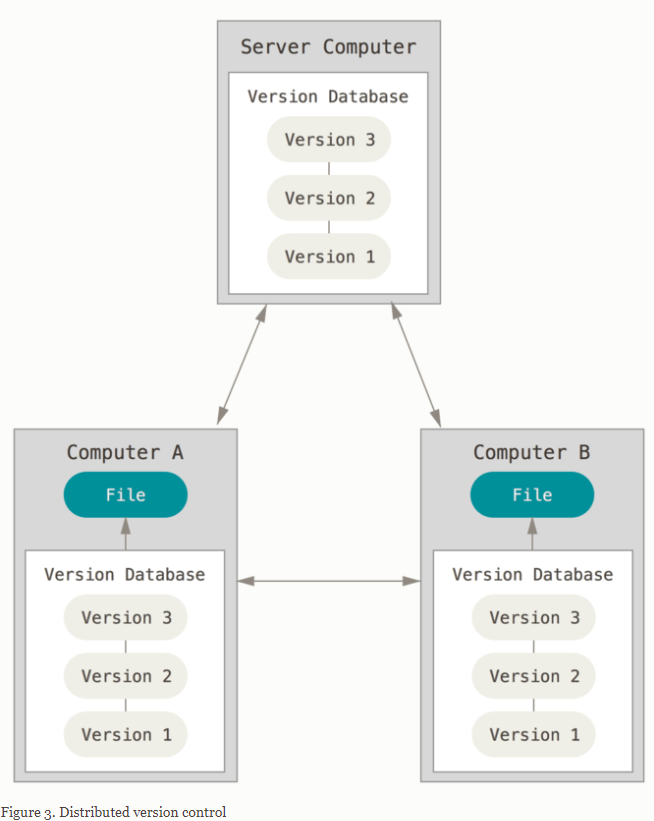
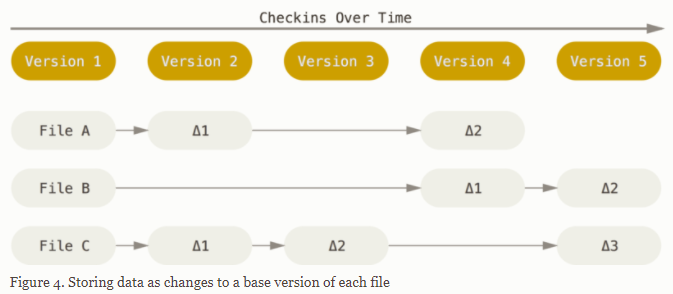
https://www.analyticsvidhya.com/blog/2020/05/git-github-essential-guide-beginners/?utm\_source=linkedin&utm\_medium=AVlinkhigh-performance-blogblogs441960.375

What is “version control”, and why should you care? Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later.

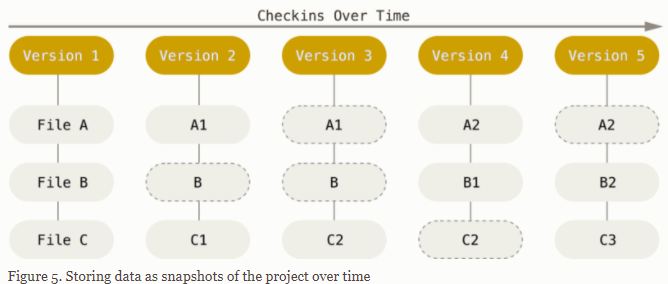
This is where Distributed Version Control Systems (DVCSs) step in. In a DVCS (such as Git, Mercurial, Bazaar or Darcs), clients don’t just check out the latest snapshot of the files; rather, they fully mirror the repository, including its full history. Thus, if any server dies, and these systems were collaborating via that server, any of the client repositories can be copied back up to the server to restore it. Every clone is really a full backup of all the data.



The major difference between Git and any other VCS (Subversion and friends included) is the way Git thinks about its data. **Snapshots, Not Differences!!!!**



Git doesn’t think of or store its data this way. Instead, Git thinks of its data more like a series of snapshots of a miniature filesystem. With Git, every time you commit, or save the state of your project, Git basically takes a picture of what all your files look like at that moment and stores a reference to that snapshot. To be efficient, if files have not changed, Git doesn’t store the file again, just a link to the previous identical file it has already stored. Git thinks about its data more like a **stream of snapshots**.



Most operations in Git need only local files and resources to operate — generally no information is needed from another computer on your network. Because you have the entire history of the project right there on your local disk, most operations seem almost instantaneous.

This also means that there is very little you can’t do if you’re offline or off VPN. If you get on an airplane or a train and want to do a little work, you can commit happily (to your **local** copy, remember?) until you get to a network connection to upload.

Checksummed:

Everything in Git is checksummed before it is stored and is then referred to by that checksum. This means it’s impossible to change the contents of any file or directory without Git knowing about it. This functionality is built into Git at the lowest levels and is integral to its philosophy. You can’t lose information in transit or get file corruption without Git being able to detect it.

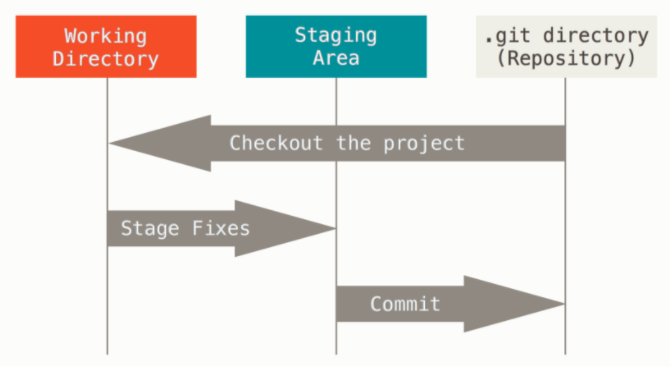
The mechanism that Git uses for this checksumming is called a SHA-1 hash. This is a 40-character string composed of hexadecimal characters (0–9 and a–f) and calculated based on the contents of a file or directory structure in Git. A SHA-1 hash looks something like this:

 In fact, Git stores everything in its database not by file name but by the hash value of its contents.

States:

Git has three main states that your files can reside in: **modified**, **staged**, and **committed**:

* Modified means that you have changed the file but have not committed it to your database yet.
* Staged means that you have marked a modified file in its current version to go into your next commit snapshot.
* Committed means that the data is safely stored in your local database.



The working tree is a single checkout of one version of the project. These files are pulled out of the compressed database in the Git directory and placed on disk for you to use or modify.

The staging area is a file, generally contained in your Git directory, that stores information about what will go into your next commit. Its technical name in Git parlance is the “index”, but the phrase “staging area” works just as well.

The Git directory is where Git stores the metadata and object database for your project. This is the most important part of Git, and it is what is copied when you **clone** a repository from another computer.

The basic Git workflow goes something like this:

1. You modify files in your working tree.
2. You selectively stage just those changes you want to be part of your next commit, which adds **only** those changes to the staging area.
3. You do a commit, which takes the files as they are in the staging area and stores that snapshot permanently to your Git directory.

Git:

Git is a widely used Distributed Version Control System (DVCS) written in C

keeping track of all the code modifications and allowing easy roll back to a previous version (in case of issues)

Ensuring that every project collaborator will have a history of the changes made on their local machine.

So people can work on different features of the project without having to communicate with the server hosting the remote version of the project.

This is super efficient and you can easily merge any changes made to the project with the remote copy.

Github:

GittHub is a widely used platform for version control that uses Git at its core.

It lets you host the remote version of your project from where all the collaborators can have access to it.

GitHub is where you can find a plethora of open-source projects with their codes. All the new and emerging technologies can be found on this platform.

Gitbash:

Now, Git is designed to work with a Unix style command-line environment. Linux and macOS already have an interface for this in their native command-line terminals.

Windows, however, has a completely different command-line interface called Command Prompt which is not a Unix style command-line environment.

Git Bash is a command-line interface for Windows that emulates the Git command-line experience.

Repo:

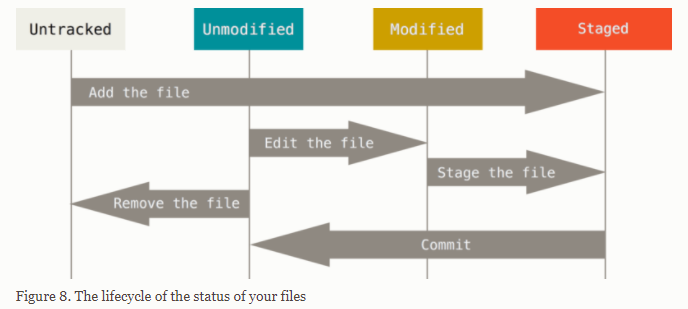
Repository or Repo is a folder that contains all the project files and the history of the revisions made to each file.

There are two repositories of your project that you will work with throughout the lifetime of your project – Remote repo and Local repo:

Remote repo contains your project that can be accessed from anywhere and by anyone. Your remote repository lives on the GitHub server and anybody can access them

Local repo is a copy of the remote repo that resides on your local machine. All the changes you make will be saved in your local repo. Y

our fellow developers will not be able to see them yet until you push/publish them to the remote repo in GitHub



Git status

Remember that each file in your working directory can be in one of two states: **tracked** or **untracked**. Tracked files are files that were in the last snapshot; they can be unmodified, modified, or staged. In short, tracked files are files that Git knows about.

Untracked files are everything else — any files in your working directory that were not in your last snapshot and are not in your staging area.

When you first clone a repository, all of your files will be tracked and unmodified because Git just checked them out and you haven’t edited anything.

As you edit files, Git sees them as modified, because you’ve changed them since your last commit. As you work, you selectively stage these modified files and then commit all those staged changes, and the cycle repeats.

Git - First Commands

git clone <Repo-URL>

Cloning means creating a copy of the remote repo on your local machine. Now you can make changes to the project on your local machine.

git commit -m “<commit message>”

When you commit a change, you save the changes you made to your files in the repo. When working with Git from your local machine, using the commit command will save your files in the local repo.

To make those changes in the remote repo, you will use the push command.

git push origin <branch>

Push command allows you to transfer all the changes on your local repo to the remote repo.

Now all the fellow developers will have access to the changes you made and they can update their local repositories.

git pull <remote-repo>

If push meant transferring code to the remote repo, the Pull command allows you to transfer all the changes from the remote repo to your local repo.

So any changes that your fellow developer pushed to the remote repo, you can transfer them to your local repo using the pull command.

git log --> see all changes / commit performed

Bash commands:

pwd -W --> pring working directory in particular format (C:/Users/brionm1)

cd ~ --> go back to your working directory (C:/Users/brionm1)

cd R --> change workding directory from briom1 to R (in C:/Users/brionm1, after pwd /c/Users/brionm1/R or pwd -W C:/Users/brionm1/R)

ls --> list content of directory

ls -a --> list content including hidden files

mkdir git --> create git folder in (C:/Users/brionm1; cd github then after pwd -W C:/Users/brionm1/github)

git --version --> know which version of git is installed

history --> all commands history & typed so far

Git Global Configuration

git config --global user.name "Michael\_Brion"

git config --global user.email "michael.brion@takeda.com"

git config --global core.editor "nano -w" --> set up your favorite text editor

git config --list --> to check current configuration

git init --> will create hidden .git folder dans le directory ; the git folder contains all internal mechanics for one project

Only once per project / repo to initialize git folder & mechanic & versionning

Once you have created and added files to your project directory, you can add them to your local Git repository using

git add <file-name> or git add . (to add all files in the staging area)

We haven’t added the files to the local repo yet. We have just told Git that some changes were made and we want to save these changes in the next commit/save.

git status command tells you where you are (branch master ? some commits already ? ..)

Now you can take a snapshot of all the changes you made, which are reflected in the staging area, and save them in the Git repo using

git commit -m “<commit message>” --> empy message will abord the commit!!!!!!

git commit -m "XXXX" --> will automatically add the message "XXXX" and not open editor

git diff FILENAME --> to see difference between versions or commits (red: deleted; green: added)

(if you already put the file in the staging area --> won't work because know already about changes"

need to use git diff --staged command if you file is in the staging area (you use git add but not yet git commit)

nano README.md (create README file in nano text editor)

insert some text inside, exit

note: you need to have username & user.email configured using git config commands

cat README.md --> to see / read the README file directly in bash

After pushing some changes and checking git log, we see:

git log --oneline

8380ef4 (HEAD -> master) log and diff commands

91862fd add new commands

f4795df To modify the git command explanation

feff535 mistake as I used ´´ instead of ``

2fd65e8 Add git add and git commit commands in the readme files

69ac873 talked about init and status for first commit

git diff HEAD~3 HEAD~2: Current branch, diff between commits 2 and 3 times back

git diff: Your working copy and staging area:

git diff --staged: Staging area and the latest commit:

git diff 4ac0a6733: Your working copy and commit 4ac0a6733:

git diff 4ac0a6733 HEAD: Commit 4ac0a6733 and the latest commit:

git diff 4ac0a6733 826793951: Commit 4ac0a6733 and commit 826793951

git-restore is a tool to revert non-commited changes. Non-commited changes are: a) changes in your working copy, or b) content in your index (a.k.a. staging area).

git-restore seems the safer option if you only want to revert local work.

git-reset can modify your repository

git-reset is about updating your branch, moving the tip in order to add or remove commits from the branch. This operation changes the commit history.

Undoing things:

At any stage, you may want to undo something. Here, we’ll review a few basic tools for undoing changes that you’ve made. Be careful, because you can’t always undo some of these undos.

This is one of the few areas in Git where you may lose some work if you do it wrong.

git commit -m 'Initial commit'

git add forgotten\_file

git commit --amend

You end up with a single commit — the second commit replaces the results of the first.

Unstaging:

For example, let’s say you’ve changed two files (README.md & Contributing.md) and want to commit them as two separate changes,

but you accidentally type git add \* and stage them both. How can you unstage one of the two? The git status command reminds you:

git add \*

git status

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

renamed: README.md -> README

modified: CONTRIBUTING.md

git reset HEAD CONTRIBUTING.md --> will unstage the contributing.md file

Note that the file contributing.md is still modified as expected but unstaged)

Unmodifying a Modified File

What if you realize that you don’t want to keep your changes to the CONTRIBUTING.md file?

How can you easily unmodify it — revert it back to what it looked like when you last committed

Luckily, git status tells you how to do that, too. In the last example output, the unstaged area looks like this:

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: CONTRIBUTING.md

It tells you pretty explicitly how to discard the changes you’ve made. Let’s do what it says:

git checkout -- CONTRIBUTING.md

git status

It’s important to understand that git checkout — <file> is a dangerous command. Any local changes you made to that file are gone —

If you would like to keep the changes you’ve made to that file but still need to get it out of the way for now, we’ll go over stashing and branching in Git Branching; these are generally better ways to go.

Remember, anything that is committed in Git can almost always be recovered. Even commits that were on branches that were deleted or commits that were overwritten with an --amend commit

can be recovered (see Data Recovery for data recovery). However, anything you lose that was never committed is likely never to be seen again.

NOTE

Git version 2.23.0 introduced a new command: git restore. It’s basically an alternative to git reset which we just covered.

From Git version 2.23.0 onwards, Git will use git restore instead of git reset for many undo operations.

git restore --staged <file>

git restore --staged CONTRIBUTING.md

git restore <file>

git restore CONTRIBUTING.md

https://git-scm.com/book/en/v2/Git-Basics-Undoing-Things

In Git terms, a "checkout" is the act of switching between different versions of a target entity. The git checkout command operates upon three distinct entities: files, commits, and branches.

git checkout can be used to view old commits.

git checkout STRING\_COMMIT\_NUMBER FILENAME (see git log --oneline) : this goes directly to the staging area

if you commit then your new HEAD will be the past commit version

however, git won't erase previous versions, there is no cycle ! it will create only a new commit (using commit 1) but you can checkout the newer versions as well

Remember that the HEAD is Git’s way of referring to the current snapshot. Internally, the git checkout command simply updates the HEAD to point to either the specified branch or commit.

When it points to a branch, Git doesn't complain, but when you check out a commit, it switches into a “detached HEAD” state.

This is a warning telling you that everything you’re doing is “detached” from the rest of your project’s development.

The point is, your development should always take place on a branch—never on a detached HEAD.

This makes sure you always have a reference to your new commits. However, if you’re just looking at an old commit, it doesn’t really matter if you’re in a detached HEAD state or not.

git checkout STRING\_NUMBER\_GITLOG FILENAME ; this will create a detached HEAD at commit

HEAD is now at STRING\_NUMBER\_GITLOG and you can start creating branches

git checkout master will bring you to the HEAD

SSH: you don't need to communicate username, password when communicating with github

ssh-keygen.exe will create public (to share) & private key (never share)

SSH will be used to communicate, exchange with github (and every project inside)

Summary:

git init: turn the current folder into a git repository

git status: let’s you know what is going on run this all the time!

git add: put file(s) into your “staging area”

git commit -m 'MY COMMIT MESSAGE': commits files in the “staging area” with the given message

git diff <file name>: compares saved changes to a file to the last commited version of the file

git diff --staged <file name>: compares a staged file to the last commited version of the file

git log and git log --oneline: looks at your git history

git log --oneline --graph --decorate --all: gives detailed log information about where you are

Remote = origin

push work to github

pull work from github

A remote in Git is a common repository that all team members use to exchange their changes. In most cases, such a remote repository is stored on a code hosting service like GitHub

In Git, "origin" is a shorthand name for the remote repository that a project was originally cloned from. More precisely, it is used instead of that original repository's URL - and thereby makes referencing much easier.

git clone https://github.com/gittower/git-crash-course.git

In the following example, the URL parameter to the "clone" command becomes the "origin" for the cloned local repository:

to synchronize your work between local & remote-origin

push <WHERE> <WHAT>

pull <WHERE> <WHAT>

To be able to collaborate on any Git project, you need to know how to manage your remote repositories.

Remote repositories are versions of your project that are hosted on the Internet or network somewhere.

You can have several of them, each of which generally is either read-only or read/write for you.

Collaborating with others involves managing these remote repositories and pushing and pulling data to and from them when you need to share work.

Managing remote repositories includes knowing how to add remote repositories, remove remotes that are no longer valid, manage

various remote branches and define them as being tracked or not, and more.

Once you have created your GitHub repository, GitHub will prompt you to upload your files to the remote repository:

As we have already created our local repository, we first need to sync our local and remote repos. We can do this using git remote add origin <URL>

To add a new remote Git repository as a shortname you can reference easily, run git remote add <shortname> <url>

git remote add pb https://github.com/paulboone/ticgit

git remote add origin git@github.com:MichaelBrionTakeda/GitNYR.git

git remote -v --> gives the URL of each remote

origin git@github.com:MichaelBrionTakeda/GitNYR.git (fetch)

origin git@github.com:MichaelBrionTakeda/GitNYR.git (push)

pb https://github.com/paulboone/ticgit (fetch)

pb https://github.com/paulboone/ticgit (push)

git fetch <remote> --> git fetch pb

As you just saw, to get data from your remote projects, you can run:

If you want to push your master branch to your origin server (again, cloning generally sets up both of those names for you automatically),

then you can run this to push any commits you’ve done back up to the server:

git push origin master

Pushing to Your Remotes

When you have your project at a point that you want to share, you have to push it upstream. The command for this is simple:

git push <remote> <branch>.

git push -u origin <branch-name>

Summary:

git clone <repo url>: downloads code from a code repository into your current directory

git remote -v: lists all the remotes and their short names (e.g., origin, upstream)

git remote add <name> <url>: adds the <url> to your remotes and gives it the short name <name> (e.g., git remote add origin <URL>)

git remote rm <name>: removes a remote by its shortname

git push <where> <what>: pushes code on the <what> branch to the <where> remote (e.g., git push origin master)

git pull <where> <what>: pulls does down from the <what> branch from the <where> remote (e.g., git pull origin master)

you can edit directy in github but you need to pull the change to reflect that in your local computer

git pull origin master

git log --oneline --graph --all: shows your the entire history, nice graph showing merge, ...

if you make a change locally in git and another change in github remotely without syncrhonizing between changes

won't be able to reconcile the two commits (local one & remote one)

need to use git pull from remote and will automatically merge the two files :) changes in two different locations can be merged automatically

if you change title locally and change title remotely and use git pull origin master

the automated merging won't be able to work ! you need to open text editor / file

you will have the two changes (two different titles) in the document and need to fix it in the editor!

to fix the conflict, use git add & commit

However, trying to fix / merge can get tedious ... so we will use branches !!!!

When you make a commit, Git stores a commit object that contains a pointer to the snapshot of the content you staged. This object also contains the author’s name and email address, the message that you typed, and pointers to the commit or commits that directly came before this commit (its parent or parents): zero parents for the initial commit, one parent for a normal commit, and multiple parents for a commit that results from a merge of two or more branches.

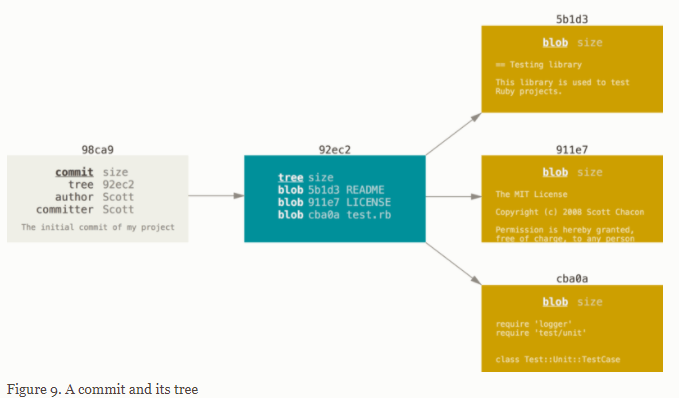
let’s assume that you have a directory containing three files, and you stage them all and commit. Staging the files computes a checksum for each one (SHA-1 hash), stores that version of the file in the Git repository (Git refers to them as **blobs**), and adds that checksum to the staging area:

$ git add README test.rb LICENSE

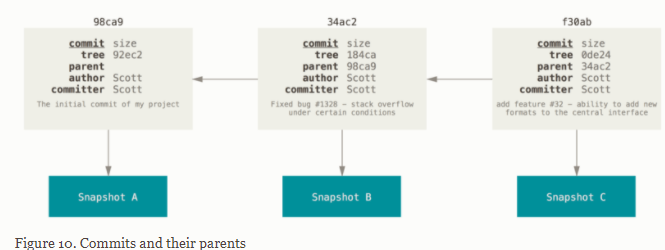
When you create the commit by running git commit, Git checksums each subdirectory (in this case, just the root project directory) and stores them as a tree object in the Git repository. Git then creates a commit object that has the metadata and a pointer to the root project tree so it can re-create that snapshot when needed.

$ git commit -m 'Initial commit'

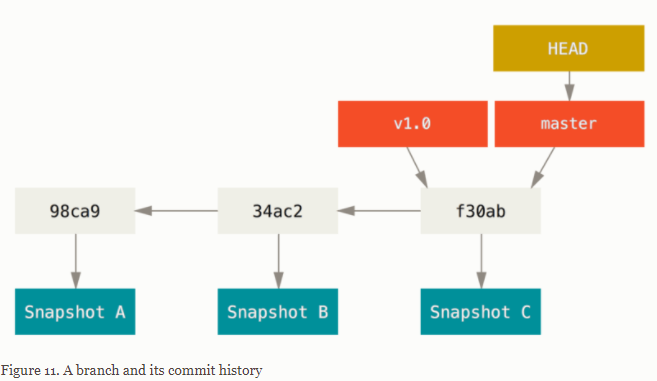
Your Git repository now contains five objects: three **blobs** (each representing the contents of one of the three files), one **tree** that lists the contents of the directory and specifies which file names are stored as which blobs, and one **commit** with the pointer to that root tree and all the commit metadata.



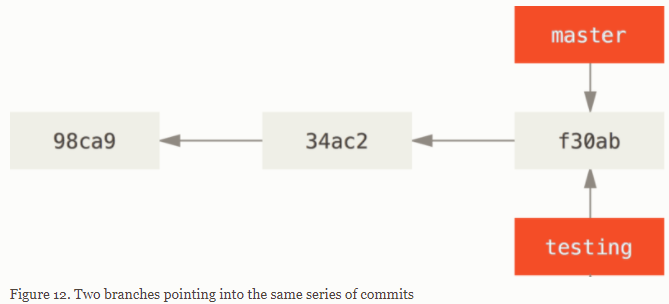
If you make some changes and commit again, the next commit stores a pointer to the commit that came immediately before it.



A branch in Git is simply a lightweight movable pointer to one of these commits. The default branch name in Git is master. As you start making commits, you’re given a master branch that points to the last commit you made. Every time you commit, the master branch pointer moves forward automatically.

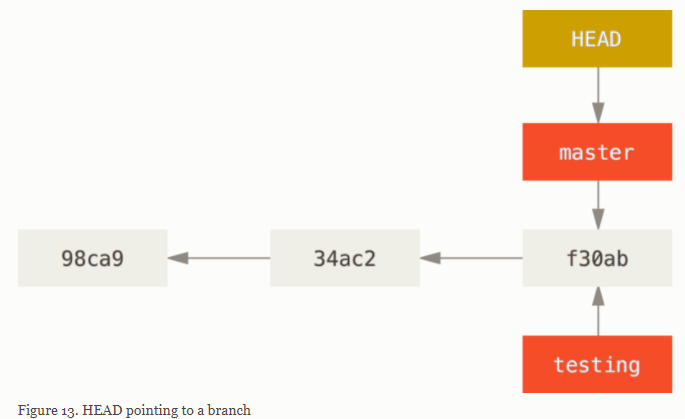


What happens when you create a new branch? Well, doing so creates a new pointer for you to move around. Let’s say you want to create a new branch called testing. You do this with the git branch command:



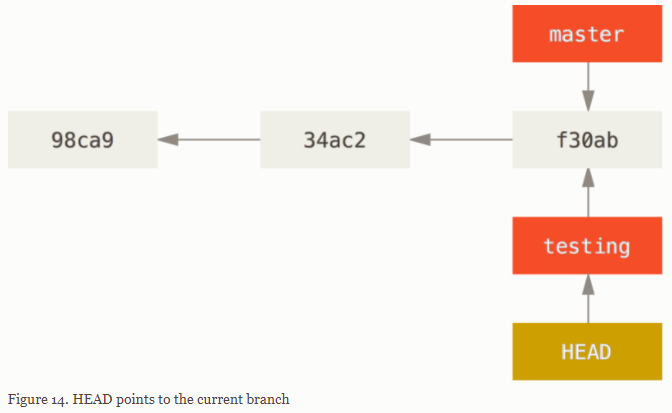
How does Git know what branch you’re currently on? It keeps a special pointer called HEAD.

In Git, this is a pointer to the local branch you’re currently on. In this case, you’re still on master. The git branch command only **created** a new branch — it didn’t switch to that branch.

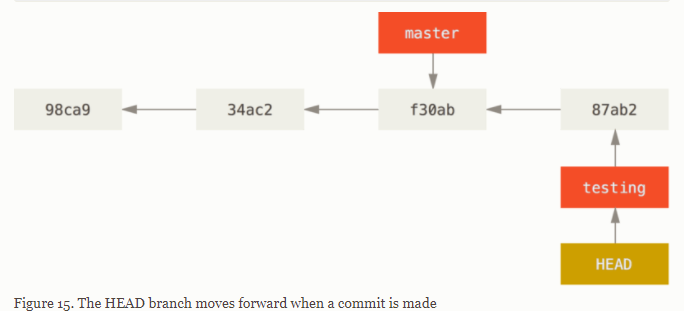


To switch to an existing branch, you run the git checkout command. Let’s switch to the new testing branch:

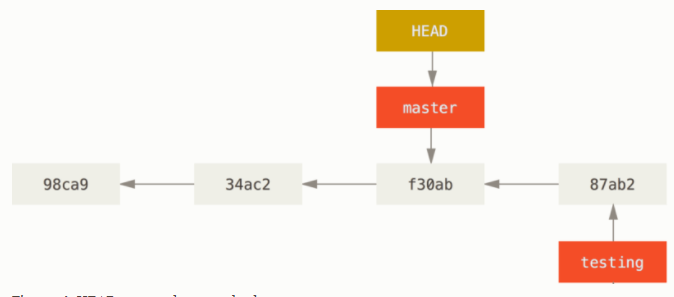
This moves HEAD to point to the testing branch.



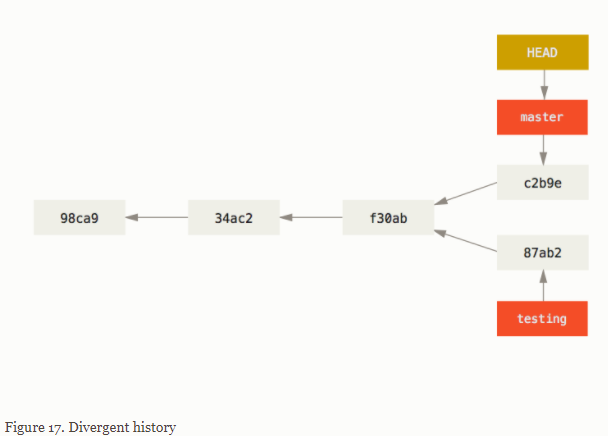
What is the significance of that? Well, let’s do another commit:



This is interesting, because now your testing branch has moved forward, but your master branch still points to the commit you were on when you ran git checkout to switch branches. Get checkout master



That command did two things. It moved the HEAD pointer back to point to the master branch, and it reverted the files in your working directory back to the snapshot that master points to. This also means the changes you make from this point forward will diverge from an older version of the project. It essentially rewinds the work you’ve done in your testing branch so you can go in a different direction.



Because a branch in Git is actually a simple file that contains the 40 character SHA-1 checksum of the commit it points to, branches are cheap to create and destroy. Creating a new branch is as quick and simple as writing 41 bytes to a file (40 characters and a newline).

my branch will contain commits not included in the master

git branch <new branch name>: creates a new branch called <new branch name>

git checkout <new branch name> (or git switch): goes to the branch, <new branch name>

git checkout -b <new branch name>: creates and checksout a branch in a single step

git log --oneline --graph --decorate --all: shows you the log in relation to all other branches

git branch testing --> new branch named testing

How does Git know what branch you’re currently on? It keeps a special pointer called HEAD.

The git branch command only created a new branch — it didn’t switch to that branch.

git checkout testing: This moves HEAD to point to the testing branch.

you can commit change then branch / head will change while you master will remain unchanged!

git log --oneline --decorate --graph --all it will print out the history of your commits, showing where your branch pointers are and how your history has diverged.

https://git-scm.com/book/en/v2/Git-Branching-Branches-in-a-Nutshell#ch03-git-branching

From Git version 2.23 onwards you can use git switch instead of git checkout to:

-Switch to an existing branch: git switch testing-branch.

-Create a new branch and switch to it: git switch -c new-branch. The -c flag stands for create, you can also use the full flag: --create.

-Return to your previously checked out branch: git switch -.

ou can run your tests, make sure the hotfix is what you want, and finally merge the hotfix branch back into

your master branch to deploy to production. You do this with the git merge command:

$ git checkout master

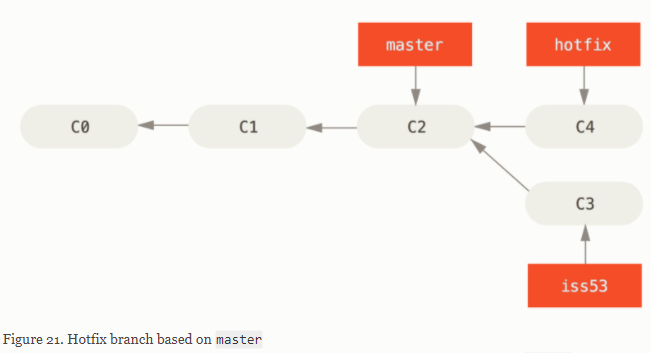
$ git merge hotfix

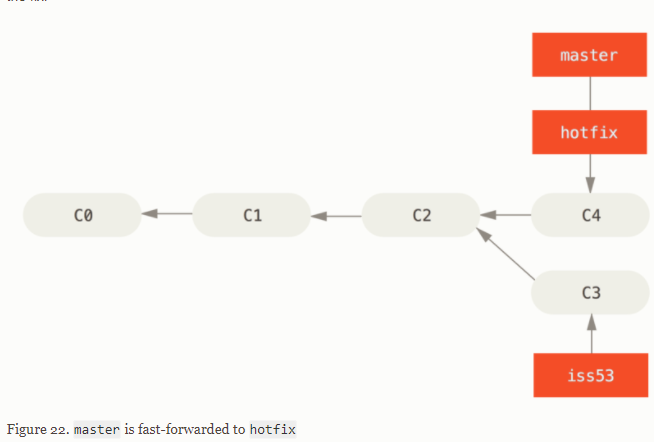
$ git branch -d hotfix --> can deleted the hotfix branch as it is no longer needed / as it is merged with master!

$ git checkout iss53 --> switch to your other issue53 branch

and son on until no issue left...

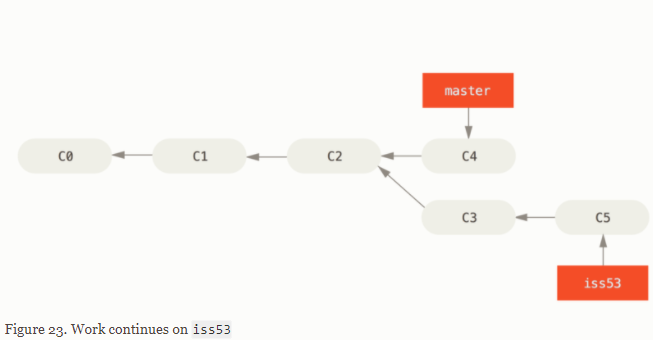
git merge <branch name> -m “<message>”

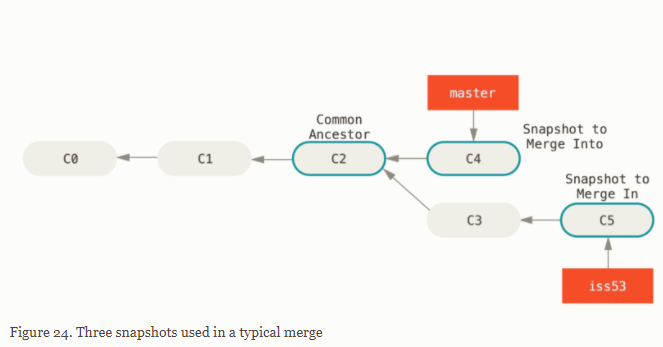




Fast-forwarded 🡪 git merge

You’ll notice the phrase “fast-forward” in that merge. Because the commit C4 pointed to by the branch hotfix you merged in was directly ahead of the commit C2 you’re on, Git simply moves the pointer forward.

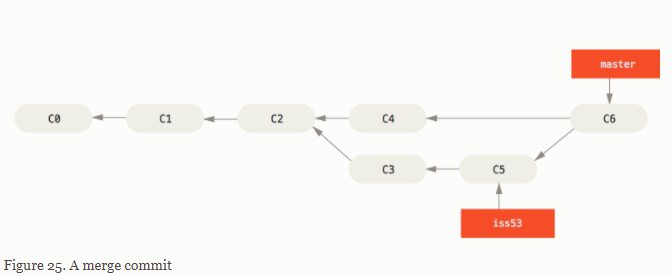




$ git checkout master

$ git merge iss53

$ git branch -d iss53

This looks a bit different than the hotfix merge you did earlier. In this case, your development history has diverged from some older point. Because the commit on the branch you’re on isn’t a direct ancestor of the branch you’re merging in, Git has to do some work. In this case, Git does a simple three-way merge, using the two snapshots pointed to by the branch tips and the common ancestor of the two.

Occasionally, this process doesn’t go smoothly. If you changed the same part of the same file differently in the two branches you’re merging,

Git won’t be able to merge them cleanly. If your fix for issue #53 modified the same part of a file as the hotfix branch, you’ll get a merge conflict that looks something like this:

--> need to resolve manually; can use git status and check for unmerged stuffs to help!

If you want to use a graphical tool to resolve these issues, you can run git mergetool, which fires up an appropriate visual merge tool and walks you through the conflicts:

You can run git status again to verify that all conflicts have been resolved:

If you’re happy with that, and you verify that everything that had conflicts has been staged, you can type git commit to finalize the merge commit.

git checkout master (HEAD -> master)

git checkout name\_branch (HEAD -> branch)

--> use to switch from branch to master and vice versa

typically branches are only for yourself (to avoid conflicts, merge, ...)

git push origin my\_first\_branch

pull requests is how we can merge branches on the master, confirm merge

pull to local computer

pull run fetch and merge at the same time

you can use only fetch (see git remote -v)

git branch <new branch name>: creates a new branch called <new branch name>

git checkout <new branch name>: goes to the branch, <new branch name>

git checkout -b <new branch name>: creates and checksout a branch in a single step

git log --oneline --graph --decorate --all: shows you the log in relation to all other branches

The git branch command does more than just create and delete branches. If you run it with no arguments, you get a simple listing of your current branches:

$ git branch

iss53

\* master

testing

Notice the \* character that prefixes the master branch: it indicates the branch that you currently have checked out (i.e., the branch that HEAD points to). This means that if you commit at this point, the master branch will be moved forward with your new work. To see the last commit on each branch, you can run git branch -v:

The useful --merged and --no-merged options can filter this list to branches that you have or have not yet merged into the branch you’re currently on. To see which branches are already merged into the branch you’re on, you can run git branch --merged:

Because you already merged in iss53 earlier, you see it in your list. Branches on this list without the \* in front of them are generally fine to delete with git branch -d; you’ve already incorporated their work into another branch, so you’re not going to lose anything.

To see all the branches that contain work you haven’t yet merged in, you can run git branch --no-merged: Because it contains work that isn’t merged in yet, trying to delete it with git branch -d will fail:

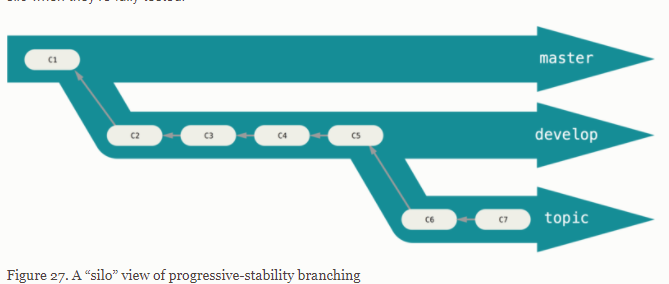
If you really do want to delete the branch and lose that work, you can force it with -D, as the helpful message points out.

if we want to create a branch from a previous commit,

we need to use get checkout <HASH> --> you'll get head detached then git switch -c

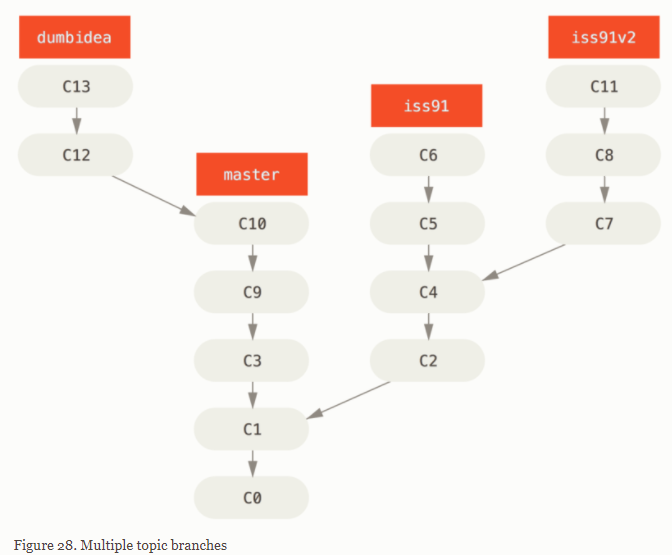
or git checkout -b branch\_name ?

Many Git developers have a workflow that embraces this approach, such as having only code that is entirely stable in their master branch — possibly only code that has been or will be released. They have another parallel branch named develop or next that they work from or use to test stability — it isn’t necessarily always stable, but whenever it gets to a stable state, it can be merged into master. It’s used to pull in topic branches (short-lived branches, like your earlier iss53 branch) when they’re ready, to make sure they pass all the tests and don’t introduce bugs.

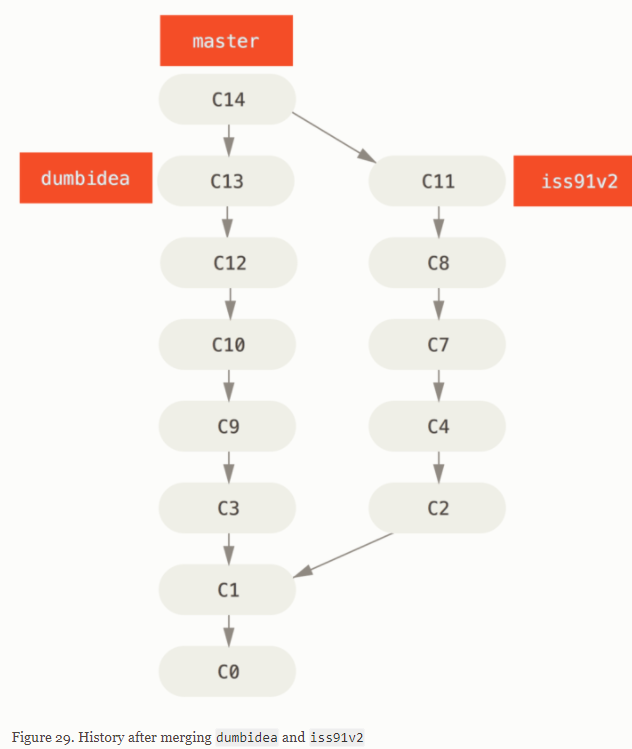


You can keep doing this for several levels of stability. Some larger projects also have a proposed or pu (proposed updates) branch that has integrated branches that may not be ready to go into the next or master branch. The idea is that your branches are at various levels of stability; when they reach a more stable level, they’re merged into the branch above them. Again, having multiple long-running branches isn’t necessary, but it’s often helpful, especially when you’re dealing with very large or complex projects.

Topic branches, however, are useful in projects of any size. A topic branch is a short-lived branch that you create and use for a single particular feature or related work. You saw this in the last section with the iss53 and hotfix branches you created.



let’s say you decide you like the second solution to your issue best (iss91v2); and you showed the dumbidea branch to your coworkers, and it turns out to be genius. You can throw away the original iss91 branch (losing commits C5 and C6) and merge in the other two. Your history then looks like this:



Dumbidea merge 🡪 fast forwarded

C14 is new merge commit master/iss91v2

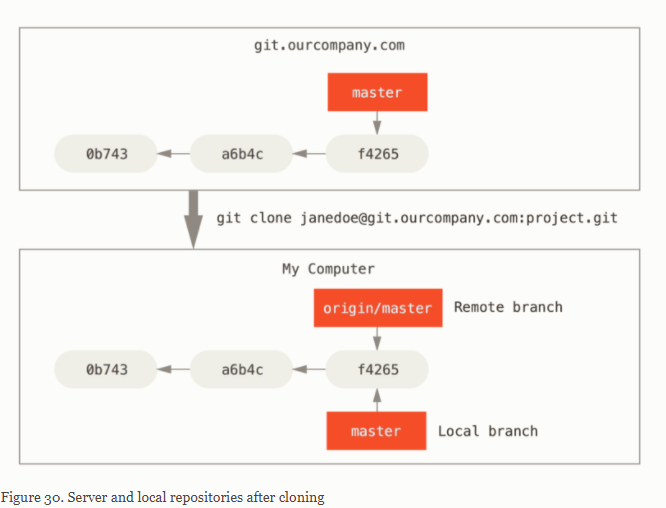
Remote branch:

Remote references are references (pointers) in your remote repositories, including branches, tags, and so on. You can get a full list of remote references explicitly with git ls-remote <remote>, or git remote show <remote> for remote branches as well as more information. Or use remote-tracking pointers.

Remote-tracking branches are references to the state of remote branches. They’re local references that you can’t move; Git moves them for you whenever you do any network communication, to make sure they accurately represent the state of the remote repository. Think of them as bookmarks, to remind you where the branches in your remote repositories were the last time you connected to them.

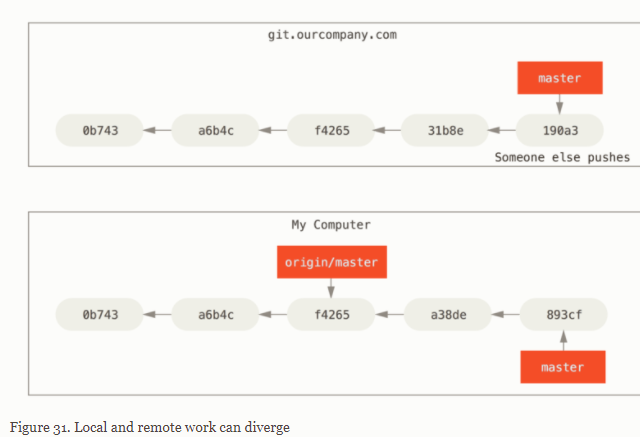
Remote-tracking branch names take the form <remote>/<branch>. For instance, if you wanted to see what the master branch on your origin remote looked like as of the last time you communicated with it, you would check the origin/master branch.

After cloning:

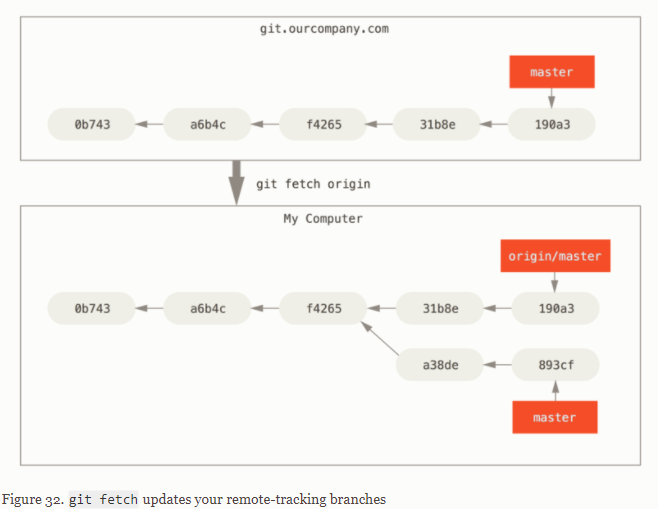


Just like the branch name “master” does not have any special meaning in Git, neither does “origin”. While “master” is the default name for a starting branch when you run git init which is the only reason it’s widely used, “origin” is the default name for a remote when you run git clone.

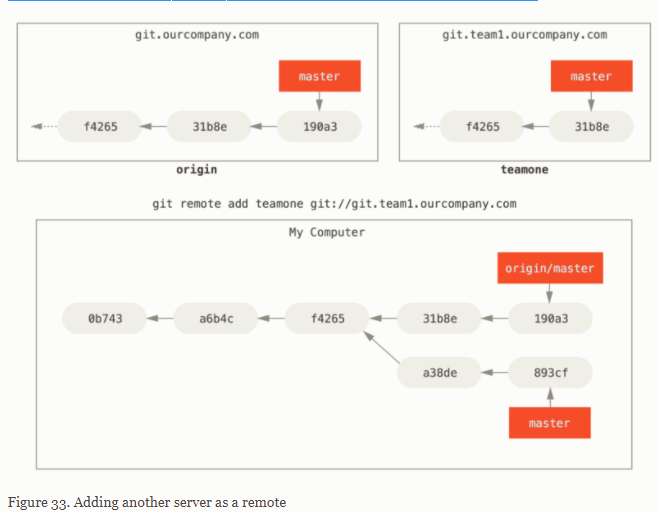
If you do some work on your local master branch, and, in the meantime, someone else pushes to git.ourcompany.com and updates its master branch, then your histories move forward differently. Also, as long as you stay out of contact with your origin server, your origin/master pointer doesn’t move.



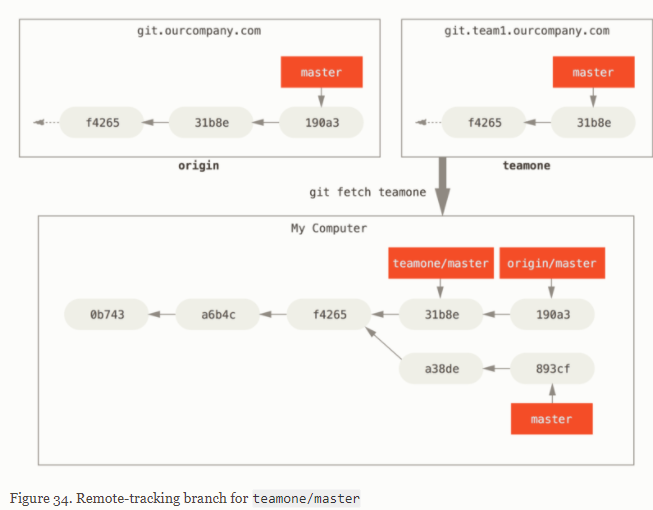
To synchronize your work with a given remote, you run a git fetch <remote> command (in our case, git fetch origin). This command looks up which server “origin” is (in this case, it’s git.ourcompany.com), fetches any data from it that you don’t yet have, and updates your local database, moving your origin/master pointer to its new, more up-to-date position.



Let’s assume you have another internal Git server that is used only for development by one of your sprint teams. This server is at git.team1.ourcompany.com. You can add it as a new remote reference to the project you’re currently working on by running the git remote add command as we covered in [Git Basics](https://git-scm.com/book/en/v2/ch00/ch02-git-basics-chapter). Name this remote teamone, which will be your shortname for that whole URL.



Now, you can run git fetch teamone to fetch everything the remote teamone server has that you don’t have yet. Because that server has a subset of the data your origin server has right now, Git fetches no data but sets a remote-tracking branch called teamone/master to point to the commit that teamone has as its master branch.



When you want to share a branch with the world, you need to push it up to a remote to which you have write access. Your local branches aren’t automatically synchronized to the remotes you write to — you have to explicitly push the branches you want to share. That way, you can use private branches for work you don’t want to share, and push up only the topic branches you want to collaborate on.

If you have a branch named serverfix that you want to work on with others, you can push it up the same way you pushed your first branch. Run git push <remote> <branch>:

$ git push origin serverfix

NOTE: **Don’t type your password every time**

If you’re using an HTTPS URL to push over, the Git server will ask you for your username and password for authentication. By default it will prompt you on the terminal for this information so the server can tell if you’re allowed to push.

If you don’t want to type it every single time you push, you can set up a “credential cache”. The simplest is just to keep it in memory for a few minutes, which you can easily set up by running git config --global credential.helper cache.

For more information on the various credential caching options available, see [Credential Storage](https://git-scm.com/book/en/v2/ch00/_credential_caching).

The next time one of your collaborators fetches from the server, they will get a reference to where the server’s version of serverfix is under the remote branch origin/serverfix:

$ git fetch origin

It’s important to note that when you do a fetch that brings down new remote-tracking branches, you don’t automatically have local, editable copies of them. In other words, in this case, you don’t have a new serverfix branch — you have only an origin/serverfix pointer that you can’t modify.

To merge this work into your current working branch, you can run git merge origin/serverfix. If you want your own serverfix branch that you can work on, you can base it off your remote-tracking branch:

$ git checkout -b serverfix origin/serverfix

This gives you a local branch that you can work on that starts where origin/serverfix is.

If you want to see what tracking branches you have set up, you can use the -vv option to git branch. This will list out your local branches with more information including what each branch is tracking and if your local branch is ahead, behind or both.

It’s important to note that these numbers are only since the last time you fetched from each server. This command does not reach out to the servers, it’s telling you about what it has cached from these servers locally. If you want totally up to date ahead and behind numbers, you’ll need to fetch from all your remotes right before running this. You could do that like this:

$ git fetch --all; git branch -vv

FETCH/PULL:

While the git fetch command will fetch all the changes on the server that you don’t have yet, it will not modify your working directory at all. It will simply get the data for you and let you merge it yourself.

However, there is a command called git pull which is essentially a git fetch immediately followed by a git merge in most cases. If you have a tracking branch set up as demonstrated in the last section, either by explicitly setting it or by having it created for you by the clone or checkout commands, git pull will look up what server and branch your current branch is tracking, fetch from that server and then try to merge in that remote branch.

Generally it’s better to simply use the fetch and merge commands explicitly as the magic of git pull can often be confusing.

Suppose you’re done with a remote branch — say you and your collaborators are finished with a feature and have merged it into your remote’s master branch (or whatever branch your stable codeline is in). You can delete a remote branch using the --delete option to git push. If you want to delete your serverfix branch from the server, you run the following:

$ git push origin --delete serverfix

To deleted remote branches

Conflicts:

Conflicts- You can have multiple people (yourself included) work on the same file and at the same place

- Git will either automatically resolve the differences or show you a conflict.- You won't know there will be a conflict, until you run `git pull` when the remote and your local computer is out of sync

- It will either auto merge the changes

- Or show you which files have a conflict.

- You open the file and manually look for the `>>>`, `===`, and `<<<` and fix the file until you are satisfied.

- You can then `git add` and `git commit` your changes and re `git push`

REBASING:

GENERAL WORKFLOW:

Make sure your master branch is up to date

-Go to the master branch: git checkout master

-Update your master branch: git pull origin master

Create a new branch (give it a useful name): git checkout -b my\_awesome\_task

-Give your branch a sensible name about what you are working on

-Don’t just give it your name or pid, nobody knows what you are doing

Go code and write commits!

-git add <my file>

-git commit -m 'look at all this cool stuff'

Push your branch git push origin my\_awesome\_task

push <WHERE> <WHAT>

Issue a pull request

Have someone (maintainers) review your code

-Does it follow coding style guides?

-Is the code “good”

No datasets are checked in

No loops when an apply or map function would suffice

There are functions for repetitive code

Are you checking your work?

Are the assumptions you are making about data tested in code?

If you are visually checking/inspecting your data to check your code, is there code written for your visual check?etc

(Maintainers) merge the pull request

-You can also checkoff a box that will also delete the branch on the remote

-Or delete the branch manually under the branches view

Go back to master: git checkout master

Pull down your merged code: git pull origin master

Delete your branch: git branch -d my\_awesome\_task (note it is a lower case d)

Clean up your branches: git fetch --prune

Accidently did work on master!!! what to do ?

Create a branch where you are now: git branch BRANCH\_NAME

Find the commit hash of where master is supposed to be

- git log --oneline --graph --decorate --all

Reset master to where you were: git reset --hard COMMIT\_HASH\_FOR\_MASTER

- make sure you do this on the master branch

Go to your branch: git checkout BRANCH\_NAME

Push your branch: git push origin BRANCH\_NAME

Create and merge the pull/merge request

Get changes from master on your branch

You are working on your branch, and the master branch changes (e.g., someone else gets their branch merged into master). The changes in master are also changes you need (e.g., the update to master is a function that you want to use), but you are still working on your branch and not ready to create a pull/merge request and/or merge your changes yet.

Go to your master branch: git checkout master

Get the new updates from master: git pull origin master

Go back to your branch: git checkout my\_branch

Rebase your branch against master: git rebase master

-You may or may not need to solve merge conflicts (see above)

Force push your branch: git push -f origin my\_branch

3.8.5 Remote server (e.g., GitLab, GitHub,

Tagging:

Git has the ability to tag specific points in a repository’s history as being important. Typically, people use this functionality to mark release points (v1.0, v2.0 and so on).

Git supports two types of tags: lightweight and annotated.

A lightweight tag is very much like a branch that doesn’t change — it’s just a pointer to a specific commit.

Annotated tags, however, are stored as full objects in the Git database.

They’re checksummed; contain the tagger name, email, and date; have a tagging message; and can be signed and verified with GNU Privacy Guard (GPG).

It’s generally recommended that you create annotated tags so you can have all this information; but if you want a temporary tag or for some reason don’t want to keep the other information,

lightweight tags are available too.

Annotated tag:

The easiest way is to specify -a when you run the tag command:

$ git tag -a v1.4 -m "my version 1.4"

$ git tag

v0.1

v1.3

v1.4

The -m specifies a tagging message, which is stored with the tag. If you don’t specify a message for an annotated tag, Git launches your editor so you can type it in.

You can see the tag data along with the commit that was tagged by using the git show command:

$ git show v1.4

Lightweight:

This is basically the commit checksum stored in a file — no other information is kept.

$ git tag v1.4-lw

$ git tag

v0.1

v1.3

v1.4

v1.4-lw

v1.5

$ git show v1.4-lw

Now, suppose you forgot to tag the project at v1.2, which was at the “Update rakefile” commit (hash:9fceb02).

You can add it after the fact. To tag that commit, you specify the commit checksum (or part of it) at the end of the command:

$ git tag -a v1.2 9fceb02

By default, the git push command doesn’t transfer tags to remote servers.

You will have to explicitly push tags to a shared server after you have created them. T

his process is just like sharing remote branches — you can run

git push origin <tagname> or git push origin --tags (if a lot of tags)

If you want to view the versions of files a tag is pointing to, you can do a git checkout of that tag,

although this puts your repository in “detached HEAD” state, which has some ill side effects:

$ git checkout v2.0.0

Detached HEAD:

You are in 'detached HEAD' state. You can look around, make experimental

changes and commit them, and you can discard any commits you make in this

state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may

do so (now or later) by using -c with the switch command.

In “detached HEAD” state, if you make changes and then create a commit, the tag will stay the same,

but your new commit won’t belong to any branch and will be unreachable, except by the exact commit hash.

Thus, if you need to make changes — say you’re fixing a bug on an older version, for instance — you will generally want to create a branch:

$ git checkout -b version2 v2.0.0

SIDE NOTE about ALIASES:

Git doesn’t automatically infer your command if you type it in partially. If you don’t want to type the entire text of each of the Git commands,

you can easily set up an alias for each command using git config. Here are a couple of examples you may want to set up:

$ git config --global alias.co checkout

$ git config --global alias.br branch

$ git config --global alias.ci commit

$ git config --global alias.st status

means that, for example, instead of typing git commit, you just need to type git ci.

$ git config --global alias.unstage 'reset HEAD --'

$git unstage fileA is equivalent to $ git reset HEAD -- fileA

brionm1@BELEJJ7YWT2N-1 MINGW64 ~/GitNYR (master)

$ cat README.md

# 2020-08-12-nyr\_git-michael

To read README.md file in Git directly

brionm1@BELEJJ7YWT2N-1 MINGW64 ~/GitNYR (master)

$ git restore README.md

brionm1@BELEJJ7YWT2N-1 MINGW64 ~/GitNYR (master)

$ git checkout 69ac873 README.md

Updated 1 path from 459ea8c

brionm1@BELEJJ7YWT2N-1 MINGW64 ~/GitNYR (master)

$ git checkout 8380ef4

Note: switching to '8380ef4'.

You are in 'detached HEAD' state. You can look around, make experimental

changes and commit them, and you can discard any commits you make in this

state without impacting any branches by switching back to a branch.

If you want to create a new branch to retain commits you create, you may

do so (now or later) by using -c with the switch command. Example:

git switch -c <new-branch-name>

- `git init`: create git repository (repo) in current folder

- only do this once per repo

- `git status`: tells you your current git status

- `git add`: add the file in the staging area

- `git commit`: type message explaining the changes & commit changes

- `git commit -m "MESSAGE"`: write a one line message and commit in 1 step

- `git log`: show you the history of commits

- `git lod --oneline`: show one line commits

- `git diff`: shows you the difference between current state and last commit

- `git diff --staged`: shows you the diff in the staging area

- `git checkout <HASH> <FILE>`: reverts <FILE> back to the state in <HASH>

- `git checkout HEAD~<NUM> <FILE>`: reverts <FILE> back to the state in HEAD~<NUM>

- `git checkout <HASH>`: move HEAD to <HASH> (moves entire state/folder)

- `git checkout master`: goes back to master from detached HEAD state

- `HEAD`: tells you where you are currently looking

- `git restore <FILE>`: throws away changes from <FILE>

- Can also use this to unstage from staging area

- Older version of git use git reset

brionm1@BELEJJ7YWT2N-1 MINGW64 ~/GitNYR ((5359584...))

$ ssh-keygen.exe

Generating public/private rsa key pair.

Enter file in which to save the key (/c/Users/brionm1/.ssh/id\_rsa):

Enter passphrase (empty for no passphrase):

Enter same passphrase again:

Your identification has been saved in /c/Users/brionm1/.ssh/id\_rsa

Your public key has been saved in /c/Users/brionm1/.ssh/id\_rsa.pub

The key fingerprint is:

SHA256:R0lv8PSfHUKZhc4GEOmfpXy4iyguozxZ12u9iYD1UpE brionm1@BELEJJ7YWT2N-1

The key's randomart image is:

+---[RSA 3072]----+

| o\* ..=. |

| .o B.= |

| E. o B... |

| .o . =..+|

| ...S + \* .o|

| .o.o. . \* . |

| o..o .o o |

|.o o .ooo.o. |

| oo +.oo +... |

+----[SHA256]-----+

brionm1@BELEJJ7YWT2N-1 MINGW64 ~/GitNYR ((5359584...))

$ cat /c/Users/brionm1/.ssh/id\_rsa.pub

ssh-rsa  brionm1@BELEJJ7YWT2N-1

brionm1@BELEJJ7YWT2N-1 MINGW64 ~/GitNYR ((5359584...))

$ git remote add origin git@github.com:MichaelBrionTakeda/GitNYR.git

brionm1@BELEJJ7YWT2N-1 MINGW64 ~/GitNYR ((5359584...))

$ git remote -v

origin git@github.com:MichaelBrionTakeda/GitNYR.git (fetch)

origin git@github.com:MichaelBrionTakeda/GitNYR.git (push)

brionm1@BELEJJ7YWT2N-1 MINGW64 ~/GitNYR ((5359584...))

$ git push origin master

Warning: Permanently added the RSA host key for IP address '140.82.118.3' to the list of known hosts.

Enumerating objects: 19, done.

Counting objects: 100% (19/19), done.

Delta compression using up to 8 threads

Compressing objects: 100% (13/13), done.

Writing objects: 100% (19/19), 1.97 KiB | 403.00 KiB/s, done.

Total 19 (delta 6), reused 0 (delta 0), pack-reused 0

remote: Resolving deltas: 100% (6/6), done.

To github.com:MichaelBrionTakeda/GitNYR.git

\* [new branch] master -> master

brionm1@BELEJJ7YWT2N-1 MINGW64 ~/GitNYR ((d209cd2...))

$ git push origin master

Everything up-to-date

Manual conflict:

Auto-merging README.md

CONFLICT (content): Merge conflict in README.md

Automatic merge failed; fix conflicts and then commit the result.

brionm1@BELEJJ7YWT2N-1 MINGW64 ~/GitNYR (master|MERGING)

$ nano README.md 🡪 open text editor to manual fix the merge issue

brionm1@BELEJJ7YWT2N-1 MINGW64 ~/GitNYR (master|MERGING)

$ git add README.md

brionm1@BELEJJ7YWT2N-1 MINGW64 ~/GitNYR (master|MERGING)

$ git status

On branch master

All conflicts fixed but you are still merging.

(use "git commit" to conclude merge)

Changes to be committed:

modified: README.md

brionm1@BELEJJ7YWT2N-1 MINGW64 ~/GitNYR (master|MERGING)

$ git commit README.md

fatal: cannot do a partial commit during a merge. / need to use git commit command

brionm1@BELEJJ7YWT2N-1 MINGW64 ~/GitNYR (master|MERGING)

$ git commit

[master d298e22] Merge branch 'master' of github.com:MichaelBrionTakeda/GitNYR

brionm1@BELEJJ7YWT2N-1 MINGW64 ~/GitNYR (master)

$ git log --oneline --graph --all

\* d298e22 (HEAD -> master) Merge branch 'master' of github.com:MichaelBrionTakeda/GitNYR

|\

| \* d9750a2 (origin/master) remote title change

\* | 7cf136f local change title

\* | 4b750de Merge branch 'master' of github.com:MichaelBrionTakeda/GitNYR

|\|

| \* 483d17f changes git github wo synchro

\* | a3fc9c7 Change git without synchro github

|/

\* 9828ad2 add local and remote headers

\* d5b4935 truc truc

\* abc774d revert back to commit 1

\* 8380ef4 log and diff commands

\* 91862fd add new commands

\* f4795df To modify the git command explanation

\* feff535 mistake as I used ´´ instead of ``

\* 2fd65e8 Add git add and git commit commands in the readme files

\* 69ac873 talked about init and status for first commit

brionm1@BELEJJ7YWT2N-1 MINGW64 ~/GitNYR (master)

$ git status

On branch master

nothing to commit, working tree clean

* Issue merging solved!

brionm1@BELEJJ7YWT2N-1 MINGW64 ~/GitNYR (master)

$ git log --oneline

d298e22 (HEAD -> master, my\_first\_branch) Merge branch 'master' of github.com:MichaelBrionTakeda/GitNYR

d9750a2 (origin/master) remote title change

7cf136f local change title

4b750de Merge branch 'master' of github.com:MichaelBrionTakeda/GitNYR

a3fc9c7 Change git without synchro github

483d17f changes git github wo synchro

9828ad2 add local and remote headers

d5b4935 truc truc

abc774d revert back to commit 1

8380ef4 log and diff commands

91862fd add new commands

f4795df To modify the git command explanation

feff535 mistake as I used ´´ instead of ``

2fd65e8 Add git add and git commit commands in the readme files

69ac873 talked about init and status for first commit

brionm1@BELEJJ7YWT2N-1 MINGW64 ~/GitNYR (master)

$ git checkout my\_first\_branch

Switched to branch 'my\_first\_branch'

brionm1@BELEJJ7YWT2N-1 MINGW64 ~/GitNYR (my\_first\_branch)

$ git switch my\_first\_branch

Already on 'my\_first\_branch'

brionm1@BELEJJ7YWT2N-1 MINGW64 ~/GitNYR (my\_first\_branch)

$ git log --oneline

d298e22 (HEAD -> my\_first\_branch, master) Merge branch 'master' of github.com:MichaelBrionTakeda/GitNYR

d9750a2 (origin/master) remote title change

7cf136f local change title

4b750de Merge branch 'master' of github.com:MichaelBrionTakeda/GitNYR

a3fc9c7 Change git without synchro github

483d17f changes git github wo synchro

9828ad2 add local and remote headers

d5b4935 truc truc

abc774d revert back to commit 1

8380ef4 log and diff commands

91862fd add new commands

f4795df To modify the git command explanation

feff535 mistake as I used ´´ instead of ``

2fd65e8 Add git add and git commit commands in the readme files

69ac873 talked about init and status for first commit

brionm1@BELEJJ7YWT2N-1 MINGW64 ~/GitNYR (my\_first\_branch)

$ nano data/script.R

brionm1@BELEJJ7YWT2N-1 MINGW64 ~/GitNYR (my\_first\_branch)

$ git status

On branch my\_first\_branch

Untracked files:

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

data/script.R

nothing added to commit but untracked files present (use "git add" to track)

brionm1@BELEJJ7YWT2N-1 MINGW64 ~/GitNYR (my\_first\_branch)

$ git add .

warning: LF will be replaced by CRLF in data/script.R.

The file will have its original line endings in your working directory

brionm1@BELEJJ7YWT2N-1 MINGW64 ~/GitNYR (my\_first\_branch)

$ git status

On branch my\_first\_branch

Changes to be committed:

(use "git restore --staged <file>..." to unstage)

new file: data/script.R

brionm1@BELEJJ7YWT2N-1 MINGW64 ~/GitNYR (my\_first\_branch)

$ git commit -m "rscript"

[my\_first\_branch 11bc511] rscript

1 file changed, 1 insertion(+)

create mode 100644 data/script.R

brionm1@BELEJJ7YWT2N-1 MINGW64 ~/GitNYR (my\_first\_branch)

$ ls data

script.R

brionm1@BELEJJ7YWT2N-1 MINGW64 ~/GitNYR (my\_first\_branch)

$ git checkout master

Switched to branch 'master'

brionm1@BELEJJ7YWT2N-1 MINGW64 ~/GitNYR (master)

$ git push origin my\_first\_branch

Enumerating objects: 25, done.

Counting objects: 100% (23/23), done.

Delta compression using up to 8 threads

Compressing objects: 100% (15/15), done.

Writing objects: 100% (16/16), 1.68 KiB | 215.00 KiB/s, done.

Total 16 (delta 4), reused 0 (delta 0), pack-reused 0

remote: Resolving deltas: 100% (4/4), completed with 2 local objects.

remote:

remote: Create a pull request for 'my\_first\_branch' on GitHub by visiting:

remote: https://github.com/MichaelBrionTakeda/GitNYR/pull/new/my\_first\_branch

remote:

To github.com:MichaelBrionTakeda/GitNYR.git

\* [new branch] my\_first\_branch -> my\_first\_branch

brionm1@BELEJJ7YWT2N-1 MINGW64 ~/GitNYR (master)

$ cat README.md

# 2020-08-12 NYR Git Workshop

These are some commands and notes from workshop

## Local

- `git init`: create git repository (repo) in current folder

- only do this once per repo

- `git status`: tells you your current git status

other blabla

## Remote

Blablabla

change on github (and another change in git) without synchronizing :)

brionm1@BELEJJ7YWT2N-1 MINGW64 ~/GitNYR (master)

$ git push origin my\_first\_branch

Everything up-to-date

brionm1@BELEJJ7YWT2N-1 MINGW64 ~/GitNYR (master)

$ nano data/script.R

brionm1@BELEJJ7YWT2N-1 MINGW64 ~/GitNYR (master)

$ git status

On branch master

nothing to commit, working tree clean

brionm1@BELEJJ7YWT2N-1 MINGW64 ~/GitNYR (master)

$ git log --oneline

d298e22 (HEAD -> master) Merge branch 'master' of github.com:MichaelBrionTakeda/GitNYR

d9750a2 (origin/master) remote title change

7cf136f local change title

4b750de Merge branch 'master' of github.com:MichaelBrionTakeda/GitNYR

a3fc9c7 Change git without synchro github

483d17f changes git github wo synchro

9828ad2 add local and remote headers

d5b4935 truc truc

abc774d revert back to commit 1

8380ef4 log and diff commands

91862fd add new commands

f4795df To modify the git command explanation

feff535 mistake as I used ´´ instead of ``

2fd65e8 Add git add and git commit commands in the readme files

69ac873 talked about init and status for first commit

brionm1@BELEJJ7YWT2N-1 MINGW64 ~/GitNYR (master)

$ ls data

brionm1@BELEJJ7YWT2N-1 MINGW64 ~/GitNYR (master)

$ ls Data

brionm1@BELEJJ7YWT2N-1 MINGW64 ~/GitNYR (master)

$ ls -a data

./ ../ .gitignore .gitkeep

brionm1@BELEJJ7YWT2N-1 MINGW64 ~/GitNYR (master)

$ git pull origin master

remote: Enumerating objects: 1, done.

remote: Counting objects: 100% (1/1), done.

remote: Total 1 (delta 0), reused 0 (delta 0), pack-reused 0

Unpacking objects: 100% (1/1), 636 bytes | 63.00 KiB/s, done.

From github.com:MichaelBrionTakeda/GitNYR

\* branch master -> FETCH\_HEAD

d9750a2..0695008 master -> origin/master

Updating d298e22..0695008

Fast-forward

data/script.R | 1 +

1 file changed, 1 insertion(+)

create mode 100644 data/script.R

brionm1@BELEJJ7YWT2N-1 MINGW64 ~/GitNYR (master)

$ ls data/

script.R

brionm1@BELEJJ7YWT2N-1 MINGW64 ~/GitNYR (master)

$ git log --online --graph --all

fatal: unrecognized argument: --online

brionm1@BELEJJ7YWT2N-1 MINGW64 ~/GitNYR (master)

$ git log --oneline --graph --all

\* 0695008 (HEAD -> master, origin/master) Merge pull request #1 from MichaelBrionTakeda/my\_first\_branch

|\

| \* 11bc511 (origin/my\_first\_branch, my\_first\_branch) rscript

| \* d298e22 Merge branch 'master' of github.com:MichaelBrionTakeda/GitNYR

| |\

| |/

|/|

\* | d9750a2 remote title change

| \* 7cf136f local change title

| \* 4b750de Merge branch 'master' of github.com:MichaelBrionTakeda/GitNYR

| |\

| |/

|/|

\* | 483d17f changes git github wo synchro

| \* a3fc9c7 Change git without synchro github

|/

\* 9828ad2 add local and remote headers

\* d5b4935 truc truc

\* abc774d revert back to commit 1

\* 8380ef4 log and diff commands

\* 91862fd add new commands

\* f4795df To modify the git command explanation

\* feff535 mistake as I used ´´ instead of ``

\* 2fd65e8 Add git add and git commit commands in the readme files

\* 69ac873 talked about init and status for first commit

!

brionm1@BELEJJ7YWT2N-1 MINGW64 ~/GitNYR (master)

$ git branch -d my\_first\_branch

Deleted branch my\_first\_branch (was 11bc511).

brionm1@BELEJJ7YWT2N-1 MINGW64 ~/GitNYR (master)

$ git log --oneline --graph --all

\* 0695008 (HEAD -> master, origin/master) Merge pull request #1 from MichaelBrionTakeda/my\_first\_branch

|\

| \* 11bc511 (origin/my\_first\_branch) rscript

| \* d298e22 Merge branch 'master' of github.com:MichaelBrionTakeda/GitNYR

| |\

| |/

|/|

\* | d9750a2 remote title change

| \* 7cf136f local change title

| \* 4b750de Merge branch 'master' of github.com:MichaelBrionTakeda/GitNYR

| |\

| |/

|/|

\* | 483d17f changes git github wo synchro

| \* a3fc9c7 Change git without synchro github

|/

\* 9828ad2 add local and remote headers

\* d5b4935 truc truc

\* abc774d revert back to commit 1

\* 8380ef4 log and diff commands

\* 91862fd add new commands

\* f4795df To modify the git command explanation

\* feff535 mistake as I used ´´ instead of ``

\* 2fd65e8 Add git add and git commit commands in the readme files

\* 69ac873 talked about init and status for first commit

!

brionm1@BELEJJ7YWT2N-1 MINGW64 ~/GitNYR (master)

$ git checkout -b branch\_notes

Switched to a new branch 'branch\_notes'

brionm1@BELEJJ7YWT2N-1 MINGW64 ~/GitNYR (branch\_notes)

$ git log --online --graph --all

fatal: unrecognized argument: --online

brionm1@BELEJJ7YWT2N-1 MINGW64 ~/GitNYR (branch\_notes)

$ git log --oneline --graph --all

\* 0695008 (HEAD -> branch\_notes, origin/master, master) Merge pull request #1 from MichaelBrionTakeda/my\_first\_branch

|\

| \* 11bc511 (origin/my\_first\_branch) rscript

| \* d298e22 Merge branch 'master' of github.com:MichaelBrionTakeda/GitNYR

| |\

| |/

|/|

\* | d9750a2 remote title change

| \* 7cf136f local change title

| \* 4b750de Merge branch 'master' of github.com:MichaelBrionTakeda/GitNYR

| |\

| |/

|/|

\* | 483d17f changes git github wo synchro

| \* a3fc9c7 Change git without synchro github

|/

\* 9828ad2 add local and remote headers

\* d5b4935 truc truc

\* abc774d revert back to commit 1

\* 8380ef4 log and diff commands

\* 91862fd add new commands

!

brionm1@BELEJJ7YWT2N-1 MINGW64 ~/GitNYR (branch\_notes)

$ nano README.md

brionm1@BELEJJ7YWT2N-1 MINGW64 ~/GitNYR (branch\_notes)

$ git status

On branch branch\_notes

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git restore <file>..." to discard changes in working directory)

modified: README.md

no changes added to commit (use "git add" and/or "git commit -a")

brionm1@BELEJJ7YWT2N-1 MINGW64 ~/GitNYR (branch\_notes)

$ git add.

git: 'add.' is not a git command. See 'git --help'.

The most similar command is

add

brionm1@BELEJJ7YWT2N-1 MINGW64 ~/GitNYR (branch\_notes)

$ git add .

brionm1@BELEJJ7YWT2N-1 MINGW64 ~/GitNYR (branch\_notes)

$ git commit -m "exercice README"

[branch\_notes 59672fc] exercice README

1 file changed, 1 insertion(+)

brionm1@BELEJJ7YWT2N-1 MINGW64 ~/GitNYR (branch\_notes)

$ cat README.md

# 2020-08-12 NYR Git Workshop

These are some commands and notes from workshop

## Local

- `git init`: create git repository (repo) in current folder

- only do this once per repo

- `git status`: tells you your current git status

other blabla

encore blabla pour tester branch exercice

## Remote

Blablabla

change on github (and another change in git) without synchronizing :)

brionm1@BELEJJ7YWT2N-1 MINGW64 ~/GitNYR (branch\_notes)

$ git push origin branch\_notes

Enumerating objects: 5, done.

Counting objects: 100% (5/5), done.

Delta compression using up to 8 threads

Compressing objects: 100% (3/3), done.

Writing objects: 100% (3/3), 355 bytes | 355.00 KiB/s, done.

Total 3 (delta 1), reused 0 (delta 0), pack-reused 0

remote: Resolving deltas: 100% (1/1), completed with 1 local object.

remote:

remote: Create a pull request for 'branch\_notes' on GitHub by visiting:

remote: https://github.com/MichaelBrionTakeda/GitNYR/pull/new/branch\_notes

remote:

To github.com:MichaelBrionTakeda/GitNYR.git

\* [new branch] branch\_notes -> branch\_notes

brionm1@BELEJJ7YWT2N-1 MINGW64 ~/GitNYR (branch\_notes)

$ git log --oneline

59672fc (HEAD -> branch\_notes, origin/branch\_notes) exercice README

0695008 (origin/master, master) Merge pull request #1 from MichaelBrionTakeda/my\_first\_branch

11bc511 (origin/my\_first\_branch) rscript

d298e22 Merge branch 'master' of github.com:MichaelBrionTakeda/GitNYR

d9750a2 remote title change

7cf136f local change title

4b750de Merge branch 'master' of github.com:MichaelBrionTakeda/GitNYR

a3fc9c7 Change git without synchro github

483d17f changes git github wo synchro

9828ad2 add local and remote headers

d5b4935 truc truc

abc774d revert back to commit 1

8380ef4 log and diff commands

91862fd add new commands

f4795df To modify the git command explanation

feff535 mistake as I used ´´ instead of ``

2fd65e8 Add git add and git commit commands in the readme files

69ac873 talked about init and status for first commit

brionm1@BELEJJ7YWT2N-1 MINGW64 ~/GitNYR (branch\_notes)

$ git checkout master

Switched to branch 'master'

brionm1@BELEJJ7YWT2N-1 MINGW64 ~/GitNYR (master)

$ git log --oneline

0695008 (HEAD -> master, origin/master) Merge pull request #1 from MichaelBrionTakeda/my\_first\_branch

11bc511 (origin/my\_first\_branch) rscript

d298e22 Merge branch 'master' of github.com:MichaelBrionTakeda/GitNYR

d9750a2 remote title change

7cf136f local change title

4b750de Merge branch 'master' of github.com:MichaelBrionTakeda/GitNYR

a3fc9c7 Change git without synchro github

483d17f changes git github wo synchro

9828ad2 add local and remote headers

d5b4935 truc truc

abc774d revert back to commit 1

8380ef4 log and diff commands

91862fd add new commands

f4795df To modify the git command explanation

feff535 mistake as I used ´´ instead of ``

2fd65e8 Add git add and git commit commands in the readme files

69ac873 talked about init and status for first commit

brionm1@BELEJJ7YWT2N-1 MINGW64 ~/GitNYR (master)

$ git pull origin master

remote: Enumerating objects: 1, done.

remote: Counting objects: 100% (1/1), done.

remote: Total 1 (delta 0), reused 0 (delta 0), pack-reused 0

Unpacking objects: 100% (1/1), 635 bytes | 57.00 KiB/s, done.

From github.com:MichaelBrionTakeda/GitNYR

\* branch master -> FETCH\_HEAD

0695008..a137196 master -> origin/master

Updating 0695008..a137196

Fast-forward

README.md | 1 +

1 file changed, 1 insertion(+)

brionm1@BELEJJ7YWT2N-1 MINGW64 ~/GitNYR (master)

$ cat README.md

# 2020-08-12 NYR Git Workshop

These are some commands and notes from workshop

## Local

- `git init`: create git repository (repo) in current folder

- only do this once per repo

- `git status`: tells you your current git status

other blabla

encore blabla pour tester branch exercice

## Remote

Blablabla

change on github (and another change in git) without synchronizing :)

brionm1@BELEJJ7YWT2N-1 MINGW64 ~/GitNYR (master)

$ git branch -d branch\_notes

Deleted branch branch\_notes (was 59672fc).

brionm1@BELEJJ7YWT2N-1 MINGW64 ~/GitNYR (master)

$ git log --oneline

a137196 (HEAD -> master, origin/master) Merge pull request #2 from MichaelBrionTakeda/branch\_notes

59672fc (origin/branch\_notes) exercice README

0695008 Merge pull request #1 from MichaelBrionTakeda/my\_first\_branch

11bc511 (origin/my\_first\_branch) rscript

d298e22 Merge branch 'master' of github.com:MichaelBrionTakeda/GitNYR

d9750a2 remote title change

7cf136f local change title

4b750de Merge branch 'master' of github.com:MichaelBrionTakeda/GitNYR

a3fc9c7 Change git without synchro github

483d17f changes git github wo synchro

9828ad2 add local and remote headers

d5b4935 truc truc

abc774d revert back to commit 1

8380ef4 log and diff commands

91862fd add new commands

f4795df To modify the git command explanation

feff535 mistake as I used ´´ instead of ``

2fd65e8 Add git add and git commit commands in the readme files

69ac873 talked about init and status for first commit

brionm1@BELEJJ7YWT2N-1 MINGW64 ~/GitNYR (master)

$ git fetch --prune

Warning: Permanently added the RSA host key for IP address '140.82.113.3' to the list of known hosts.

brionm1@BELEJJ7YWT2N-1 MINGW64 ~/GitNYR (master)

$ git log --oneline

a137196 (HEAD -> master, origin/master) Merge pull request #2 from MichaelBrionTakeda/branch\_notes

59672fc (origin/branch\_notes) exercice README

0695008 Merge pull request #1 from MichaelBrionTakeda/my\_first\_branch

11bc511 (origin/my\_first\_branch) rscript

d298e22 Merge branch 'master' of github.com:MichaelBrionTakeda/GitNYR

d9750a2 remote title change

7cf136f local change title

4b750de Merge branch 'master' of github.com:MichaelBrionTakeda/GitNYR

a3fc9c7 Change git without synchro github

483d17f changes git github wo synchro

9828ad2 add local and remote headers

d5b4935 truc truc

abc774d revert back to commit 1

8380ef4 log and diff commands

91862fd add new commands

f4795df To modify the git command explanation

feff535 mistake as I used ´´ instead of ``

2fd65e8 Add git add and git commit commands in the readme files

69ac873 talked about init and status for first commit

brionm1@BELEJJ7YWT2N-1 MINGW64 ~/GitNYR (master)

$