# Git and GitHub Essentials

# Types of Version Control

1. **LCVS**: Local Version Control System: this involves the simple process of copying files to create different versions. It is simple but resource intensive.
2. **CVCS**: Centralized Version Control System: this involves storing all files on a central server computer. It provides easy access, but it has the inevitable risk of all data being destroyed.
3. **DVCS**: De-centralized Version Control System: this would be something like Git.

# Git History

Git is a version control system and is very popular around the world. A version control system is essentially a software that tracks and manages changes to files over time.

Version control systems allow things like revisiting earlier versions of files, comparing changes made between different versions and files, undoing changes, sharing changes with other people, etc.

There are other version control systems like Subversion, CVS, Perforce, ClearCase, and Mercurial, but they are not nearly as popular as Git.

In 2005, while Linus Torvalds, a legendary software engineer and creator and main developer behind Linux and Git, was working on Linux, he became frustrated with the available version control systems. The existing tools were slow, closed-source, and usually paid. Linus and his team was working with BitKeeper at that time when the company suddenly decided not to allow Linux to use its system for free anymore.

Linus wanted a free version control system that had features he desired, and at the time, there were only paid systems that had those features. He wanted to make something fast and open-source. So he spent most of April 2005 just writing the code to make Git himself. He wrote the most basic features of Git in just a few days, and after a month or two, left the developing job for other developers.

Git means ‘unpleasant person’ in English. Torvalds said: “I’m an egotistical bastard, and I name all my projects after myself. First ‘Linux’ and not ‘git’.

# Git Behind the scenes

When we look into the hidden ‘.git’ folder in a Git repo, we see a list of files and folders: HEAD, config, description, hooks, info, objects, and refs.

Let’s now take a look at each.

## File: config

The config file is for configurations related to a single Git repo. In addition to configuring global settings like our name and email across all Git repos, we can also configure things per repo using the config file.

In order to add a configuration to the local config file we can use:

git config --local <configuration>

For instance, if we want to configure a local username and email:

git config --local user.name “omid armat”

git config --local user.email [omid@gmail.com](mailto:omid@gmail.com)

This will insert these configuration into the config file of the current repo.

## File: HEAD

Head is just a text file that keeps track of where HEAD points.

If it contains **refs/heads/master**, this means that HEAD is pointing to the master branch.

In **detached HEAD**, the HEAD file contains a commit hash instead of a branch reference.

## Folder: refs

Basically, refs, which stands for references, contains all the pointers (branch pointers or tags) of a repo. Inside the refs directory, we can find 3 other directories: heads, remotes, tags. Remember that when you have just created an empty repo, you don’t have the ‘remotes’ folder in refs.

* **refs/heads** contains one file per branch in a repo. Each file is named after a branch and contains the hash of the commit at the tip of the branch. For instance, refs/heads/master contains the commit hash of the last commit on the master branch.
* **refs/tags** contains one file for each tag in the repo.
* **refs/remotes** contains a folder for each remote. Each folder contains a file per branch on that remote repo, representing the last commit at the tip of those branches.

## Folder: objects

The objects folder is really the core of Git. This is where all the backups of our files and all the content is stored. All of our commits live here. So it basically contains all the repo files. Files are all **compressed and encrypted**.

Inside the objects folder, we have other folders including the ‘info’ and ‘pack’ folders. If there are any commits in the repo, there will be other folders. So the number of these folders depends on the commits that are made in the repo. The name for each folder actually comes from the first 2 characters of the file name in the folder. The file name is a hash. The file contains an encrypted content, which is probably not readable in our text editors. This is how Git stores full snapshots of files.

Inside the objects folder, there are 4 different types of basic Git objects: commit, tree, blob, and annotated tag. These are the Git’s core and they are all hashed using the SHA-1 algorithm.

### Git database

Git is a key-value data store. We can store any kind of data into a Git repo, and Git will hand us back a unique key we can later use to retrieve that content. These keys that we get back are SHA-1 checksums.

For instance, whenever we commit a change, we are actually storing a different version of our file in Git. Git generates a key for each file. These keys correspond to the data inside each file. But this is only one use case of hashes. Git uses these hashes all over the place.

We can actually directly use the hashing functionality of Git using the hash-object command. We give it a file with a certain content, and it will return a hash generated based on the content of that file. This hash will always have 40 hexadecimal digits.

git hash-object <file>

We can also use another variant of this command in order to pass a content directly into it, and not from a file.

echo “<any-content>” | git hash-object --stdin

Note that these commands will not store the content that we are hashing. This will just return the hash that Git would use if it were to store it. However, if we add the –w flag to the command, it will tell Git to store the content.

echo “<any-content>” | git hash-object --stdin -w

Now a hash is generated based on the content that we passed into the command. This hash is not from a commit, it is from a **blob**. The first two letters of this hash will be used to create and name a new folder in the objects folder, while the rest of the characters of this hash will be used to create and name a file inside the new folder. This file will contain the encrypted version of the content that we passed into the command.

In addition to storing data in this low-level way, we can also retrieve data that we stored, using the cat-file function. The –p flag actually tells Git to pretty-print content of the object based on the type of object it is.

git cat-file –p <object-hash>

This will basically find to which file is the hash related, then decrypts the file and returns its content to us.

### Objects: blobs

Git blobs (stands for Binray Large Object) are the object type Git uses to store the contents of files in a given repo. Blobs don’t even include the filenames of each file or any other data. They just store the contents of a file. Filenames are tracked using trees, which are another type of Git objects. As we know, each blob gets its own hash. It is similar to a commit hash, but it is a blob hash.

### Objects: trees

In a large project with hundreds of files in nested folders, when we tell Git to check out this commit or switch to that branch, it can change all of the files and folders in our working directory. How is it able to keep track of our file structure? This is where trees come in.

Trees are Git objects used to store the contents of a directory. So each tree represents a certain folder. This tree contains pointers that can refer to blobs and other trees.

Each entry in a tree contains the SHA-1 hash of a blob or tree, as well as the mode, type, and filename. If we have nested directories, we would have trees inside trees.

For instance, if we have one folder and inside that one folder there is one file, Git would represent this simple structure with one tree, and that tree would point to one blob for that file. So a tree does not store the contents of files.

### Objects: commits

Commit objects combine a tree object along with information about the context that led to the current tree. Commits store a reference to **parent commit**(s), the **author**, the **committer**, and the **commit message**.

Every commit has a hash. If it is an initial commit, it won’t have any parent commit. It would have a tree, an author, committer, and message. The tree is the most important part. The tree is the current content of the index, the staging area. So there is an actual index file containing basically everything that we have worked on, all our changes and everything staged and prepared to commit.

Now if we make another commit, so when we run ‘git commit’ command, another tree is generated. This tree will again reflect the contents of the index and then that is included in the newly created commit object. This commit object will also have a reference to its parent commit.

So every commit is tied to a tree, and that tree represents the structure of the application, and in turn, the blobs that contain all the data in the files.

## Folder: hooks

The folder contains shell scripts that trigger actions in response to specific events. These scripts help you automate your development lifecycle. Every Git repo has 12 sample scripts.

For instance, the ‘pre-commit.sample’ script executes when w a commit is submitted but before it is permitted. The ‘commit-msg.sample’ script executes after a commit message has been submitted.

## Folder: info

Holds additional information about the repository. One of the best-known files is the ‘exclude’ file. It decides which pattern will be ignored. To define the ignored files and folders, we use a file in our project called .gitignore.

# Git Bash Terminal

Here is a list of basic commands in terminal. Remember that these commands are executable in **Git Bash**.

## Listing and navigating in terminal

### Cmd: ls

short for ‘list’. This lists the content of the current directory.

### Cmd: ls –a

This lists all the content of the current directory, including the hidden files and folders. Hidden files will be listed with their names starting with a dot.

### Cmd: ls –l

This lists all the content of the current directory. In this list, directories start with the letter ‘d’, hidden files (link) start with ‘l’, and entries starting with ‘\_’ refer to files.

### Cmd: ls –la

This lists all the content of the current directory, including the hidden files and folders. Hidden files will be listed with their names starting with a dot.

### Cmd: ls –ltrh

This lists all the content of the current directory. ‘t’ sorts the list by time, ‘r’ reverses the sorting, and ‘h’ makes the list human-readable.

### Cmd: ls <folder name>

Shows the content of a specific folder existing in the current directory.

### Cmd: ls <folder>/<folder>

Shows the content of a specific folder existing in another folder that exists in the current directory.

### Cmd: ls ../

Lists the content of the parent folder.

### Cmd: start .

Opens file explorer on windows. On Mac it would be ‘open .’.

### Cmd: pwd

Stands for Print Working Directory. Prints the path to where the terminal is currently running.

### Cmd: cd <folder name>

Stands for Change Directory. Moves to the specified directory.

### Cmd: cd ..

Moves one level back in the path.

### Cmd: cp <file name> <destination path>

Copies a given file to a given destination path.

### Cmd: mv <file name> <destination path>

Moves a given file to a given destination path.

### Cmd: echo “<string>” > <file name>

Inserts a given string into a given file.

### Cmd: echo “<string>” >> <file name>

Appends a given string to an already existing string inside a given file.

### Cmd: cat <file name>

Prints the content of a text file.

### Cmd: code .

This will open VS Code right at the location of the terminal.

## Git configurations

### Cmd: git config --help

Opens a PDF file containing a Git manual for configuration commands.

### Cmd: git config --list

This will list all the configurations of a Git repo.

### Cmd: git config --global <configuration> “<value>”

This can implement a configuration to any Git repo that will be built after this point. For instance, configurations like ‘user.name’ and ‘user.email’ are usually set globally.

### Cmd: git config init.defaultbranch

Used to set a custom name for the master branch.

### Cmd: git config --global <configuration>

This will retrieve the value set for a given global configuration.

## File and folder creation and deletion

### Cmd: touch <file name>.<extension>

Creates a file with the specified name and extension in the current directory.

### Cmd: touch <file> <file> <file>

Creates multiple files with the specified names and extensions in the current directory.

### Cmd: touch <folder>/<folder>/<file>

Creates one file with the specified name and extension in the specified directory, not the current directory. Creating multiple files is not possible with this command.

### Cmd: mkdir <folder name>

Creates one folder with the specified name in the current directory.

**NOTE | if you want to put spaces in the folder name, you should put the name in quotes. However, it is not recommended to put spaces in folder names, since this will cause problems in terminal navigation. Try to use this format for naming folders: FolderName**

### Cmd: rm <file name>.<extension>

Stands for remove. This will remove a file with the specified name and extension from the current directory.

**NOTE | Remember that this command will not make the deleted file end up in the recycle been, it will totally be gone. However, if the deleted file was being tracked by Git, we would be able to retrieve it using one simple command.**

**NOTE | this command cannot be used to delete folders. For deleting directories with this command we can use 2 flags.**

### Cmd: rm <file> <file>

Removes multiple files with the specified names and extensions from the current directory.

### Cmd: rm –r <folder name>

Removes a folder. ‘r’ stands for recursive, and makes deletion also happen for all files and folders inside the target folder.

### Cmd: rm –rf <folder name>

In the flag, ‘r’ stands for recursive, and ‘f’ stands for force. This will also make the folder completely gone.

### Cmd: rm ./

Removes the current directory

### Cmd: rm / (NEVER USE)

Removes the root path.

# Git

## Git repo

Git repo is a Git workspace. Remember that installing Git on a machine does not mean that Git is active. We have to manually tell Git when we want to use it and in which directories we have to create repos. Every Git repo has its own history. Histories of different Git repos are not linked or connected in any way.

## Git <command> -h

This command will list all the options and flags available for a certain command.

## Git status

Using this command in the Git Bash in a specific directory:

git status

will respond back with the status of Git repo for that directory. If there is no Git repo in the directory, it will say so. If there is a Git repo in that directory, it will confirm and report the branch and the commits.

## Initializing a new repo

When we initialize a new repo in a directory, that directory will become the home of that repo. We do this by using this command in the Git Bash. This is a command that we run only one time per project.

git init

If we now try to inspect what is inside the current directory with an –a flag, we will see a .git folder. Inside this .git folder, there are a couple of folders and files. If we delete this .git directory, our git history of this specific project is gone. So the whole Git repo will be gone. To reinitialize a new Git repo here, we can use the command above again.

**NOTE | A Git repo will watch everything in the directory where it is based. That would be every file and every folder and all nested folders in this base directory. So if we run the git status command in any of the nested folders, we will still receive a report of the branches and commits. So never initialize a new Git repo in a Git repo, never run a git init command inside a Git repo. Always use git status first.**

## Git add and commit

Each checkpoint in time that we create on the progress of our project is called a commit. A Git repo would end up having tons of commits one after another. Each commit has a message attached to it, explaining briefly what feature or change that specific checkpoint holds in it.

It is important to know that we don’t have to commit all the changes implemented into a project together. We can group some of the changes in one commit, and group the other in another commit. So to actually make a commit, there 2 associated commands, first of which is used to select and group the changes, which is also called staging changes, because it sends the changes from the **Working directory** to the so-called **Staging area**. Then we would send the staged changes to the Repository itself, which is the .git folder.

### git add

If a create a new file in the root directory, also called Working directory (where we initialized the Git repo), and then run the git status command, we see that Git reports ‘Untracked files’. It lists the new files added to the project. Now to add these files to the staging area, we use this command:

git add <file1> <file2>

using the git status command again, the report will contain a list of files added to the staging area, and then another list of untracked files. We can add more files to the current staging area by using another git add command.

In order to add all modifications and new untracked files to the staging area, we use this command:

git add .

**NOTE | Git notices changes once new files are created or modified files are saved (Ctrl + S).**

**NOTE | Git add can also be used with two dots (git add ..) which will refer to files in the parent folder.**

git add ..

### git commit

This command will send the staged files to the repo. Whenever we commit files, Git expects a message describing or summarizing the changes included in the staged files. To commit our staged files along with a commit message we use this command:

git commit –m “<messaged>”

after committing, if we use the git status command again, we see that we have a branch report, in addition to a message saying that there is nothing in the staging area anymore, or basically, nothing to commit. So Git is tracking our existing files for any changes, they are no longer untracked. If we now make changes to these tracked files, they will appear in a list that represents ‘changes not staged for commit’. These files will be reported as **Modified**. Git doesn’t say what has been modified in the modified files at this point, but there is a way for us to see what exactly is changed in these files. However, if we add new files at this point, the new files will be reported as **Untracked** again.

**NOTE | If the commit command is used without the –m flag, a text editor will be opened asking for the commit message. This text editor would be Vim by default. We usually change this default to VS code using this command in the Git bash.**

git config --global core.editor "code --wait"

**In case this didn’t work, you may need to define the ‘code’ name representing VS code in the PATH of your system. This should be done in the command palette of VS code: install ‘code’ command in PATH.**

**NOTE | we can use one command for both adding and committing, but this will commit all modifications and untracked files:**

git commit –a –m “<message>”

**NOTE | There is also another variation of this command which only works for tracked files.**

git commit –am “<message>”

## Git log

This is very similar to git status command, it does not do anything, it just reports a log of commits for the current repo. Each reported commit includes the commit hash (similar to ID), the Author, the Date, and the commit message.

git log

in order not to see long commit messages in the result of git log command, we can use the oneline flag.

git log --oneline

This is actually the shorthand command for 2 commands:

git log --pretty=oneline --abbrev-commit

the ‘—abrev-commit’ command makes the commit hashes appear in their short format. Also, this command will only show the first line of the commit message. So according to a convention, the first line of a commit message should summarize the whole commit message.

## Amending commits

You may create a new commit and then realize that you forgot to include a relevant file, or you may find out that your commit message has a typo. There are a couple ways to fix these problems, but here we amend the commit. This allows us to amend just the latest commit.

So after making a commit, if you want to add another file to that commit:

git add <forgotten file>

then we should put it into the latest commit:

git commit --amend

This command will open the configured text editor (VS code) and let you edit the commit message you inserted previously.

## Ignoring files

Using a file called ‘.gitignore’ we can tell Git not to track certain files and folders.

* File names should be written along with their extensions.
* Folder names should end with a / character.
* In order to ignore any file with a specified extension, we can use \*.<format> syntax.

## Git branches

### HEAD

The HEAD is a pointer that refers to the current location in a repo. It points to a particular branch reference. Using an analogy of a book and its bookmarks, at any given point in time, a book can be open only to one of its bookmarks that shows where someone left off reading. Likewise, HEAD is our current location that we are viewing or checking out at a branch. So if we are in the master branch, HEAD refers to the latest commit of the master branch where we left off coding. Remember that the latest commit of a branch is called the **tip** of that branch.

### Creating, deleting and switching to branches

Remember that in a typical workflow, we usually use some predefined branch names such as master, develop, feature, test, and hotfix. As a recommended workflow, we might want to create a branch called ‘develop’ from the main/master branch. From this develop branch, we would want to create another branch called ‘feature’ where we develop a new feature, and then merge it back to the develop branch once we are kind of sure that the feature works fine. Then, we would also want to create a ‘test’ branch from the develop branch in order for a test team to actually test the developed feature. If everything is fine, we will then merge the develop branch into the main/master branch. Also keep in mind, that once we are fully done with developing a new feature and it is already merged into the main/master branch, we usually delete the feature branch.

The hotfix branch is usually used when some critical error has occurred in the main/master branch, and a very high priority has been set to correct the error. In this case, we create a hotfix branch from the main/master branch and do the development there until the error is gone, then we merge it back to the main/master branch.

#### Cmd: git branch

This command will report back the branches that we have in our Git repo. The active branch is marked with an \* sign.

#### Cmd: git branch <branch-name>

This command will create a new branch based upon the current HEAD. Branch names should not include spaces. However, this will just create the branch, and will not switch to it.

#### Cmd: git switch <branch-name>

This command will make Git now move to another branch that we created. We can also use git.checkout with a branch name.

**NOTE | we can create and switch to a branch in one go:**

git switch –c <branch-name>

#### Cmd: git switch -c <branch-name>

Used to create and switch to the branch at once.

#### Cmd: git checkout <branch-name>

This does the exact same thing as git switch does, but it can also restore working tree files.

#### Cmd: git checkout -b <branch-name>

Used to create and switch to the branch at once.

**NOTE | if we try to switch to another branch while our modifications to the current branch are not committed, we receive an error that warns us that our changes would be lost if we switch. It would also suggest us to commit or stash our changes, and then switch. It is important to note however, that this problem only occurs if our modifications cause conflicts between branches. For instance, if we modify a file that exists in both branches (current branch, and the one that we want to switch to), that would be a conflict and Git will warn us about it. But if we create a new file in the current branch, put something in it and then try to switch to another branch while not committing that new file, no error will occur, because this modification will not cause any conflicts. The newly created file only exists in this current branch and it cannot get into conflict with the other branch. The new file will just follow you along while being marked as untracked.**

#### Cmd: git branch –d <branch-name>

This command will delete a branch with a given name. It won’t work if we are currently in that specific branch. If the branch that is going to be deleted is not merged, Git will respond with an error. However, if we want to delete the unmerged branch by force we can use the uppercase D in the command’s flag.

git branch –D <branch-name>

This is actually a shorthand for:

git branch --delete --force

#### Cmd: git branch –m <new-branch-name>

To rename a branch you should switch to that branch first.

### Merging branches

A common workflow is to treat the master or the main branch as the source of truth or the most stable build of an application where you don’t do any experiments. In case you want to add a new feature to your application, you work on a feature branch, from where the new feature would eventually be incorporated back into the master branch.

There are two different situations when we merge a feature branch into a master branch. First, the master branch does not have any additional work in it. So the master branch stopped at some point, then the work was continued in the feature branch, and finally this branch was merged into the master branch. This is also called a fast-forward merge.

The second situation however, is a bit tricky. That is when some work has been done on the master branch while some other work has been done on the feature branch. The merging process in this case will be problematic and some additional works should be done compared to the fast-forward merge process.

#### Fast-forward merge

First we have to switch to the destination branch into which another branch is going to be merged. Then we use this command:

git merge <branch-name>

After merge is done, the merged branch still exists, and additional work can be done on it. This additional work will not be reflected on the destination branch.

#### Non-fast-forward merges

We use the same process and use the same command, but Git may not be able to do the merge automatically. It depends on the particulars of our commits. So for example, we edited a line in the master branch, and we edited the same line in the feature branch. Which edition will be persisted through the merge process? So again we have two situations. One is when Git can do the merge automatically, and another is when it can’t.

* **Merge commit:** this will automatically result in a new commit on the destination branch, prompting us for a commit message. This would be a commit that, for the first time, would have two parent commits.
* **Merge conflicts:** in this situation, Git responds with an error telling us that there are conflicts in the content. Conflicts in this situation should be resolved manually. After resolving the conflict in the content, we should also remove the conflict marks generated by Git. After resolving conflicts, if we run git status, Git will report the unmerged conflicting paths. So the merge seems to still be in progress, and we should now add our changes and make a commit.

## Git rebase

Rebasing is increasingly becoming a part of workflows or companies workflows and engineering teams. Merging and rebasing are two different ways of integrating changes from different branches. It is important to know when not to use git rebase.

There 2 different things that we can do with rebasing: 1) use instead of git merge, 2) clean up your own commits or clean up your git history.

### Rebase for merge

When we start a new feature branch, while we are working on this branch, the master branch might get updated by other members. Since we want our branch to be updated with the latest changes of the master branch, we merge our branch with the master branch using the git merge master command, which will make us a merge commit. Now imagine we should work on our branch for quite a while, then our branch would end up with lots of merge commits. These commits however, don’t actually say anything about the work that we have been doing on this branch. Note that all other members should follow this process, because everyone is working on their specific branch. Finally, when the feature branches are merged into the master branch, the master’s history would be filled with a bunch of useless merge commits. This is where rebasing can help us.

What happens with rebase is that we rewrite history and this is why it sometimes becomes problematic. With this command, we have git create new commits for us based on the original feature branch commits. This will create a linear structure of commits, with our feature branch commits added to the tip of the master branch. So it will basically rebase my feature branch commits at the tip of the master branch, but the point is that it rewrites our feature branch commits, it replays our work and make new commits. The commit hashes of our feature branch would be different now, because they are actually new commits made by git.

To use the rebase command, we should switch to the feature branch first, and then use:

git rebase master

**NOTE | rebasing can meet failure due to content conflicts. In this situation, rebasing will pause, we would have to manually resolve the conflicts and then run:**

git rebase --continue

#### When not to rebase

Never rebase commits that have been shared with others. If you have already pushed commits up to GitHub, do not rebase them unless you are positive no one on the team is using those commits. You do not want to rewrite any git history that other people have on their machine.

You can rebase commits that you have on your machine and other people don’t. You don’t want to rebase the master branch, because other people have that master branch.

### Rebase to clean up history

We can also use rebase to rewrite history, we can edit commits, we can change commit messages, and we can also change the contents of a commit. We can drop or delete a commit. We can even reorder commits.

Again, we don’t want to rewrite history on work that people already have. This is something we do before sharing our work with others. For instance, we might have some half-complete commits during our work, but when we want to share our work, we don’t want those commits to be there.

In this use case of rebase, we use the git rebase command, but we don’t specify a branch. Instead, we will rebase a series of commits onto the head that they are currently based on. So instead of rebasing onto the master branch, for example, we will rebase whatever branch we are on, onto where it currently is. In this command we should provide a range, determining how far we want to go back and recreate each one. We should also put -i flag in this command which stands for interactive.

git rebase –I HEAD~<range>

using this command will open our configured text editor. In there, we will have a list of commits that are within the range that we mentioned. For each commit, we have a series of options to perform:

* Pick: use the commit
* Reword: use the commit, but edit the commit message
* Edit: use commit, but stop for amending
* Fixup: use commit contents but meld it into previous commit and discard the commit message
* Squash: use commit, but meld into previous commit
* Drop: remove commit

**NOTE | when we do something on one commit, the hashes of all later commits are changed, because each commit takes into account the previous commit as its parent commit.**

**NOTE | if you want to edit only the latest commit, you can use the git commit amend command:**

git commit --amend

## Git diff

We use the git diff command to view changes between commits, branches, files, our working directory, and more. It doesn’t do anything to the repo, just like git status and git log.

This is usually used when we apply some changes to our code in different files, and then before adding them for committing, we want to see what we have modified throughout our project. Then we use git diff. The git diff command has a couple of variations.

In the output of the git diff command, the first line shows the two files that are being compared for changes. Usually they are the same file, one from the last commit (a), and one from the working directory (b). It could be different files though. Next line is the metadata about the files that are being compared, which is not important. Then there are two lines indicating how changes are represented by – and + in each file, and finally we see the changes. This part includes the changes you made to the file, along with a bit of code before and after that specific point. This part, at its first line, shows double @@ signs with 2 numbers inside. First number refers to file A, and the second refers to B. These numbers tell us how many lines have been modified in each file, and starting from which line. For instance -3.4 means from file A (-) 4 lines have been modified starting from line 3. This indication is usually confusing and we just look at the listed changes directly.

In the listed changes, lines with a – sign are the lines existing in file A, and line with a + sign are the lines existing in file B. Other lines with none of the – and + are unchanged lines. They exist in both files.

### Cmd: git diff

This shows us the changes that are made after the latest commit and that could be added to the staging area for the next commit. So it lists the changes in our working directory that are not staged yet.

**NOTE | Remember that for Git to notice what is staged and what is not, Git should be tracking the related files. If a file is untracked and you have made changes to it, git diff will not be able to list its changes. So if you create a new file and put something in it, using git diff will report empty, because this file is not un-staged for Git, it is untracked and Git cannot know if its changes are staged or not.**

### Cmd: git diff HEAD

This will list all changes in the working tree since the latest commit. This will include staged and un-staged changes. So this will list anything new in the working directory, since HEAD.

### Cmd: git diff --staged (also --cached)

This will list the changes between the staging area and our last commit. So it would be sensitive to only the changes that are staged.

### Cmd: git diff HEAD <file-name>

This lists the staged and un-staged changes of a specific file.

### Cmd: git diff --staged <file-name>

This lists the changes between the staging area and our last commit, limited to one specific file.

### Cmd: git diff <branch1>..<branch2>

This lists the changes between the tips of branch 1 and branch 2. Obviously this will include all the files in the branch tips. The order in which branches are typed in this command matters.

### Cmd: git diff <commit1-hash>..<commit2-hash>

To insert commit hashes we can use the short version listed by the oneline flag of git log.

## Git stash

Stashing is mostly used when we are in a certain branch, we have made changes and we want to switch to another branch while we still don’t want to commit the changes we have made in the current branch. By default, if we attempt to switch branches at this stage, we would face 2 situations: Git will make the changes come with you to the other branch, or the changes would cause conflicts and Git will not allow you to switch until you commit or stash changes. So stashing basically allows us to switch branches without having to commit changes, and it also prevents changes to come with us across different branches.

Git stash allows us to stash uncommitted changes and it also allows us to return to them later. The git stash command will take all uncommitted changes, including staged and un-staged, stashes them, meaning that it kind of hides them away so that they won’t come with us to other branches, and we can come back and retrieve them using the git stash pop command. Git stash pop is used at any point to remove the most recently stashed changes and re-apply them to any working directory that we are in. So we can retrieve stashed changes into another branch, but that might not be a very usual thing to do.

### Git stash (save)

This is the command:

git stash

which is a shorter command for

git stash save

This will stash all staged and un-staged changes. Note that after this, if we command git status, Git will report nothing to commit, and the working tree is clean. Now we can normally switch to other branches.

### Git stash pop

Back to the branch where stashed some changes, we can now use the stash pop command:

git stash pop

As a result, Git will report changes not staged for committing, and all changes are brought back to our files. Also remember that after this command, all stashed changes are removed from the stash.

**NOTE | git stash and git stash pop are probably the only stashing commands you are going to use throughout all your projects.**

### Git stash apply

This will do the same thing as git stash pop, with the difference that the popped changes will still exist in the stash. So it seems that a copy of the stashed changes is brought back to our files using this command, but we can still use the stash to apply those changes in multiple places.

**NOTE | Keep in mind that applying stashed changes to a branch other than the branch where the changes were originally stashed might bring up conflicts that need to be handled manually.**

### Multiple stashes

We can stash multiple changes on a branch. We can make a change, stash it, make another change, stash it, and so on. Of course, at some point we would want to restore some of the stashed changes.

git stash list

Using this command, Git will report back with a list of numbered stashed changes. In order to apply a certain stash we should refer to that using the number of that stash.

git stash apply stash@{<number>}

### Removing stashes

In order to just remove a stash without re-applying to our files we use the stash drop command followed by a reference to the stash that we want to be removed.

git stash drop stash@{<number>}

We can also clear the whole stash with one command.

git stash clear

## Git tags

The main idea behind Tags is that we can tag particular commits. We can label commits by creating a git tag, a reference to a moment in time. We can name these tags whatever we want, but typically tags are used to mark version releases for projects.

There are two different types of tags: lightweight tags which include just a name or a label, and annotated tags, which include additional information, including a tag message, the author’s name and email, date and other stuff. This is why annotate tags are generally preferred over lightweight tags.

### Semantic versioning

Semantic versioning is a protocol that dictates how version numbers are assigned and incremented. This versioning system includes 3 numbers with 2 dots between them. Each number indicates something different.

2.4.1

On the far left, we have major releases, then we have minor releases in the middle, and then patches. Typically, when you start developing some application or some library, the initial public facing release will be 1.0.0.

#### Patch releases

Once the initial release is done, small changes would be introduced as patch releases on the far right digit: 1.0.1

Patches do not contain new features or significant changes, definitely no breaking changes. It often just contain bug fixes and minor changes that don’t impact how people use the project. Patches are small and they happen frequently.

#### Minor releases

Minor releases introduce new features, new functionalities, but everything is still backwards compatible, without any breaking changes. Whenever there is a minor release, we reset the patch number to zero: 1.1.0

#### Major releases

These are for significant changes that are not backwards compatible. Some features may be entirely removed, and it will often include breaking changes. When there is a major release, minor release and patch release number are reset to zero: 2.0.0.

### Git tag commands

There are a couple of things we can do with tags.

#### Viewing tags

The simple command, git tag, will list all the tags in the current repo.

git tag –l

We can simply omit the –l flag.

we can also filter tags or search for specific tags by specifying something like a wildcard after the –l flag.

git tag –l “\*beta\*”

This will return all the versions that include ‘beta’.

git tag –l “v17\*”

This will return all the versions starting with ‘v17’.

#### Checking out tags

To view the state of a repo at a particular tag, we can use the git checkout command with the tag name. This puts us in the detached HEAD state.

git checkout 15.3.1

#### Comparing tags

We can use the git diff command to compare the state of our project in two different tags.

git diff v17.0.0 v17.0.1

#### Making tags

Again we have two types of tags, we mentioned earlier. To create a lightweight tag:

git tag <tag-name>

With this command, the tag will point to where the HEAD is pointing at the moment in time.

To create annotated tags:

git tag –a <tag-name>

This command will open our text editor, prompting us for a message and some additional details.

To view more information about annotated tags, we use the git show command.

git show <tag-name>

This will show us the tagger name, email, date, the commit related to that tag including the commit hash, the author, the date, the commit message and some more stuff.

**NOTE | when we create a tag, the tag name should be unique.**

**NOTE | we can add more than one tag to a specific commit.**

#### Tagging previous commits

We can also go back and tag any previous commit. To do this, we usually want to first see the commit log using the git log command with a online flag.

git log --oneline

We should copy the hash of the commit that we want to tag.

To tag this specific commit we use:

git tag <tag-name> <commit-hash>

#### Replacing tags with force

In case we need to move tags in order to make it refer to a different commit, we can use the –f flag at the end of the command mentioned right above.

git tag <tag-name> <commit-hash> -f

#### Delete tags

git tag –d <tag-name>

#### Pushing tags

By default, git push command does not transfer tags to remote servers. If you have a lot of tags that you want to push up at once, you can use the –tags option to the git push command. This will transfer all your tags to the remote server.

To push a specific tag to the remote server:

git push origin <tag-name>

To push all tags to the remote server:

git push origin --tags

# GitHub

It is a hosting platform for Git repos. We can then access these from anywhere and we can share them with other people and also collaborate on them.

## Collaboration on GitHub

Public repos are accessible to everyone, and anyone can clone it to their machine, but in order to be able to push to a repo, one must have collaborator privileges. Both public and private repos can have collaborators.

## Signing up and SSH keys

After signing up in GitHub, we need to set up our SSH keys. You need to be authenticated on GitHub to do certain operations, like pushing up code from your machine, otherwise, your terminal will prompt you every single time for your Github email and password.

Follow GitHub instructions to create SSH keys for your different machines.

## Cloning GitHub repos

This is an important part of working with remote repos in collaborative projects. The two Git commands that is related to getting and creating projects are git init and git clone.

### Git clone

Git clone gets a repo that is not on your machine and brings it to your machine. It downloads the content of a repo based on some URL that we provide. Typically, that URL will be from a service like GitHub.

git clone <URL>

**NOTE | make sure you are not inside of a repo when you clone.**

This command will initialize a new repo on your machine, and this will give you full access to the repo’s history.

**NOTE | if you run the git remote –v command on this cloned repo, you will see the remote’s name, usually called origin, and the fetch and push URLs.**

#### Getting clone URL

To get the URL needed for the clone command, we have two ways: we can simply go the repo’s page on GitHub and copy the URL in the address bar, or we can click on the green ‘Code’ button, and copy the Clone URL from there.

#### Running clone command

Now wherever we run the git clone command, that would be the place where Git is going to create a new folder for you. In this new folder we can see some files now. And also note that once we move to this new folder using the Git Bash, we will see the ‘master’ word, showing us that we are currently in a Git repo.

**NOTE | You are not allowed to just push up your changes to a repo that someone else owns. There is whole workflow for suggesting changes for becoming a contributor.**

## Creating a GitHub repo

We have 2 situations.

### Existing repo

If there is an existing local repo on your machine and you want to send it to GitHub:

* Create a new empty repo on GitHub
* Connect your local repo (add a remote): a remote a destination URL that we tell Git about, and we also give it a name. This name would be a label for the URL that we can then push code up to or fetch or pull down new code from. To see if we have already any remotes in our repo we use the git remote command:’

git remote –v

Now to add a new remote we use another version of this command:

git remote add <name> <URL>

A standard name usually used here is ‘origin’.

**NOTE | you can rename or delete remotes using these commands:**

git remote rename <old-name> <new-name>

git remote reove <name>

**NOTE | it is a common thing to have multiple remotes especially when working on open-source projects. See the workflows section.**

* Push up your changes to Github

### Start from scratch

If you have not begun working on your local repo:

* Create a new repo on GitHub
* Clone it down to your machine: the cloning process would automatically configure the remote for us.
* Do your work locally
* Push up changes to GitHub

## Push

To push our changes up to a remote repo on GitHub:

git push <remote> <branch>

This command is usually used to push the master branch to the origin remote.

git push origin master

There is currently a trend between developers that rename their Master branch to Main.

### Local branch and GitHub branch

When we command git push origin master for the first time, the master branch is created on our GitHub repo. Before that, there is no branch there. However, if there are different branches on the GitHub repo, we can push our changes to its different branches.

git push <remote> <local-branch>:<remote-branch>

It means that you don’t have to push local master to remote master necessarily. You don’t even have to push any local branch to the remote master branch. You can choose to push it to any remote branch available on your GitHub repo.

#### Establishing upstream

This would allow us to connect a branch on our local repo to a branch, usually with the same name, on the GitHub repo. Also, Git would remember this connection. This basically allows us to push the changes in the local master branch to the remote master branch using this simple command when we are located in the local master branch.

git push

To establish an upstream between a local branch and a remote branch:

git push –u origin master

This would establish a connection between the local and remote master branches. It can be used for any branch.

Git push –u origin <branch>

This would establish a connection between the local and remote <branch>. So whenever we are checked out at the local <branch>, we can simply use git push command to push the changes in this local <branch> up to the remote <branch>.

## Fetching and pulling

Git fetch and Git pull are the main two commands that have to do with getting changes down from a remote repo. These are used if we are working with collaborators and our repo is changed by other people.

### Remote Tracking Branch

When we clone a remote repo into our machine, the master branch sits in a local repo on our machine, and we have two branch reference. One is the regular branch reference, usually called master, acting like any other branch reference. Adding new commits will move this branch reference, making it always point to the tip of the branch on our machine. The other branch reference is called a Remote Tracking Branch. It is a pointer like the other one, but it does not move with our commits, it points back to the last known commit on the master branch of the origin remote.

We can see the remote branches that our local repo knows about:

git branch -r

So for instance, if we work and commit 2 changes on our local repo after cloning it from a remote repo, by running git status command, Git reports back that ‘your branch is ahead of origin/main’ by 2 commits.’

At this stage, you can checkout at origin/master to see what was the project like when you first cloned it.

git checkout origin/master

This will put us in ‘detached HEAD’ state. We can then switch back to our main branch again.

git switch main

if we push our local changes to the remote repo using the git push command, and then get git status, we see that the Remote Tracking Branch has now moved to the current commit position, it is no longer left behind.

### Working with remote branches

When we clone a remote repo, what we get on our machine is a local repo with only 1 branch, and that would be the remote repo’s main branch. If the remote repo has other branches, we won’t have them on our local cloned repo, or we will?

If we command:

git branch

Git reports back only 1 branch called main. But if we command:

git branch –r

Git reports all the remote branches that are available on the remote repo. So just like before, we can check out at these remote branches like:

git checkout origin/puppies

But this, again, will put us in a detached HEAD state. We want to be able to work on a local puppies branch which is connected to the remote puppies branch. Thanks to the switch command we can switch to a local branch with the exact same name of remote branch, and Git will make that happen for us.

git switch puppies

So Git will detect that we are going to create a new branch with a name that already exists in the remote repo’s branches and it will assume that we want them connected. So our local branch will be updated with all the content and files and folders of the remote branch. Again, if you work and make changes on this local branch, you will get ahead of the Remote Tracking Branch which points to where the remote puppies branch were when you first created your local puppies branch and connected to it.

### Git fetch and Git pull

When we have a local repo cloned from a remote repo, there are situations when that remote repo is upgraded with some commits while we were working on our local cloned repo. In this case, we would need commands other than git add, git commit, and git push. These were used to add things to the remote repo when we were ahead of the remote repo. But in this case, the remote repo is ahead of us. So we would need git fetch and git pull that works in the opposite direction of the three commands mentioned, so they get new commits from the remote repo.

#### Git fetch

Gets commits from the remote repo and put it into our local repo. It download changes from a remote repo, but those changes will not be automatically integrated into our working files. It just lets you see what others have been working on.

git fetch <remote>

if you leave the remote repo name empty, it would by default be origin. If you have multiple remotes, you should define from which one you want to fetch. As a result, the origin/master branch reference will move on to include the latest commits from the remote repo, while my local master branch stays right were we were working on it.

We can be more specific with branches also.

git fetch <remote> <branch>

**NOTE | to see what has been upgraded in the fetched repo or branch, we should checkout to the remote branch that is now fetched and put into our local repo.**

Git checkout origin/<branch>

**NOTE | if the remote repo is upgraded with some commits, our local repo will not know about it until we fetch the upgrades. So if the remote repo is upgraded and we didn’t run git fetch, git status will respond that our local repo is up-to-date. If we command fetch, and then get git status, it will report that we are behind.**

#### Git pull

Gets commits from the remote repo and put it into our working directory. So the pull command will update our working directory.

git pull <remote> <branch>

Where we run this command from matters. Whatever branch we are on, that is where the changes will be merged into when pulling. So pulls can result in merge conflicts.

**NOTE | As a good practice, before you ever push something up to GitHub, you’d better pull down and see if there are changes, if there is anything that is going to be problematic and that it would have conflicts. If there are conflicts, we would have to resolve them manually, then commit the final changes, and then use the push command to push it up to the remote repo. Note that after resolving the conflicts, git status will usually tell you that your local branch is ahead of the remote branch by 2 or more steps. That is because we pull first, do some changes, commit them, and then we push.**

**NOTE | it is not recommended to use git pull command if you have uncommitted changes in your local repo.**

**NOTE | if we have only one remote repo connected to our local repo, we can use the short version of git pull command.**

git pull

**which will default for the origin remote repo, and it will also default for the branch name that we are currently in. So if we are in the food branch and we command git pull, it will get all changes from the food remote branch and put it into our local branch and merge it into our working directory.**

## Repo’s Readme

A README file on a GitHub repo is used to communicate important information about a repo including:

* What the project does
* How to run the project
* Why it is noteworthy
* Who maintains the project

If you put a ‘README.md’ file in the root of your project, GitHub will automatically grab its content and show it on the repo’s page.

READMEs are markdown files. Markdown is a very succinct convenient syntax to generate formatted text.

### Markdown syntax

Markdown is not limited to GitHub, you can use it anywhere. You can visit this link for full guide: <https://markdown-it.github.io/>

#### Headings

For creating headings 1 to 6, we use the respective number of hashes, then a space, and then the heading content.

# h1 Heading1

## h2 Heading 2

#### Horizontal rules

To create horizontal lines separating different sections we can use:

---

\_\_\_

\*\*\*

#### Emphasis

To add different kinds of emphasis to a part of your text:

\*\*This is bold text\*\*

\_\_This is bold text\_\_

\*This is italic text\*

\_This is italic text\_

~~Strikethrough text~~

#### Block quotes

To insert a block quotes in different levels we use the respective number of > signs.

> Block quote

>> Nested block quote

> > > Nested block quote

#### Create separate paragraphs

To create a separate paragraph we should insert one empty line between the new paragraph and the previous one.

<paragraph1>

<paragraph2>

#### Lists

To create unordered lists we should start a line with +, -, or \*.

+ list item

Sub-lists are made by indenting 2 spaces, and if marker character is changed, a new list will start.

+ list item

- list item

#### Code syntax

To create code syntax for a single word:

`Code`

To create a block of code syntax:

\\ Some code

continuing code block

To create block code fences:

```

Code

```

If we want to implement automatic syntax highlighting according to the language, we should define the language at the fence.

``` js

Code

```

#### Links

To create a link text we should place a [link text] at the beginning of the line:

[link text](<URL>)

A link with a title would be like:

[link with title](<link> “title”)

## Collaboration workflows

This will all be about collaboration workflows: different ways of configuring a repo, different ways of working on branches, when to create branches, how to merge changes, how to ask to merge changes, how to get permission.

### Feature branches

In this workflow, no one works on the master branch. Everyone works on a feature branch. This means treating the master or main branch as the official project history. The content of the feature branches will be merged back into the master branch when appropriate.

Imagine that Steve wants to work on a feature for a project. He should create a new branch with proper name and implement the feature in that branch.

**NOTE | A good practice at this point is that before creating the new branch, we usually want to pull all recent work on the master branch. We always prefer to have the most recent version of the master branch on our system before creating a new feature branch.**

After creating the new branch and working on it, at some point, when Steve’s work is not done yet, he decides to check his code with some coworker, Pamela. Steve should push his new branch to the origin repo on GitHub.

**NOTE | this is not to suggest that every time you make a feature branch, you need to push it up to GitHub. Most of the time, you won’t do it unless you have a reason for it, for instance, when you need to share the feature with someone else.**

So a new branch will now be available on GitHub, and Pamela will be able to pull it down on her system to take a look at it. Of course, at this point, Pamela will first fetch the new branch from the origin repo. She can then checkout at that new branch, which will make her go into the detached head state and allow her to take a look at the code that Steve has added. Now if Pamela decides to help Steve and add some code for him on that branch, she would have to leave the detached head state. What she will have to do now is to switch to that branch on her system even though it currently doesn’t exist on her machine. Git will automatically create the branch based on the origin repo that it is tracking. Now the branch that Steve added to the project is also available on Pamela’s machine and she can work on it. Note that in this scenario, Steve would probably have to stop working on this branch while Pamela is working on it. Finally, when Pamela is done, she can push up this branch to the same branch on the origin repo. From here, Steve would be able to pull the updated branch to his machine and continue working on it.

#### Merging feature branches

Merging feature branches or some of them at least is inevitable. We need to merge them into the master or main branch. So back to the example scenario above, none of the works done by Steve or Pamela is not pushed to the master branch, they are not integrated.

In order to merge your feature branch to the master branch you usually discuss this beforehand through emails or chat messages. When everything is fine, then we push our branch to the master branch on the origin repo.

**NOTE | it is a good practice to pull all the recent changes applied to the master branch before pushing your branch to it.**

However, the most common way to go is the pull request.

#### The pull request

You want to make sure that whatever you are merging in or whoever is doing the merge, has some discussion around it, has some approval. So there is some code review process involved and pull requests are in integral part of this process.

Pull requests allow developers to tell other team members that they have some new work they want to be reviewed. It would be a new work that is on a branch that they want to be discussed, then accepted or rejected. They can even update the work on the pull request.

When you push a new branch onto the origin repo, you will see kind of an alert message on the repo’s page on GitHub, suggesting a compare and pull request. So we can send the pull request along with some comment, and wait for the boss to respond.

The boss may respond with some additional tasks for you in order to fix some things before your work can be merged into the master branch. Finally, when everything is fine, the boss can decide to merge the pull request into the master branch on the origin repo. So this is only for the repo’s owner to decide, not you as a collaborator, unless you are granted the permission to do so.

After the merge is done on GitHub, the master branch on the origin repo will have code that the repo’s owner and also other collaborators don’t have them on their machine. So they would have to use the git pull origin main command to update their local repo. Even the person who worked on the feature branch, will not yet have his code integrated into the master branch on his local repo. He would also have to use the same pull command to update.

**NOTE | We usually don’t want branches to remain existent after they are merged into the master branch. They are normally deleted after the merge process is complete.**

**NOTE | Merging pull requests can also face conflicts.**

### Fork-and-Clone

This workflow is only used in large-scale projects where there are thousands of contributors, and the project owners are not able to invite each and every individual developer and allow them one by one to work on the project.

In this workflow, instead of just one centralized repo on GitHub, every developer has their own GitHub repo in addition to that centralized repo. Individual developers can make changes and push their own versions (forks) before making pull requests.

The fork-and-clone workflow enables anybody to try and make a contribution. You won’t need permissions, you make your own copy, you try making changes and then you make a PR.

#### Forking

This is a feature of GitHub. When we fork a repo, it creates a personal copy of someone else’s repo on our GitHub account. Compare this with the situation where you clone a public repo and do some changes. You will not be able to create a pull request on the repo’s GitHub page unless you contact the owners and ask them to give you collaborator privileges. That is not the way to go.

After forking a repo, we can clone the forked repo to our machine. This will allow me to simply push my changes to this forked repo on GitHub, but if we cloned the original repo, we could not push to it, because we are not permitted to do so.

By cloning the forked repo on our machine, we can do any changes we want with the code and the branches and literally everything. Once you make changes and push them to your forked repo, GitHub will give you the option to attempt to share your work from your forked repo to the original repo by making a pull request.

#### Upstream: your link to the originals

The forked and cloned repo is usually called Origin, just like any repo that is cloned to our machine. This would be a remote for our local repo. Note that your forked repo is frozen in time, it does not update itself automatically with the original repo. So in addition to this remote, we need to set up another remote, usually called Upstream. This would refer to the original repo that we forked. Remember we said before that we can have more than one remotes for our local repo.

These two remotes allows you to do you own work and push changes up to your forked repo, but also you will be able to get the latest changes and updates to the original repo into your machine. The second remote will allow you to pull changes to your machine.

In this workflow, after you are done with some work that you have done, you can send a pull request from your forked repo to the original repo. Then it is up to the original repo’s owners to accept or reject your request. If they accept you request, the original repo will now include your changes. So the next step would be to use the second remote (upstream) to pull down the changes made to the original repo, in order to have your upstream remote updated with the original repo on GitHub.

#### The workflow

1. Go to the original repo’s page on GitHub and fork the repo.
2. Go to the forked repo on your account and copy its link, and use it to clone this forked repo to your machine. Use the git clone command. This adds the first remote to your local repo.
3. Go to the original repo’s page on GitHub and copy its link, and use it to create the upstream remote for your local repo. Use git remote add command. This adds the second remote to your local repo. You now have two remotes: origin, upstream.
4. You can now work on your local repo and push your work to your forked repo on GitHub. Then you can send a pull request to the original repo. You can also update your local repo with the latest changes applied to the original repo using the upstream repo with the git pull command.
5. If your changes in a pull request are accepted and merged to the original repo, you should remember to use the upstream remote to update your local repo with the new state of the original repo.

# Open-source projects

* React
* VS code
* TensorFlow