GIT INSTRUCTION MANUAL

Evan Bergeron
Sunny Gakhar
Nishad Gothoskar
Frederick Lee
Ziyang Wang

April 26, 2017

Contents

1.

Introduction

This is a manual on how to set up Git on your computer and set up a basic Git workflow. We cover installing, setting up, and working with git. Intermediate workflow tips are provided, as are recommendations for popular editor integrations.

This document is intended for users who want to learn Git to use it in a fast-paced setting such as a hackathon, this document is also useful as a reference for experienced Git users who want to refer to some specifc concept or command which they need.

How to use this document

Complete beginners to `git` will want to start with chapters 2 and 3. Chapter 4 covers branches; a topic perhaps not essential to basic usage. Readers already familiar with `git` may find the final chapters helpful in elevating their workflow efficiency.

What is Git?

Git is a version control system designed to be used for working on small and large projects. It can be used for tracking changes between files and coordinating work on project files among multiple people.

Why use Git?

When you were in school and you had short homework assignments, you would just start them and finish them in a short span of time (not longer than a week). But when you move on to designing and working on bigger projects with other people, there are multiple issues that come into play. Say you are participating in a hackathon and have finalized your idea and distribution of work among the teammates. How do you actually work on the project together?

Having all of them work on one computer is not optimal. You might have each teammate work on his own piece independently, but how do you merge everyone's work? Moreover, what if two or more teammates work on the same file, but do different modifications unknown to the others? And what if someone wants to explore a different direction to work on, while keeping the original work intact? Enter Git.

Benefits of Git

1. Using Git, if you're working on a project, you can "commit" the changes you have made and Git will keep track of all your commits.

2. If you want to explore a new direction of work which you don't want to integrate with your main project just yet, you can create a new branch and work on the branch without disturbing your main project. You can easily switch between multiple branches to work on multiple features, can when the time comes, you can merge with the main branch.

2.

Installing Git

For Windows

If you have Windows, one easy way of installing Git is from this website:

`https://git-scm.com/download/win`

Once you have downloaded the installation file, you can run it and proceed through the installation steps.

For Mac

One way of installing Git on is from this link

`https://git-scm.com/download/mac`

Another way of installing Git is using the Xcode Command Line Tools. Open up Terminal and simply type `git`. If you dont have Git installed already, it will prompt you to install it.

For Linux

If you're working on Linux, you can install Git using a basic package management tool that comes with your distribution.

Debian/Ubuntu

```
$ apt-get install git
```

Fedora

```
$ yum install git
```
(up to Fedora 21)

`$ dnf install git` (Fedora 22 and later)

Arch Linux
`$ pacman -S git`

FreeBSD
`$ pkg install git`

Solaris 9/10/11 (OpenCSW)
`$ pkgutil -i git`

Solaris 11 Express
`$ pkg install developer/versioning/git`

OpenBSD
`$ pkg_add git`

3.

Getting Started

Initializing Repository

To make sure you have Git set up, type `git` into your console (Terminal for Mac/Linux users and Command Prompt for Windows users) and the following should show up (the full output has been elided here).

```
$ git
usage: git [--version] [--help] [-C <path>] [-c name=value]
[--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
[-p | --paginate | --no-pager] [--no-replace-objects] [--bare]
[--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
<command> [<args>]

...

'git help -a' and 'git help -g' list available subcommands and some
concept guides. See 'git help <command>' or 'git help <concept>'
to read about a specific subcommand or concept.
```

To start a new repository, first go to type `git init` into the console. It should output the following:

```
$ git init
Initialized empty Git repository in <path to current directory>/.git/
```

This creates a `./git` directory in your current directory, which consists of all information about the repository. It consists of a HEAD file, which points to the current version of the repository.

Adding files to version control

The staging environment, or sometimes referred as index, is a file that stores information that you want to edit to the git repository. After you have initialized a repository, you may want to add new files to the repository. However, git does not know which files you want to include in the staging environment. `git add` command is used for adding files to the staging environment.

For example, after creating a `test.txt` file in your working directory, you can check the current status using `git status` command. You should get the following message:

```
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    test.txt

nothing added to commit but untracked files present
(use "git add" to track)
```

In order to add the file to the staging environment and notify git for this change, type `git add test.txt` and check status. You should get the following message:

```
$ git add test.txt
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    New file: test.txt

```

Now the new file is ready for commit, i.e. ready to be added to the git repository.

If the file you want to add is not in the working directory, use `git add <pathspec>` instead to add the file to the staging environment.

Committing changes

A commit is a record of what files you have added or changed since the last time you committed changes. Running the command `git commit -m "Message for the commit"` creates a new commit for all the changes you have made. The message after you creating the commit should contain relevant information about the changed you made.

For example, if you changed some contents in the file `index.html` and created a new file `new.txt`, a commit message should look like something as following:

```
$ git commit -m "Change index, add new file"
[master 3651e36] Change index, add new file
2 files changed, 10 insertions(+)
create mode 100644 new.txt
```

The `"-m"` flag allows you to give a message without opening a text editor. Alternatively, using `git commit` will open your text editor with some message like the following:

```
$ git commit
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Your branch is up-to-date with 'origin/master'.
#
# Changes to be committed:
# new file:   new.txt
# modified:   index.html
#
~
~
~
".git/COMMIT_EDITMSG" 9L, 283C
```

It is worth noting that anything that is still unstaged (files you have created or modified that you havent run git add on since you edited them) wont go into the commit.

Creating a commit does not change any remote repository. If you are using GitHub, you probably want this commit changes respective contents in the remote repository on GitHub. You can use `git push` to update remote repository with committed changes.

```
$ git push
```

Reverting changes

One reason people using git as version control is that it can undo changes easily. When you make a new commit, git stores the status of your repository at that specific moment. In the future, you can go back to an earlier version by going back to the status git stored.

Here are several undo commands and the scenarios for using them:

1. `git revert <SHA>`. You have used `git push` to change your remote repository, but want to undo the change. This command creates a new commit that does the inverse of the given SHA. In other words, it undoes the previous commit by giving opposite changes.

2. `git commit --amend`. You created a new commit but have not pushed it to remote repository. There is a typo in your commit message. This command allows you to retype the commit message.

3. `git checkout -- <filename>` You want your files to go back to the state in the last commit. `git checkout` changes file in the working directory to a state previously recorded by git.

4. `git reset <SHA>`. You created several commits but have not pushed them to remote repository. The changes you made in the last commit is horrible and you want to go back to certain commit. `git reset` rewinds your repository's history back to the specified `<SHA>`.

5. `git reflog`. You used `git reset` to undo changes, but now you want redo them. `git reflog` recovers your directory's history.

Ignoring Files

The best way to ignore files in a repository is to create a `.gitignore` file at the base of your repository with the names of all files to ignore.

Often you will find that in your projects, you will have files in your repository that do not need to be tracked. These include automatically generated files, larger libraries that are really external dependencies, or system specific files such as the `.DS_Store` file in Macs.

For example, let us have a repository with two files: `code.py` and `generated.txt`. Here we only want to keep track of changes to `code.py` while ignoring changes to `generated.txt` which is automatically generated every time `code.py` is run.

Without a `.gitignore` file, running `git status` would get the following output:

```
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    code.py
    generated.txt

nothing added to commit but untracked files present
(use "git add" to track)
```

We create a `.gitignore` file with the following content:

```
generated.txt
.gitignore
```

Running `git status` again, we see that the generated file and the `.gitignore` itself are ignored by Git as desired.

```
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    code.py

```

```
8  nothing added to commit but untracked files present
9  (use "git add" to track)
```

4.

Branching

In git, branching is used to keep track of the different paths of development for your project. The master branch is the main branch of your project and after features or changes are verified, they are usually added to the master branch of the project.

Creating branches

To create a new branch (for example if you want a new branch named addfeature):

```
1 $ git branch add_feature
```

This creates a new branch in the development tree. Now you can work on this branch of the project without effecting the master branch.

As the figure above shows, branching creates a separate flow. Now in order to bring your current local state to that branch you will have to check it out by:

```
1 $ git checkout add_feature
```

Now the changes you make will be modifying this new branch rather than the master branch. To go back to the master branch you will use the same checkout command with master rather than addfeature.

Note that to checkout a different branch your current changes must be committed or stored away before you can switch.

A simple shortcut is to run:

```
$ git checkout -b add_feature
```

Note that to checkout a different branch your current changes must be committed or stored away before you can switch.

Now when you are done with development you can delete your branch by using:

```
$ git branch -d add_feature
```

If you want to rollback and see the state of your project at some previous point, you can use `git log` to look at the commit history and find the commit hash

and use:

```
$ git checkout 1c0191a6a6264f2e90f6905636321fde897b86df
```

To checkout the previous commit and see the state of the project at that point.

To combine branches, we have two main options: merging and rebasing.

Merging Branches

Now after working on these separate branch and adding your new feature, you want to merge this work into your master branch so it will persist and new branches can build upon or use the feature you added. This is what merging does. It takes a branch and merges it into another branch. This can be done by first checking out the branch you want to merge INTO:

```
$ git checkout master
```

And then merging the other branch in by:

```
$ git merge add_feature
```

This may happen immediately, unless there are conflicts between the two branches that need to be resolved before they can be successfully merged. By using git status, you can see in which files these conflicts are located. Then within the files you will find sections of the form:

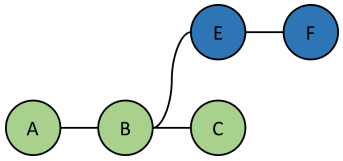
```
<<<<< HEAD
stuff in the current branch
=====
things in other branch
>>>>> add_feature
```

And you will have to replace these sections with the content you want and then commit the changes to fully process the merge of the two branches.

Rebasing Branches

The rebasing and merging are the two primary ways of combining changes from separate branches with Git. One primary advantage of rebasing is that the Git history is made more linear which makes it easier to track changes over time.

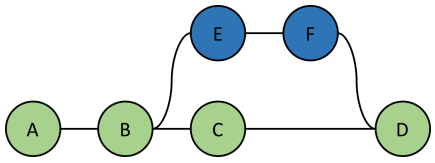
For example, say that you have been working on a separate branch called `feature` (shown in blue) on which you have two commits. At the same time, a friend of yours has committed change "C" to master (shown in green).



A regular merge command such as

```
$ git checkout master
$ git merge feature
```

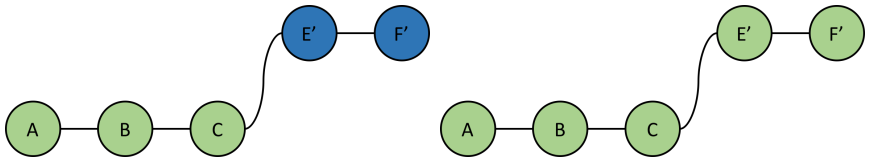
would result in the following Git history:



However, if you want a more linear history, you would ideally want your commits "E" and "F" to be stacked on top of your friends commits. To do as such, simply run as follows:

```
$ git checkout feature
$ git rebase master
$ git checkout master
$ git merge feature
```

The initial rebase command changes the root of your `feature` branch to your friend's last commit as shown below on the left image.



The final merge of your feature onto master then forwards master's history to match your feature branch as shown above on the right. As you can see, the history follows a much more linear pattern compared to the branching pattern of a regular merge alone.

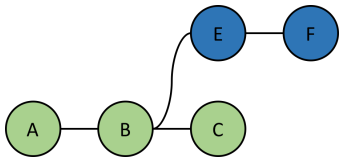
5.

Intermediate Git

Cherry-picking Commits

Cherry-picking is particularly useful when you don't want to apply all changes from a branch to another branch through a merge or rebase, but only want to apply a few specific commits.

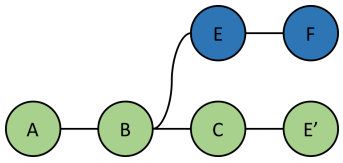
For example, take the hypothetical situation where your most recent commit "F" in your feature branch (shown in blue) is suffering from a application-breaking bug, but your manager wants your changes in commit "E" to be rolled out by this afternoon.



To apply commit "E" by itself to the master branch (in green), first find the commit-hash of your desired commit by running `git log` and searching for your commit. A commit-hash is an identifying string of letters and numbers that may look something like `f3def414605008d5c899b2691ff1b6d1d3798a0a`. After you have found your commit-hash, to apply your desired commit to the master branch, run the following commands:

```
$ git checkout master
$ git cherry-pick <commit-hash of E>
```

Running these commands gets your final desired result shown below:



where commit "E" has been applied to the master branch from your feature branch without merging your entire bug-ridden feature branch to master.

Collaboration with Github

Github is a website that manages git repositories. You can create an account on the site. Ever user gets an unlimited number of public repositories. Be warned that public repositories can be viewed by anyone accessing github.com. By default, you retain the copyright to whatever code you post to Github.

You can use github for a more user-friendly version controlling experience. Almost everything you can do using the command line can be done using the desktop client. You can easily select files, read their diffs, and commit them.

Using the online interface, you can create repositories, add contributors, and track progress very easily.

The workflow using github usually involves many collaborators who make pull requests to the managers of the repository.

Easily Reordering Commits

Reordering commits is a common task and can be performed in a single command. A naive workflow for this task might look like

```
$ git checkout -b tmp
$ git checkout master
$ git reset --hard HEAD~
$ git cherry-pick tmp
```

This works fine, but theres a simpler method. We can simply say:

```
$ git rebase -i HEAD~2
```

This will open an interactive git rebasing session (the `-i` stands for interactive). The window will display something along the lines of

```
pick 370e221 Commit one
pick c342396 Commit two
```

Simply reorder these lines to reorder the commits. Much shorter!

Adding partial files

It can sometimes be helpful to stage only part of the changes made to a single file. (Especially if there are two logically different salient tasks). A naive workflow might involve creating separate branches and manually editing files. Instead, we may say:

```
$ git add -p <file>
```

This will bring up an interactive prompt. This prompt will cycle through the different areas of the diff. For each section, you will be asked if you want to stage each section. You may hit `y` or `n` for yes or no.

Once you're done adding the subset of changes you want to commit, you can double-check you have the right changes staged by saying

```
$ git diff --cached
```

One everything looks good, simply `commit` as normal.

Git aliases

Typing out full commands such as "`git commit`" and "`git log --oneline`" can take a lot of time. Git aliases are a useful tool to save yourself commonly typed commands. For instance, we might say

```
$ git config --global alias.l "log --oneline"
$ git l
```

to save ourselves typing out "`log --oneline`" everytime. It is worth spending a couple moments to create a handful of shortcuts.

6.

Vim Integration

Vim is a popular text editor. Its key feature is called modal editing, which allows users to edit text very quickly. Many professional software developers and professors use vim every day. You should read this chapter if you use vim.

At the time of writing, perhaps the most feature complete vim-git plugin is Tim Popes vim-fugitive. Consequently, we will assume usage of this plugin throughout the entire vim workflows tutorial.

Installing vim-fugitive

There are a number of ways to install vim-fugitive. The one suggested by Tim Pope is as follows:

```
$ cd ~/.vim/bundle
$ git clone git://github.com/tpope/vim-fugitive.git
$ vim -u NONE -c "helptags vim-fugitive/doc" -c q
```

Vundle is a great plugin manager for vim – if you use this, you may simply add the line

```
Plugin 'tpope/vim-fugitive
```

to your vimrc and run the `PluginInstall` command.

Using vim-fugitive

`vim-fugitive` provides a number of functions that wrap common command line `git` commands. All of these can be mapped to a keybinding of your

choosing.

- `Glame` opens a vertical split immediately to the left of your source code showing the output of `git blame`. `git blame` can be very helpful when browsing a larger codebase,

  A naive solution might first exit vim, manually type `git blame <filename>`, then search for the relevant line in the resulting output.

- `Ggrep` corresponds to `git grep`, a common way to search large codebases for keywords. The output opens in another buffer, which the authors finds more useful than a separate `tmux` split.

- `Gread` is similiar to `git checkout -- <file>`, but `Gread` operates on a buffer rather than a file. You can use `u` to undo `Gread`, which the authors finds much cleaner than dealing with various warnings external to vim.

7.

Emacs Integration

At the time of writing, Magit is the most feature-complete git wrapper for emacs. We will thus assume usage of this package.

Installing Magit

Perhaps the easiest way to install is through MELPA. Just run

M-x package-install RET magit RET

Using Magit

Magit's interface is extensive, so cover merely the basics here. In this section, we adopt the typical `emacs` key stroke notation.

- M-x magit-status RET opens an interface buffer. n moves to the next item, p moves to the previous itme. Press <TAB> to expand the selected file. This will open the diff of that file.

  In the file diff view, git add -p functionality is readily available. Simply while cursor'd over a part of the diff, type s to stage this chunk of the diff.

  When the appropriate chunks have been staged, press c to commit and P to push.

- `M-x magit-log RET master RET` displays an interface log of the git repository. `magit-log` presents a similar tree structure as `magit-status`. `n`, `p`, and `<TAB>` are used similarly for navigation.

`q` exits almost any screen in `magit`.