Git

Marcel Tomàs Bernal

November 20, 2025

1 Setup

Installation. Install git in your machine. GitLens for VSCode is helpful.

```
username@ubuntu:~$ sudo apt-get install git -y
```

Connect to GitHub account. If already existing, look for an SSH key.

```
username@ubuntu:~$ ls -al ~/.ssh
```

If you don't have one, generate an SSH key.

```
username@ubuntu:~$ ssh-keygen -t ed25519 -C "your_email@example.com"
```

Press enter for default configuration, the key is saved in `~/.ssh/id_ed25519`. Add it to the SSH agent.

```
username@ubuntu:~$ eval "$(ssh-agent -s)"
username@ubuntu:~$ ssh-add ~/.ssh/id_ed25519
```

Copy the public key from this.

```
username@ubuntu:~$ cat ~/.ssh/id_ed25519.pub
```

Then go to GitHub → Settings → SSH and GPG keys → New SSH key, and paste it there. Finally, test the connection with this command.

```
username@ubuntu:~$ ssh -T git@github.com
```

2 Basic workflow

Initializing git repo. Git repository means git is tracking the files. Check your git version.

```
username@ubuntu:~$ git --version
```

Initialize repo (once per project). Adds hidden .git folder to keep track of all files and sub-folders. (Use `ls -la` to see hidden files). Run `git status` to see if the repo is being tracked.

```
username@ubuntu:~$ git init
username@ubuntu:~$ git status
```

A commit is like a checkpoint, allows you to compare changes between commits.
Working Dir → git add → Staging area → git commit → Git Repo → git push → GitHub
Run this to add files to the staging area (to be tracked). You can add all the files with . or individual files.

```
username@ubuntu:~$ git add file1 file2 || git add .
```

Running status shows untracked files (not added yet) and tracked files with "changes to be commited". Run this command to commit. You must add a commit message, if you don't, your text editor will open (likely vim, press I, write your commit message, press ESC and type :wq to save it).

```
username@ubuntu:~$ git commit -m "Commit message"
```

See log with all the commits (use `--oneline` for a summarized version):

```
username@ubuntu:~$ git log
username@ubuntu:~$ git log --oneline
username@ubuntu:~$ git log --graph --oneline --decorate
```

Git configuration. Run to change global configuration in your system.

```
username@ubuntu:~$ git config --global user.name "First Last"
username@ubuntu:~$ git config --global user.email username@gmail.com
```

To see where the .gitconfig is stored and its contents, run this.

```
username@ubuntu:~$ git config --list --show-origin --show-scope
```

Change default editor (sets default editor to vscode).

```
username@ubuntu:~$ git config --global core.editor "code --wait"
```

Modify the **.gitignore** file to specify which files you don't want git to track. You can use a gitignore generator.

Commits Each commit has a hash (the id), a message, information and a "parent" (except the first commit which has a Null parent). The commit depends on the previous commit (to track the changes done between commits). Make atomic commits: one task at a time (one feature, one component, one fix). Write the message in present tense, imperative.

3 Branching

Branches are "timelines" of git commits ("checkpoints"). Each node (commit) points to another node (the "parent" or previous commit). You can have multiple branches that can be understood as different "paths" taken.

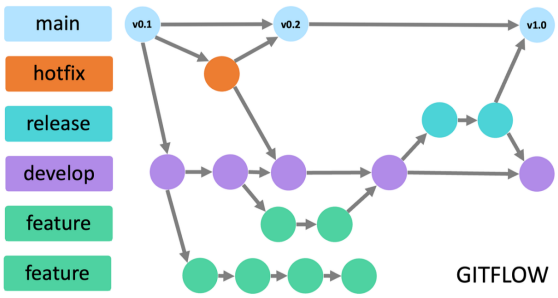

Figure 1: Example of a standard Git branching strategy.

This allows you to work in your branch ("timeline") without affecting other people's branches. By default git creates a master branch by default (usually renamed to main). To see the branch, we run this (current branch has a pointer *):

```
username@ubuntu:~$ git branch
```

To create a new branch on the current commit (in the HEAD), we run this command:

```
username@ubuntu:~$ git branch "branch name"
```

To switch to an existing branch, we run (**always commit before switching to an existing branch**):

```
username@ubuntu:~$ git checkout "branch name"
username@ubuntu:~$ git switch "branch name"
```

If we make work in a given branch and commit it, it will add an extra "node" (commit) in that branch. If we move back to another branch, we won't see those changes, and we can make changes in that branch that won't affect the other one (until we eventually merge branches). You can create and move to the new branch directly with the following.

```
username@ubuntu:~$ git switch -c "branch name"
username@ubuntu:~$ git checkout -b "branch name"
```

Head. Points to where your branch is currently at. It doesn't necessarily point to the end of the branch. Go to the .git folder and check out HEAD file to know where the head is pointing towards. Some more commands:

```
username@ubuntu:~$ git checkout <hash>  # Detach head: creates new "experimental" branch
username@ubuntu:~$ git switch master    # Re-attach head (discards "experimental" changes)
username@ubuntu:~$ git switch -c "new branch" # Create a new branch retaining the changes.
username@ubuntu:~$ git switch -         # Undoes the previous operation
username@ubuntu:~$ git checkout HEAD~2  # Look at 2 commit prior
username@ubuntu:~$ git restore filename # Get back to last commit version
```

Merging the branches. If you keep the main branch unchanged, work on a secondary branch, and then merge the secondary branch directly to the main branch, it's a **fast forward merge**. To merge a branch into the main branch, first switch to that branch, and then merge:

```
username@ubuntu:~$ git switch master
username@ubuntu:~$ git merge "branch name"
```

If we check the log in the main branch, the commits from the secondary branch will show up. After this, the secondary branch has served its purpose, and you can delete it with this command (it won't delete the commit history, it is now part of the branch you merged the secondary branch into).

```
username@ubuntu:~$ git branch -d "branch name"
```

In a **non fast forward merge** your alternative branch is working, your main branch is also working and eventually you want to merge them. Git tries best to solve **conflicts**. After trying to merge, it will list you the files with conflicts, and the files that have been modified by the two branches (with conflicts) will have something like:

```
<<<<<HEAD
line 1   # Branch you're currently on
line 2
=====
line 3   # Conflict that came from another branch
>>>>>bugfix
```

Keep whatever you want, remove markers and save. Usually you'll have to comment the changes with your coworker. After resolving the conflict, commit the changes.

3.1 Git diff

Informative command, compare same file in working stage and staging stage.

```
username@ubuntu:~$ git diff
```

To see the differences between different commits (you can get the ids from `commit log --oneline`):

```
username@ubuntu:~$ git diff "commit1_id" "commit2_id"
username@ubuntu:~$ git diff "commit1_id".."commit2_id"
```

You can do the same between branches:

```
username@ubuntu:~$ git diff "branch1".."branch2"
```

If there are no differences, it doesn't return anything. If there are differences, we get something like this (File a is the version you put first, File b the one you put later, by default they are staging and working):

```
a/file "staging" b/file "working" # Same file over time
--- a/file                        # Indicates changes in file
+++ b/file                        # - changes in a, + changes in b
@@ -1 +1,2 @@                     # Changes
Preview of the changes
```

If you swap the order of branches or commits in the `git diff` command, the + and - indicators will be swapped.

3.2 Stashing

Imagine we create a repo, work and commit on the main. Then, we switch to branch1 and do some work. Imagine you don't finish work in branch1, and need to make a change in another branch. Conflicting changes do not allow you to switch branch without commits. In this situation, `git stash` "stashes" your changes in branch1 and allows you to switch branches. Once you finish, you can go back to branch1. To bring back the stashed changes and keep working on them, you need to run `git stash pop` (applies the changes and removes the stash at once).

```
username@ubuntu:~$ git stash
username@ubuntu:~$ git stash pop "optionally add stash id"
```

Stash is not limited to a branch. You can pop the changes into a different branch, which can be a bit messy. You can use the following commands to list all stashes, and to apply only a specific stash (`apply` only applies the changes, doesn't remove the stash):

```
username@ubuntu:~$ git stash list
username@ubuntu:~$ git stash apply stash@{0}
```

Stash is meant for very brief situations and should be used carefully. For example, (1) you make some changes in a branch and want to check a different branch to copy some code, so (2) you stash your changes and switch branch to copy the code, then (3) go back to the original branch and pop the stash and continue working.

4 Rewriting

Undo last commit (`--soft` to keep changes, `--hard` to discard changes):

```
username@ubuntu:~$ git reset --soft HEAD~1
username@ubuntu:~$ git reset --hard HEAD~1
```

Unstage a file:

```
username@ubuntu:~$ git restore --staged "file"
```

Discard changes:

```
username@ubuntu:~$ git restore "file"
```

4.1 Git rebase

Alternative way of merging and clean up tool (rewrites the history). **NEVER** run this command in the main or master branch. It is meant to be ran from the **side branch**.

```
username@ubuntu:~$ git rebase master
```

This takes the side branch and "replants" it into the master (the branch is gone and its placed at the end of the master branch, the "merge" commits are gone). If there are any conflicts when rebasing, you need to resolve the conflicts, add the modified files with `git add` and run

```
username@ubuntu:~$ git rebase --continue
username@ubuntu:~$ git rebase --abort # Stops the rebase
username@ubuntu:~$ git rebase --skip   # Skips the conflicting commit
```

NEVER rebase commits you have shared with other people or if you're pushing to GitHub.

5 Remotes

Github documentation: https://docs.github.com/en/get-started. To generate an SSH key and connect to GitHub, see Section 1. You can create a repository in GitHub. Rename the master branch to main:

```
username@ubuntu:~$ git branch -M main
```

To check if you have a remote repository set up, run this command:

```
username@ubuntu:~$ git remote -v
```

To add a remote repository

```
username@ubuntu:~$ git remote add "name" https://guthub.com/yourname/reponame.git
```

You can remove or rename the remote repository with:

```
username@ubuntu:~$ git remote rename "old name" "new name"
username@ubuntu:~$ git remote remove "name"
```

Usually the name is set to origin. To push a branch to the remote repository, run this:

```
username@ubuntu:~$ git push <remote> <branch>
username@ubuntu:~$ git push origin main
```

Using `-u` sets up an upstream (linking the remote and the branch) that will allow you to push with `git push` directly (if you push from the same branch, other branches won't have the same upstream).

```
username@ubuntu:~$ git push -u origin main
username@ubuntu:~$ git push
```

5.1 Cloning, fetching and pulling

You can clone a repository on your current directory with

```
username@ubuntu:~$ git clone https://guthub.com/username/reponame.git
```

If you are actively working with a remote repository, you can run these to get the info from the repo:

```
username@ubuntu:~$ git fetch https://guthub.com/username/reponame.git
username@ubuntu:~$ git pull https://guthub.com/username/reponame.git
```

`git fetch` puts information directly on your local repository not modifying your current work, while `git pull` puts information on your working area directly.

```
username@ubuntu:~$ git pull = git fetch + git merge
username@ubuntu:~$ git pull origin main # Changes will be merged to main.
```