Deddy Tandean Follow
Mar 8, 2021 · 10 min read ★ · Listen

16 Git Commands That I Use Almost on a Daily Basis

Terminal commands that can help accelerate your workflow

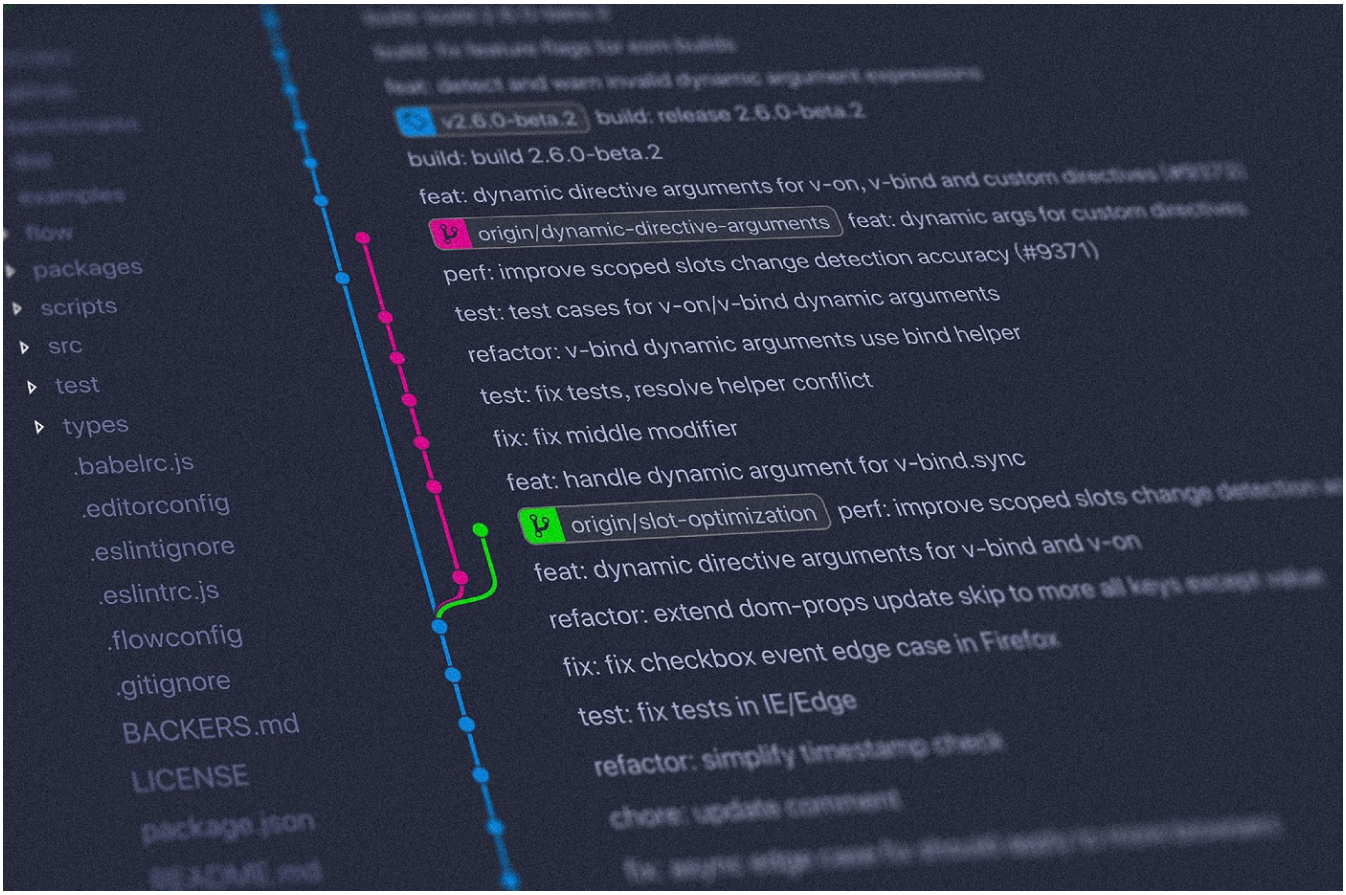

Photo by Yancy Min on Unsplash.

Git might be the most widely used version control system today. Although there are alternatives like Bitbucket, you would have at least heard about Git whenever projects

These Git commands are usually used with the terminal whereby one can type certain commands to do things. If you are a GUI kind of person, this might not be interesting to you. But if you are interested in using the terminal to type a bunch of commands like Hackerman and be perceived as cool, you can continue reading.

Jokes aside, I have tried to compile an overall list with a quick explanation of each one for you so that it is easier for you to refer to. There are a lot of Git commands with different permutations (i.e. when each command is applied with different flags). However, there is a subset that people will usually use every time.

So, let's get started!

Note: You can always get more information by running `git [command]- — help` *(e.g.* `git init --help`*) in your terminal.*

Here is the list of commands that I typically use or at least encounter:

1. `git init`
2. `git clone`
3. `git branch`
4. `git checkout`
5. `git status`
6. `git fetch`
7. `git pull`
8. `git add`
9. `git commit`
10. `git push`
11. `git remote`



14. `git stash`
15. `git reset`
16. `git config`

1. git init

This is to initialise an empty Git repository. Basically, it is to create an empty Git repository in your intended directory. A `.git` directory with its subdirectories is generated.

Once this is done, you can start using Git commands in the directory.

```
$ git init
```

2. git clone

This is to clone or download source code from a remote repository (like Github, etc.) into your directory.

It basically makes an identical copy of the latest version of a project in the remote repository and saves it to your computer.

I typically use the link to clone, which is more familiar.

To clone the project repository into a main branch, run `$ git clone [YOUR_HTTPS_LINK].`

For example:

To clone the link into a new branch, you can run the following command:

`$ git clone [YOUR_HTTPS_LINK] -b [new-branch-name] .`

For example:

```
$ git clone https://github.com/randomname/awesome-project-example.git -b my-branch
```

3. git branch

This is to list all your available Git branches locally.

It is a highly used Git command. I use this every time — especially when I have to work with different stuff located in different directories, and each of them has different branches.

Branches are really important for collaboration purposes and also to help maintain the main branch and prevent it from being messed up with unfinished or untested code for example.

You can use the `git branch` command for creating, listing, and deleting branches.

To create a new branch (locally), use `$ git branch [new-branch-name] .`

For example:

```
$ git branch new-branch
```

To make sure that the remote repository has this branch too, push it by running `$ git push -u [remote-origin] [your-new-branch] .`



To list all your local branches and check which branch you are on now, run `$ git branch` or `$ git branch --list`.

The branch that you are on currently will usually be highlighted.

To delete your local branch, run `$ git branch -d [your-unwanted-branch]`.

For example:

```
$ git branch -d myBranch
```

4. git checkout

This is also one of the most used commands — especially when working with different branches.

This is usually used when you want to switch to another branch. You can also use it for checking out files and commits.

To switch to another branch, run `$ git checkout [another-branch-name]`.

For example:

```
$ git checkout myBranch
```

To create a new branch and then switch to that branch, run `$ git checkout -b [new-branch]`.

For example:

```
$ git checkout -b myBranch
```

The command above is a shortcut to first create a branch with the `-b` flag and then check it out to switch over.

To create a new branch from a specific remote branch, you can run `$ git checkout -b [your-new-branch] [origin-remote-branch]`.

For example:

```
$ git checkout -b dev "origin/dev"
```

There are criteria with which you can then switch over to the other branch:

- The changes in your current branch must be committed or stashed before you switch.
- The branch you want to check out should exist in your local.

Another use is if you changed a file in your branch and would like to revert to the original version. Run `$ git checkout [YOUR_FILE_PATH]`.

Or I typically just run `$ git checkout [YOUR_FILE_PATH] --`.

5. git status

This gives us all the necessary information about the current branch.

Information like:

1. Whether the current branch is up to date.
2. Whether there is anything to commit, push, or pull.
3. Whether there are files staged, unstaged, or untracked.



The process typically needs the files with changes (created, modified, or deleted) to be staged first and then you can commit it by adding all the files into the stage. After committing, you can push the files out to the remote repository.

6. git fetch

To get updates for the project from your remote repository, run `$ git fetch`.

This only fetches the updates for your available branches that are listed locally and also remotely. To fetch all the updates for all branches located remotely, you can run `$ git fetch --all`.

Take note that this does not apply the changes immediately to your local files.

7. git pull

I use this command more often than the `git fetch` command.

This command is also to get updates from the remote repository, except it is like a combination of `git fetch` and `git merge`. This means that after fetching the updates, it immediately applies the changes to your local files.

```
$ git pull
```

You can also run this command: `$ git pull [remote]`.

For example:

```
$ git pull “origin/master”
```



For example:

```
$ git pull "origin/dev" dev
```

I suggest always doing a `git pull` before you push your changes up, as conflicts might happen.

Therefore, it is always better to pull the changes to your local machine and solve any conflicts first. Then you can push it up to have a smooth merging process later on. This also prevents any troubles that you might cause your teammates.

8. git add

When changes are done, be it creation, deletion, or modification, they are done locally and hence not reflected remotely yet.

Usually, after the changes are done, you would need to stage the files up where you can then run the `git commit` command to really have your changed files be committed to the changes — or be saved in a sense. Then you do a `git push` to push to the remote repository for the changes to be reflected remotely as well.

Hence, the first step is to put your files in the stage first. How? By running `$ git add .`

This command is to add all your files to the stage. Or, you can also do:

```
$ git add -A
```

To add an individual file, simply run `$ git add [FILE_PATH]`.

This is probably the most used command. Every time you want to save your changes, you have to use this. When you want to push your changes, you have to use this command first. Even if you don't intend to push your changes, you would still use this command.

Sometimes, after solving an issue or finishing a task, you would commit it first to ensure that the changes are locked locally by setting a checkpoint. It is pretty much like a staple.

There are two ways of doing this:

1. Committing with a short message: `$ git commit -m "commit message".`
2. Committing with a longer message: `$ git commit.`

This command will open up an editor for you to then edit your commit message.

In case you would like to edit your commit message, there are also two ways to do so:

1. Editing the most recent commit message the short way: `$ git commit --amend -m "new commit message".`
2. Editing the most recent commit message the longer way: `$ git commit --amend.`

This will open up an editor for you to edit the commit message.

If you are changing the commit message of a commit that has been pushed, you would need to force re-push again by running `$ git push [remote-branch] [local-branch] --force` or `$ git push [remote-branch] [local-branch] -f.`

Be cautious when you are force-pushing from your local to the remote!

10. git push

This command is run after committing your changes to send your changes up to the



Or, you can run this to be more specific on which branch:

```
$ git push [remote-branch] [local-branch]
```

For example:

```
$ git push "origin/dev" dev
```

However, If you have not tracked your branch because it is newly created, you can run these instead: `$ git push --set-upstream [remote-branch] [your-new-branch]` or `$ git push -u [remote-branch] [your-new-branch]`.

11. git remote

To show all the remote Git URLs available to your local machine, run `$ git remote`.

However, it usually does not show the URLs — only the names or aliases to the URLs. So, in order to show everything, including the URLs associated with the names, run `$ git remote --verbose` or `$ git remote -v`.

If you want to see the full output and you are on a network that can reach the remote repo where the origin is, run `$ git remote show origin`.

When you clone a repository, there usually is an origin remote being tracked already. However, if you would like to add a new one to point your local repository to track a remote repository, you can run `$ git remote add [name-alias-for-url] [remote-git-url]`.

If you want to delete the remote URL for some reason, you can run `$ git remote -rm`



```
$ git remote -rm origin
```

When cloning a repository (from GitHub or any source repository for that matter), the default name for the source of the clone usually is `origin`.

12. git merge

This is to merge the branch with the parent branch or master branch (whichever you are using for the main one). Usually, it is done when you have completed development and everything works fine locally, and the next step is to do other kinds of testing like integration testing or something else.

You must ensure that you are in your current branch that is changing. Then you run `$ git merge [parent-branch]`.

What happens is that your current branch or feature branch is merged into the parent branch, which can be `master` or `dev` for further testing.

13. git log

This is to see the commit history of your current branch. It shows the history information from the latest one. You can see the author, date, merge code, and commit message.

```
$ git log
```

14. git stash



It literally means what its name indicates: It stashes away or stores your changes. Then you can proceed with something else.

Sometimes, you might want to do this when there are changes that you don't want to commit yet, but you need to check out another branch for some reason, yet you cannot check out without committing or pushing your changes.

This command can come in handy.

You can use this to check your stashed changes:

```
$ git stash list
```

You can then pop it back to your working branch from the stash by running `$ git stash pop`.

15. git reset

This is to reset the current `HEAD` or changes of your local branch to a specific state.

To reset your changes, you can run `$ git reset [FILE_PATH]`.

It also comes with two options in which you can reset:

1. Hard: The changes are thrown away or deleted (`$ git reset --hard`).
2. Soft: The changes are staged instead (`$ git reset --soft`).

16. git config

I usually use this to set up my user email and username configuration for my Git



```
$ git config --global user.email yourEmailAddress@example.com
```

Conclusion

This might seem overwhelming at first — especially when you've just started your journey and see this long list of different command options. However, as you get involved in more projects that use Git frequently, you will gradually start noticing how you use some of the commands over and over again every day. And when you start to list them out, you will notice how you have used a lot of commands yourself. Eventually, you'll know these like the back of your hand.

In the end, always remember to do these things:

1. Fetch before you want to push your changes.
2. Then pull the changes if there are any from the remote.
3. Then add your files.
4. Commit your files.
5. Push your changes up to the remote.

Remember to solve any conflicts if there are any. Don't stubbornly push up your changes without pulling changes from the remote repository. This might cause issues not only for your own project but among your team members.

A clear commit message will help your projects in the long term, and I wish you the best on your exciting journey!

That's all for today's list. If you have other common Git commands that were not discussed in this article, feel free to share them in the comments section below! We would all love to learn together.

Thanks for reading.