Git commands Memento

Courtines Elana - @irinacake on GitHub

Introduction :

Git itself is nothing more than a tool to manage files (*Distributed Version Control System - DVCS*). Instead of managing "files" themselves, it manages modifications done to a repository (adding/modifying lines, changing names, etc). The use of remote repositories requires the creation of such remote first, before creating any local folder. This can be done with websites such as https://github.com/ or by setting up a private git server on your own computer (feel free to google).

Setup :

- `git config --global -l` : check your current global config ;
- `git config --global user.name "First Last"` : sets up your name (NOT your "pseudo") globally. It will appear when making commits for example ;
- `git config --global user.email "email"` : sets up your email ;
- `git config --global push.default upstream` ;
- `git config --global color.ui auto` : allow git to make use of colors during interactions. While it is not required, it makes it easier to understand the results of commands (for example, `git status` will color file names in green/red so that you can easily see what has been added/deleted)
- `git config --global merge.log true` : affects the content of logs by making them more explicit ;
- `git config --global rebase.stat true` : *whether to show a diffstat of what changed upstream since the last rebase. False by default*;
- `git config --global core.editor "editor"` : set the default editor that is used for various needs (commit messages for example). You may want to google what to type instead of "editor" depending on the one you want.

Repository setup :

- `git init` : initialize the current folder to be a **local** repository. You can also add a folder name as an argument to create and initialize the local repository in there ;
- `git clone path/url` : initialize the current folder by cloning an existing local repository (located at `path`) or a **remote** repository (by providing its `url`, please note that :
  - `git clone https://github.com/YOUR-USERNAME/YOUR-REPOSITORY` : uses HTTPS which requires a personal token (which can be retrieved by accessing your github account from your browser) ;
  - `git clone git@github.com:YOUR-USERNAME/YOUR-REPOSITORY` : uses SSH which requires you to generate a SSH key on your computer and to associate it to your github account. Here is how you would do that :
    - `ssh-keygen -t rsa -b 4096` : generates a ssh key. It will prompt where to store the key, which is `~/.ssh/id_rsa` by default. Simply press enter to accept or store it elsewhere if you want ;
    - add this newly generated key to your github account at https://github.com/settings/keys. To do that, you need to copy paste the content of the `~/.ssh/id_rsa.pub` file (the public one with the `.pub` extension, not the other one). Keys are computer-specific. If you want another computer to access your github account with ssh, you will need to generate another key on that computer and add it to your computer account again ;
    - `ssh -T git@github.com` to check that the ssh connection is working. Note : github will not *accept* the connection, it will only tell you whether it is working or not. A typical **positive** answer would be this :

```
$ ssh -T git@github.com
Enter passphrase for key '/Users/fraise/.ssh/id_rsa':
Hi irinacake! You've successfully authenticated, but GitHub does
not provide shell access.
```

Local Repository management

Git tracks what happens in a repository, as such, it will detect any change made to the files that are *tracked* : addition/deletion/modification. Git does not consider folders, and will simply ignore

them unless specified otherwise. You can check the status of a repository at any time by being anywhere within the repository and typing `git status` (or `gitk` if you prefer a more visual tool...).

- `git add` : will *add* to the "tracked files list" the specified files or add to the list of changes (called the index) whatever has changed for already tracked files. New files are not automatically tracked. Notables options are :
  - `git add .` : adds every file in the current *tree* ;
  - `git add -A` `git add --all` : adds every file in the repository.
- `git reset files` : does the opposite of `git add` (entirely resets the index if no files are specified) ;
- `git commit` : commits all the added changes if no option is specified. To perform a commit, a message must be added. You can add the message by typing it when the editor opens (save then quit, otherwise the commit will be aborted), or you can use the `git commit -m "commit message"` to skip opening the editor. Note : commits can be done in an interactive way by specifying `git commit --interactive` (feel free to google) ;
- `git commid --amend` : opens up the editor to modify the last commit, which only implies modifying the commit message if no other arguments are given ;
- `git tag` : creates a "tag" that is used to refer to a commit. Commits are not necessarily easily identifyable. The only way to *refer to a commit* is by fetching its <SHA> with the command `git log`, or by creating a tag, a "pointer" to that commit. A complete command would be : `git tag -a "tagname" <commit-SHA> -m "Message"`, which creates a tag name `"tagname"` refering to the commit with the given <SHA> and with a small message. If no <SHA> is specified, the created tag will refer to the current commit (the last commit of the current branch) ;
- `git diff <commit-ref> <commit-ref>` : much like the `diff` shell command, shows the differences between the two given commits ;
- `git reset --[mode] <commit-ref>` : similarly to resetting added files, this command can undo a specified commit entirely, depending on the specified mode. See Git - git-reset Documentation for more details (be careful, data **can** be lost with `git reset --hard` ...)
- `git revert <commit-ref>` : reverts the changes of a specified commit. Note : this command does **not** delete the specified commit, instead, it creates a new commit that reverts what has been done in the specified commit ;

- `git branch` : *(I will not be explaining how branches work in this document, feel free to google how it works)* if no arguments are given, lists all the existing branches. Branches can be deleted by specifying `git branch -d branchname`. Notable arguments : `git branch -a` to see local and remote branches, `git branch -vv` to see more details on branches status ;

- `git checkout filename` : ⚠️ **this command reverts all changes to the files that are not yet in the index**. Be careful, data can be lost. While this command can cause loss of data, it can also restore files that have been deleted by reverting the deletion (until it has been committed) ;

- `git checkout branchname` : switches to the specified branch. It is also possible to **create and switch** to a new branch by typing `git checkout -b newbranchname` ;

- `git checkout <commit-ref>` : will *detach* the HEAD and have it point the specified commit. You can find more information about the *Detached HEAD state* at Git - git-checkout Documentation (scroll down to the Detached HEAD section). A short explanation would be : this command allows you to "temporarily go back in time" to a previous commit, to *make some changes that do not affect the present*, but that you can still save by creating a new branch where you commit these changes (`git checkout -b "newbranch"` while in the *Detached HEAD state*) ;

- `git merge branchname` : will merge all the commits from the specified branch to the current branch. By default, *if it is possible to find a common ancestor*, the merge will happen with a fastforward : all the commits that were made on the specified branch will be directly added to the current branch. You can force the merge to happen without a fastforward (`git merge branchname --no-ff`), which will result in the creation of a new "complete" commit in the current branch (commit that contains all the changes made in the specified branch). For more details and explanations, see What is git fast-forwarding? - Stack Overflow, and Git fast-forward VS no fast-forward merge - Stack Overflow (*the understanding of this concept is capital*). It should also be noted that, sometimes, conflicts can happen (different commits affecting the same files). When that is the case, it is necessary to resolve them manually during the merge ;

- `git cherry-pick <commit-ref>` : allows to merge a specific commit to the current branch. This command can be very useful when creating "hotfixes" to an application. It allows you to specify which "hotfix commit" exactly you want to merge to a public release.

- `git rebase` : is an interactive way to cherry-picking multiple commit to the current branch. An editor will open where you will be able to manage every commit invidually : select which one to take into account, which one to ignore, which one to combine (squash), etc... ;

Remote Repository commads :

- `git push origin remotebranch` : *push* all the *unpushed* commit **of the current branch** to `remotebranch` in the remote repository. By default, newly created local branches do not have a corresponding remote branch. As such, pushing a branch requires that you specify the remote branch name. Associating a local branch with a remote branch is done by entering `git push --set-upstream origin remotebranchname` (you can also use the shortcut `-u` instead of `--set-upstream`) while in the local branch you want to associate ;

- `git fetch origin remotebranch` : *fetches* the changes that have been made to the `remotebranch` (as well as any tag that points to it). This command only fetches them, which allows you to see the changes locally without applying them. To apply them, you can then either use `git merge` or `git rebase` (with not argument). Note : if no arguments are given, it will fetch the origin by default, or the associated remote branch if the current branch has one ;

- `git pull origin remotebranch` : combines both `git fetch` and `git merge` if there are no conflicts. Note : same note as for `git fetch` regarding local-remote branches associations.