Git - Command Line Guide

Anthony McGlone

May 23, 2022

Table of Contents

Chapter 1 Introduction

1.1 What is Git?

Git is a repository which is used to store files. Git is used by Software Engineers to store and version code for software releases. It's a powerful tool for collaborating on large projects.

Git versions the files within its repository, so every time a commit operation is made, the files in the repository are saved - a unique ID is also created and associated with the commit. This means that you can see the history of the changes in the repository. You can also delete the changes in the repository by going back to a specific commit.

In this guide you will see how the command line tool `git` is used to manage this repository.

1.2 Installing Git

1.2.1 MacOS

1. Open a terminal. Then install Homebrew (steps located: here).
2. Install Git by running `brew install git`
3. Open a terminal and run `git --version` (if Git is installed, the version number should be printed to console)

1.2.2 Windows

1. Install Git by downloading the Windows binary from git-scm (here)
2. Run the installer
3. Open a Git Bash terminal and run `git --version` (if Git is installed, the version number should be printed to console)

1.2.3 Linux (Ubuntu)

1. Install Git by opening a terminal and running `sudo apt-get install git`
2. Run `git --version` (if Git is installed, the version number should be printed to console)

1.2.4 Linux (Red Hat)

1. Install git by opening a terminal and running `sudo yum install git`
2. Run `git --version` (if Git is installed, the version number should be printed to console)

Chapter 2 Basic Git

2.1 Setting up a local repository

First create a folder called `store`. If you're on MacOS or Linux, create it under the `/home/` folder, so your `path` to the folder will be `/home/store`. If you're on Windows create it under the `C` folder (when you're navigating to it the `path` will be `/c/store`). Now, open a terminal and navigate into that folder using the `cd` command (i.e. `cd /c/store`).

Now, run this command:

```
git init
```

This command creates the repository and also creates a `.git` subdirectory. This subdirectory stores information about commits, and the location of your remote repository. This remote repository is stored on a server (hosted on the internet or on a company's internal network). To share your commits with others, you have to push your local commits to the remote repository. We'll set up the remote repository later.

Let's add your username and email to your local repository now. This will be required so your commits can be associated with you. Start with the username. Run the following command (before doing so, add your full name between the quotes in the command below):

```
git config --global user.name "INSERT NAME HERE"
```

Now run the command to set up your email (again, insert your email address between the quotes):

```
git config --global user.email "INSERT EMAIL HERE"
```

Now run the command `git config --global user.name`. If everything is correct, you should see your username. Run `git config --global user.email` to see if your email prints out to console.

Run the following command to verify that your local repository was set up:

```
git status
```

You should see the following text in your terminal:

```
On branch master

No commits yet

nothing to commit (create/copy files and use "git add" to track)
```

That's it! Your local repository is now successfully set up. The branch `master` is the main branch in your local repository. By default, it's the place where your files are stored. You can create other branches (basically copies of `master`) and work on edits there before saving them back into the `master` branch. For now, we'll just work with the `master` branch. We'll cover branching strategies and remote branches (stored in a remote repository) later.

2.2 Committing files into the `store` repository

First, create a `code.txt` file in the `store` folder. Then run `git status` in your terminal. You should see the following output:

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

        code.txt
```

The file is untracked, meaning Git doesn't see it yet. If you tried to do a commit operation now, Git wouldn't save the file or version it. Run the add operation below in your terminal to get Git to track the file.

```
git add code.txt
```

Now run `git status` again to see the tracking:

```
On branch master

No commits yet

Changes to be committed:
    (use "git rm --cached <file>..." to unstage)

        new file:  code.txt
```

Now commit these changes by running `git commit -m "This is my first commit!"` You can run `git log` after to see the commit ID. The ID is the code after the word `commit`:

```
commit b973b70df3ffcfd8dbe0284c76e0d3bd7c30f3b6 (HEAD -> master)
Author:  anthonymcglone2022 <anthonymcglone2022@gmail.com>
Date:  Wed May 11 20:14:36 2022 +0100

    This is my first commit!
```

2.3 Creating a remote repository

The next step to take is to create a remote repository, where you can upload your changes. GitHub is a website that offers free hosting of remote Git repositories. In order to create a repository there, there are a few steps that have to be completed.

- Signup to GitHub: here
- Generate an SSH key: Procedure here (select one of Mac, Linux, Windows)
- Adding that SSH key into GitHub: Procedure
- Create a repository (name it `store` and skip the README initilization step. Stop after Create repository step): Procedure

Let's push the commit we made to the remote repository. First, we will point our local repo to the remote repo. Take your GitHub user name and replace `username` in the following command - then run it.

```
git remote add origin git@github.com:username/store.git
```

To see if your local repository was updated, run `git remote -v` in your terminal. You should see `git@github.com:username/store.git`. Now, push your code to the remote repository (i.e. your remote master branch). Run the following:

```
git push origin master
```

Open up your `store` repo in GitHub. If your command was successful then you should see the `code.txt` file inside it.

2.4 Basic Branching and Merging

At this point, you have shared your work with others. Your local master branch is synched with the remote master branch. Your colleagues will clone your repository to their local machines using the `git clone` command. They will make changes, commit them to their local master branches and push them up to the repository that you just created. Git's strength is that it will track these changes. It also offers the tools to make sure that you don't overwrite anyone's work when you push up your own changes. We'll see how to do this now. Then we'll talk a little about branches.

Let's mimic a colleague by cloning your remote repository to a different folder on your machine. Make sure this folder is outside of your existing `store` directory. Create a folder called `colleaguesrepository` and then navigate inside that folder using the terminal. Then run the following command (replacing `username` with your GitHub user name):

```
git clone git@github.com:username/store.git
```

When the command finishes, you should find another copy of the `store` directory downloaded. Create a file called `colleaguesfile.txt` in this new `store` directory. Then navigate inside the directory using the terminal. We will add, commit and push this file to our remote repo. Run the following:

```
-git add colleaguesfile.txt
-git commit -m "I am a new colleague committing some new code"
-git push origin master
```

Ok, so our colleague has committed their changes. We have our own work to finish, and we need to push it to remote. So let's do that now. Use your terminal to navigate back to your **initial** `store` repository. Create a file in there called `mysecondfile.txt`. Then run the add, commit, push sequence of commands:

```
- git add mysecondfile.txt
- git commit -m "I am pushing up my work now"
- git push origin master
```

Now your push should be rejected. You will see a message like:

```
To github.com:username/store.git
!  [rejected] master -> master (fetch first)
error:  failed to push some refs to 'git@github.com:username/store.git'
hint:  Updates were rejected because the remote contains work you dont
hint:  have locally.  This is usually caused by another repository pushing
hint:  to the same ref.  You may want to first integrate the remote changes
hint:  (e.g., 'git pull ...')  before pushing again.
hint:  See the 'Note about fast-forwards' in 'git push --help' for details.
```

This occurred because your remote repository has been updated with your colleague's work. This is how Git is useful. It will stop you from overwriting another person's work. Another mistake we made here was to commit our changes directly to local master. This is where branching strategy becomes important. You should always create a local branch from your local master. Then if your colleagues push to remote while you are working on your changes, you can pull their changes down to your local master. That way it always stays in synch with the remote master branch. When you're ready to push your own changes up, you can "merge" your changes into your updated local master and push it to remote. Let's do that now. Firstly let's undo the commit we just made. Run:

```
git reset --hard HEAD~1
```

If you run `git log` now, you will see that your most recent commit has been wiped from the Git version history. The file `mysecondfile.txt` is also gone. This is what a hard reset does. A soft reset `git reset --soft HEAD~1` would undo the latest commit but keep the files associated with that commit (it will take you back in time to the point just before you made your last commit). We'll explore poweruser commands like these in more detail later.

Let's create a new branch (we'll call it `first`) to make our changes on:

```
git checkout -b first
```

If you run `git status` now you will see the output `On branch first`. If you run `git log` to see your commit history, you will see that you only have one commit. Your branch `first` is an exact copy of local master. You can switch between branches by running `git checkout branchname` replacing `branchname` with the name of the branch you want to switch to. As an excerise, switch now to `master` and examine your commit history, then switch back to the `first` branch.

Now, let's commit some changes on the `first` branch. Afterwards, we will try to merge our changes to local master. Start by creating a file in the `store` repo called `mysecondfile.txt`. Then run the add and commit commands:

```
- git add mysecondfile.txt
- git commit -m "My first commit on a branch"
```

Run `git log` to check the commit was successful. Now let's push our changes to remote. The following gives an overview for the commands below it:

```
- Switch back to local master
- Get our colleagues changes from remote master
- Merge our changes into updated local master (2 commands)
- Push our changes to remote master
```

Here is the command flow:

```
- git checkout master
- git pull origin master
- git merge --squash first
- git commit -m "Commiting branch changes"
- git push origin master
```

If you check your GitHub repository now, you should see that `mysecondfile.txt` is added to it. Now let's unpack what happened with the above commands.

The command `git pull origin master` was used to pull the remote changes down to our local master branch. You must be on the master branch to do this. You should always pull down the remote changes before you merge your work to local master. This ensures your master branches are always synched.

Always switch to the branch you want to merge your changes into. We wanted to merge our work (i.e. the `first` branch) into master, so we ran our merge command while being on master. The `--squash` argument squashes multiple commits into one commit. That is why we had to run a second commit on the local master branch. It's good practice to squash your commits into one, because it makes it easier to read the commit version history when you run `git log`. Finally, you always give the name of the branch that has to be merged at the end of the command.

2.5 Merge Conflicts

In the last section, we had a merge that was straightforward. However, this does not happen often in the real world of big projects and multiple engineers. A colleague of yours might be working on the same files as you at any given moment, and may even be working on the same lines of text or code within those files. This can lead to Git throwing up a "merge conflict". You will have to resolve these conflicts before you can push your work to remote. We'll see

see how this happens in this section, and you will learn how to fix it.

We're going to pretend that our Boss has asked us to add our suggestions for improving team efficiency to `mysecondfile.txt`. Our Boss wants all the team's suggestions, because it's useful for an open discussion later. So let's pretend to be our colleague again. Navigate back to your colleagues `store` folder. Then pull the latest changes down to the master branch. Run `git log` afterward to make sure your colleague has got the changes you pushed up a few moments ago. We'll not create a branch this time (for brevity) but you can assume that your colleague would have branched off and gone through the merge process.

Open up the `mysecondfile.txt` and add the text `Have less meetings` on **line 1**. Make sure you only add the text to the first line. Save the file. When you are happy with the changes, run the add, commit, push flow as usual:

- `git add mysecondfile.txt`
- `git commit -m "My colleague is making another commit"`
- `git push origin master`

Now navigate back to your **original** `store` folder using the terminal. Let's create another branch called `second`. Run:

```
git checkout -b second
```

Open the file `mysecondfile.txt` and add the text `Do more off site team building` on **line 1**. Save the file, then run the following commands to do the merge:

- `git checkout master`
- `git pull origin master`
- `git merge --squash second`

You should see the following text:

```
Auto-merging mysecondfile.txt
CONFLICT (content): Merge conflict in mysecondfile.txt
Squash commit -- not updating HEAD
Automatic merge failed; fix conflicts and then commit the result.
```

We need to fix the conflict that has arisen. Here's what our text file `mysecondfile.txt` looks like now:

```
<<<<<<< HEAD
Have less meetings
=======
Do more off site team building
>>>>>>> second
```

The text between `HEAD` and ======= is what is in the remote repo - what your colleague pushed up. The text below the ======= is what is on your branch `second`. Since you and your colleague have been working on the same area of the same file, Git does not automatically know which edits to keep. In this case it's quite easy to decide what to keep - we want to save both. However, sometimes, it may not be immediately apparent what you need to keep (like in the case that your colleague is refactoring code, but you're working on that same code during the refactor). When this happens, you have to sit down with your colleague and see what needs to be kept.

Ok, back to our current merge. We know that we want to keep both lines of text, so let's edit our file and resolve the conflicts with `git`. Open `mysecondfile.txt` and remove everything except the two sentences. Then save the file. On the command line, run the add command to resolve the conflict fully:

```
git add mysecondfile.txt
```

Then run the usual flow to push up your changes:

```
-git commit -m "Resolved some conflicts from my branch second"
-git push origin master
```

2.6 Rebasing a branch

In section Basic Branching and Merging, we saw a rather straight-forward merge. In the section Merge Conflicts, we saw how the merging process can become rather complicated. However, there are a number strategies to keep merging easy, especially when your working on a local branch for hours or days at a time. If you keep your local branch up to date while you're working on it, merging to master can be quite a painless process. In this section, we will see a process called rebasing, which will allow us to do just that.

Create a new branch (called `testrebase`) in your own `store` repo. Create a new file called `rebase.txt`, run the add and commit commands only. Run `git log` and examine the version history (take note of your most recent commit).

Switch over to your colleagues `store` repo. Pull down the latest changes to local master. Again, we will not branch off. Create a file `anewfile.txt`, add it, commit it and push it to remote. Repeat this process with another file called `anotherfile.txt`. When you are done, you can check your version history to see the commits (remember this for when we do the rebase).

Okay, we are ready to do the rebase. Navigate back to your own repo. Switch to the master branch and pull down the latest changes. Check the version history and compare them with the version history of your colleagues `store`. When you're ready, switch to your `testrebase` branch. Run the following command to rebase.

```
git rebase master
```

You should see the following message in your terminal

```
First, rewinding head to replay your work on top of it...
Applying:  rebase file
```

Now if you run `git log` on your branch, you should see that your commit is at the top of the version history, with your colleague's commits underneath it. This is what a rebase does, it re-bases the changes on your branch against another branch (master in this case). That way, you can keep up to date while you are working on your changes.

2.7 Stashing your changes

During development of a new feature, you may be asked to fix a software defect. You may have just started work on the feature, and you have some changes, but you're not ready to commit them yet. You have to create a new branch and switch to it but you don't want to lose your work. This is where the `git stash` command comes in. We'll see an example of how to do this now.

Create a new branch called `stashexample`. Create a new file called `filetostash.txt`. Now run:

```
git stash list
```

There should be no output in your terminal screen. That is because there is nothing in your stash. Run the following commands:

```
-git add filetostash.txt
-git stash
```

If you run `git stash list` now, you should see the following:

```
stash@{0}:  WIP on stashexample:  <last commit> <commit message>
```

If you run `git status` now, you should see `nothing to commit` in your terminal. So at this point, you're free to move away to create another branch. But we'll stay on this branch - we can assume that we've fixed the software defect and now we have come back to continue our work. We want to get our changes out of stash. You can do this via running the `apply` command. You have to give it the id of the entry in the stash list (in our case, we only have one: `stash@{0}`). To retrieve the changes, run:

```
git stash apply stash@{0}
```

If you run `git status` now, you should see `filetostash.txt` is being tracked again.

Chapter 3 Advanced Git

With the material you covered in Chapter 2, you now have enough knowledge to a) use Git in the workplace and b) explore the wider Git ecosystem. In this Chapter, we'll cover the topics that will set you up to be a poweruser.

3.1 Submodules

3.2 When do you rebase and when do you merge

3.3 Cherrypicking