stash branch my_new_branch
```

Whoops! I was on the wrong branch when I committed

If you forgot to switch from master branch to your feature branch, made a commit and then realized your mistake just the moment after you pressed enter: don't panic. You can recover from this.

```
$ git branch newbranch
$ git reset --hard HEAD~3
$ git checkout newbranch
```

Tagging

A tag in git is like a little flag you stick into a commit to say "This particular commit is special". Tagging is used for many purposes but the most useful is for flagging releases. Multiple tags can be applied to a given commit.

Creating

To create a tag on the current HEAD:

```
$ git tag my_tag_name
```

Git also allows for adding messages to a tag similar to a commit messages

```
$ git tag -a my_tag_name -m "This tag has a message"
```

Tags with messages are called annotated tags. Tags without messages are called lightweight tags.

The standard commit referencing options can be used to tag any commit in the history.

Switching

To switch to a tag you simply use a checkout

```
$ git checkout my_new_tag
```

Git Flow

What is Git Flow?

Git flow is simply a methodology for using Git. It's been developed over time as people found efficient ways of using git to manage their workflow as well as their code.

Git contains no built in controls to ensure you follow git flow. Git flow is something your team needs to agree to follow and then actually follow. That's the hard part. As with any good workflow system it makes your life easier on average but it makes some small repetitive tasks more work. It's often these small tasks that make a developer fall off the wagon.

Why use it?

The reason for using git flow is simple. It makes development predictable and repeatable. It makes sure that all team members can stay up to date on what's happening or to figure out the situation when they need to figure something out.

It also helps a codebase remain navigable and the history clean.

It also provides a useful buzzword for helping with the onboarding process for new team members. It's easier to say, "We follow git flow", and have the new employee read up on it than to say, "Yeah... Our process is kinda scattered but don't worry you'll pick it up soon", then watch as ever team member does their own thing and makes a mess of the project history.

How Git Flow works

In Git Flow a single repository is considered the primary. This repo has the last word on anything.

Important branches

On this repo two branches are created: master and development. These two branches have different purposes. Master is where releases are taken from and all commits in the master branch are to be considered official releases. For this reason it is usually good to make sure that only certain people/roles have access to push up to master.

Development branch contains the full history of the project. All feature branches are created off of development and rebased to development and merged back into development.

Feature branches

Every task that needs development work gets it's own feature branch. Feature branches should be named according to the issue number in the issue tracking system. Every commit message should begin with the ticket number. For example: "Ticket-503: Updated scoring algorithm to take distance into account".

When a feature branch is finished, the feature branch is rebased to development and a pull request is made to the merge manager to pull the feature branch into development.

Release branches

As your release comes near a feature freeze is put in place, a release branch is created from development and then the feature freeze is lifted. From that point forward only bug fixes, documentation or other related tasks can be performed on the release branch. Once the release branch is deemed ready for releaseit is tagged using the project's agreed upon release numbering system and merged into both master and develop.

Feature branches that were not ready to be included in the release branch or were started after the release branch was created still get merged into development but will have to wait until the next release is ready to go before making it out to prod.

Hotfixes

Sometimes a bug is discovered after a release has been out for a while and it's not serious enough to force a full rollback but it's serious enough to not wait until the next release can be handled in a maintenance or hotfix branch. These branches are created directly off of master and only tackle the single issue to be addressed by the hotfix. Once the hotfix is ready it it tagged and merged into master and development and possibly any currently open release branch.

Caveats

Git Flow is a good workflow to follow but it is far from the only one out there and certainly not perfect.

A big problem with git flow is that for small dynamic teams that are looking to make frequent small releases they will be frustrated by the slow process required to get a release out. Small teams will also find it difficult to deal with the issue segregation requied by feature branches. It adds extra overhead to a developer who is working on possible several issues at once as often happens in small teams.