Git Tutorial and Helpful commands

Table of contents:

Git is ... a Persistent Map

The secret to git is not about knowing the commands, either proclaim or plumbing, instead the secret to git is knowing the conceptual model behind the git.

Git Is..

…a Persistent Map

Git commands:

Proclaim commands	***Plumbing commands (Low level)***
• git add • git commit • git push • git pull • git branch • git checkout • git merge • git rebase • …	• git cat file • git hash-object • git count-objects • …

On its core, git its just a map, a table with keys and values. The values are just sequences of bytes, for example the content of a text file, etc. Git calculates the key for the value with SHA1 algorithm.

SHA1s are 20 de bytes in hexazecimal format, so 40 de hexazecimal numbers. This will be gits key to store the value on the map. We can also calculate the hash in the command line: echo "Apple Pie" | git hash-object --stdin

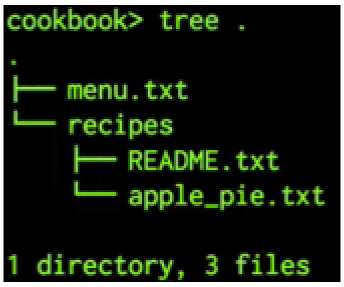
Every object in a git repository has its own SHA1.

- git init -> create git repository
- echo "Apple Pie" | git hash-object --stdin -w (it will print the hash and save the content)
- In the folder ".git/objects/" there are folders with numeric name. The name of this folders represents the first two digitis of the hash value, the rest of the has value is the name of the containing files.
- git cat-file hash_value -t (will return the type)
- git cat-file hash_value -p (pretty printing)

So, git is able to take any piece of content, generate a key for it (a SHA1), and then to save the content of it in a repository as a map.

Git, the stupid content tracker ...

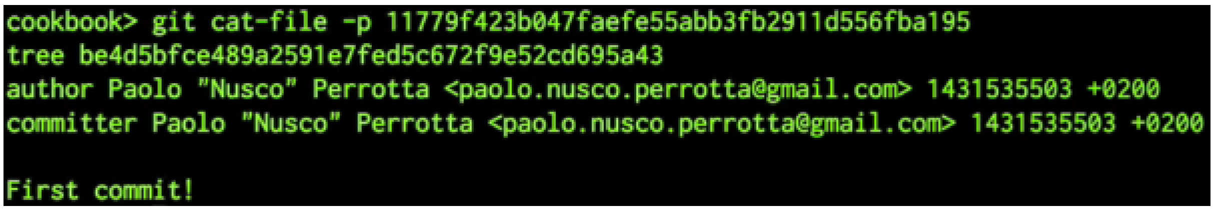
We need an example so, he created this:



```
cookbook> tree .
.
├── menu.txt
└── recipes
    ├── README.txt
    └── apple_pie.txt

1 directory, 3 files
```

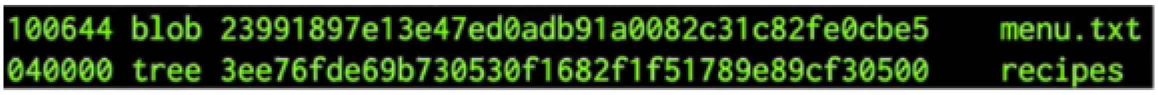
- git init -> create git repository
- git status -> to see all untracked or modified files
- To commit a file you have to put it in a station area, like a "launchpad state"
- git add file_path -> to put it in launchpad state
- git commit -m "First commit!" -> "-m" - so you can add a message to commit
- git log -> so you can see log informations, commits, hashes, etc

A commit is compressed just like a blob:



```
cookbook> git cat-file -p 11779f423b047faefe55abb3fb2911d556fba195
tree be4d5bfce489a2591e7fed5c672f9e52cd695a43
author Paolo "Nusco" Perrotta <paolo.nusco.perrotta@gmail.com> 1431535503 +0200
committer Paolo "Nusco" Perrotta <paolo.nusco.perrotta@gmail.com> 1431535503 +0200

First commit!
```

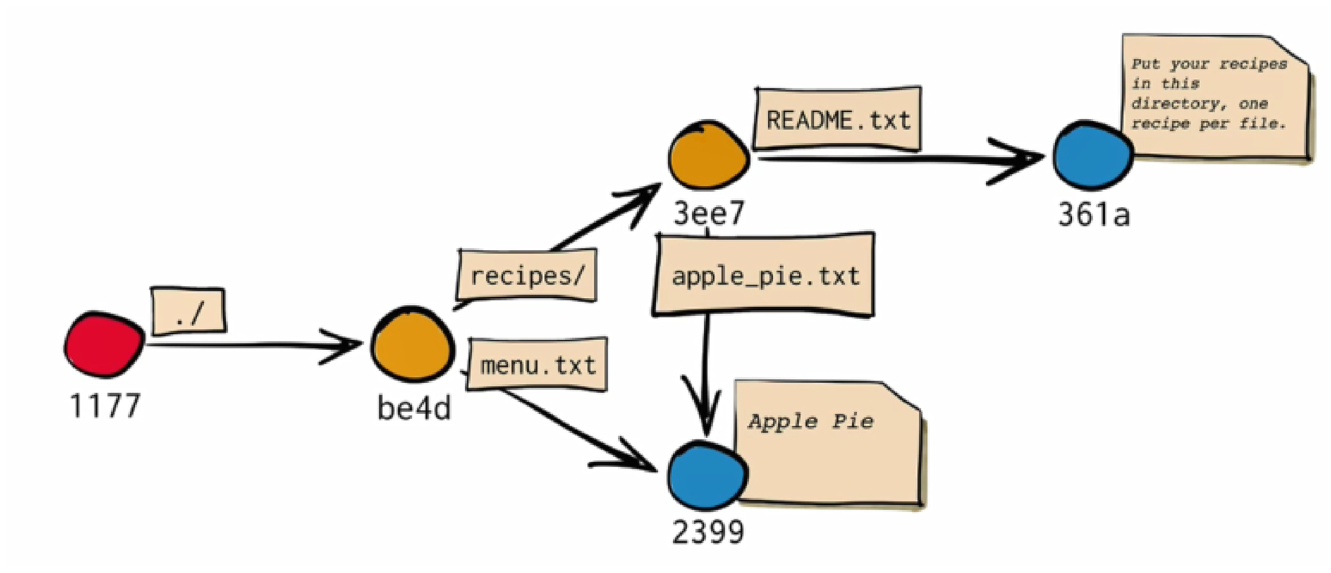
- commit it’s a simple and short piece of text that is generated by git and it stores it to the project database.
- tree is a directory stored in git



```
100644 blob 23991897e13e47ed0adb91a0082c31c82fe0cbe5    menu.txt
040000 tree 3ee76fde69b730530f1682f1f51789e89cf30500    recipes
```

- blob is a file stored in git

The commit points to a tree, root. This tree points to a blob menu.txt and another tree. This tree points to a blob README_FILE.txt and it also points to a blob apple_pie.txt. This blob is the same as menu.txt blob bacause the contents of the file is the same.



So, to be tricky:

- A blob is not a file, it’s the content of a file
- The file name and permissions are not stored in the blob, but in the tree

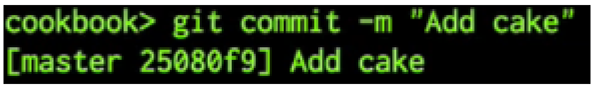
Git is ... a Stupid Content Tracker

Git Is..

…a Stupid Content Tracker

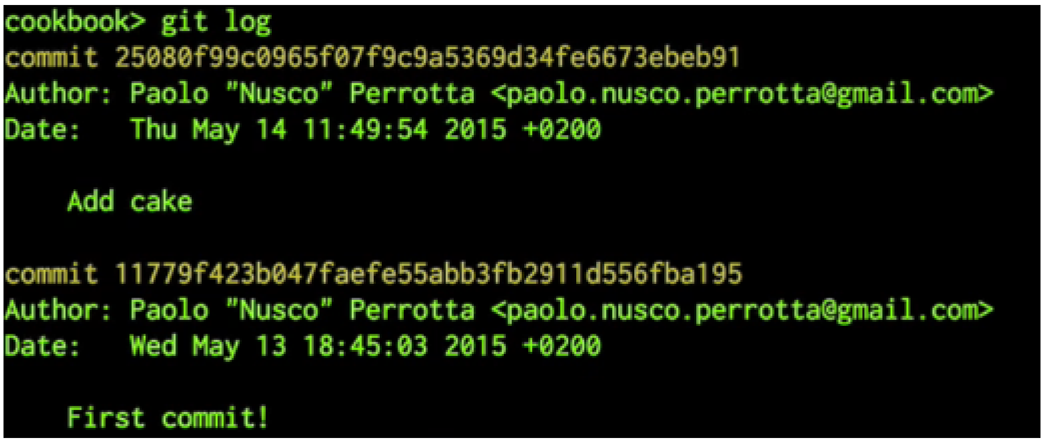
Let's talk about versioning:

Let's assume that we modify menu.txt. After that we will stage it (with git add) and then we commit.



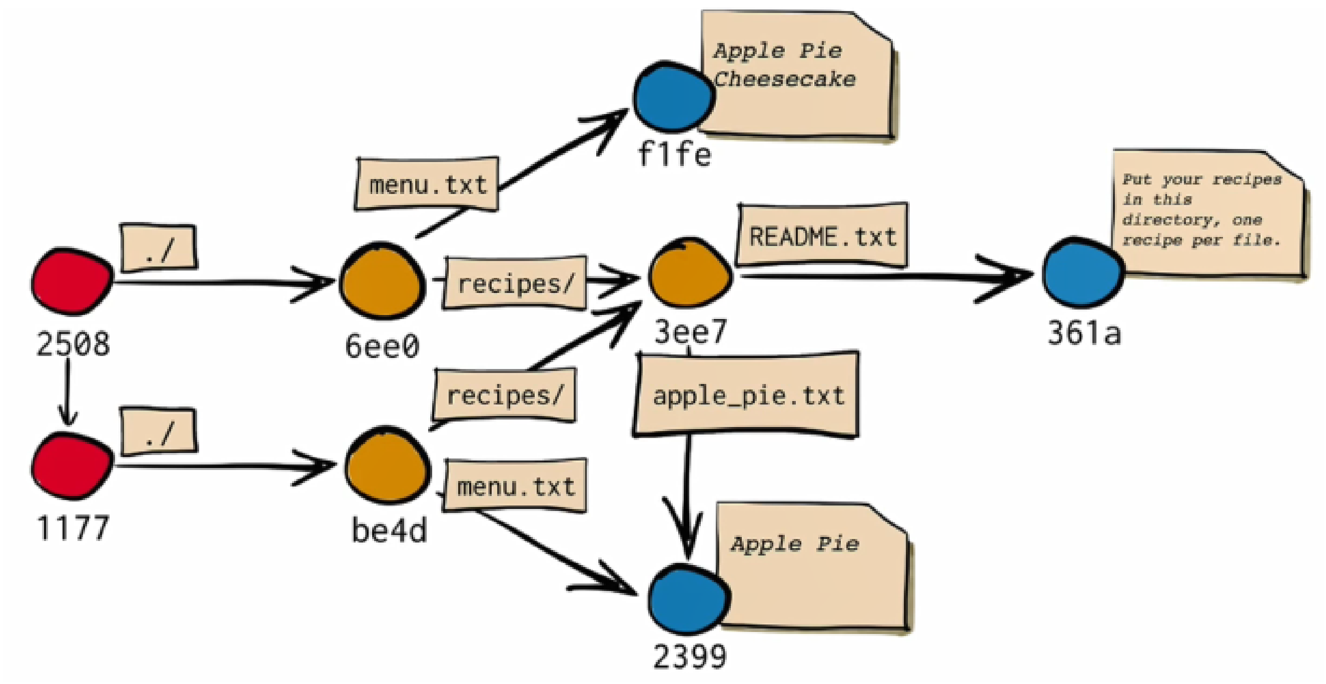
```
cookbook> git commit -m "Add cake"
[master 25080f9] Add cake
```

Then our log will look like this:



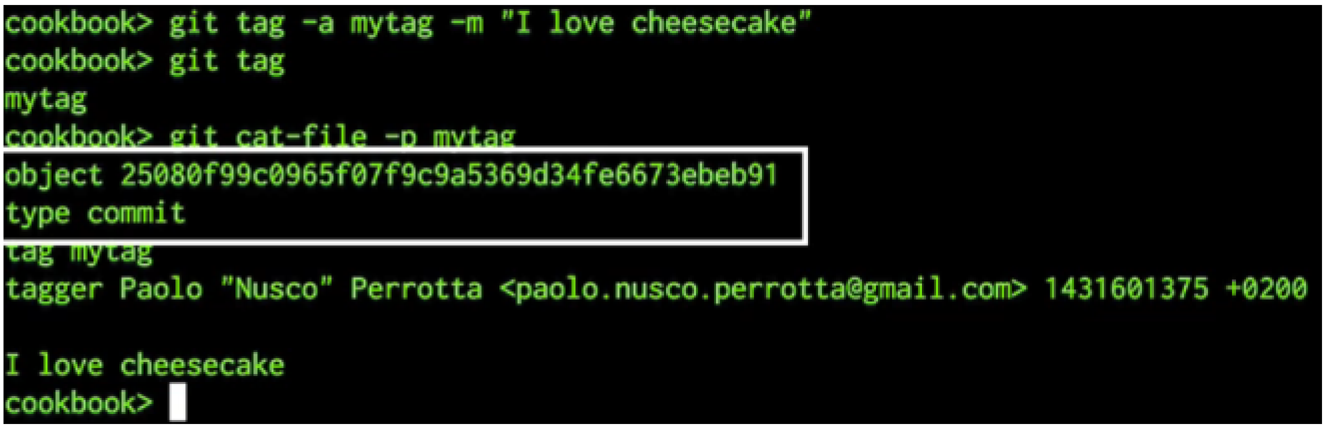
```
cookbook> git log
commit 25080f99c0965f07f9c9a5369d34fe6673ebeb91
Author: Paolo "Nusco" Perrotta <paolo.nusco.perrotta@gmail.com>
Date:   Thu May 14 11:49:54 2015 +0200

    Add cake

commit 11779f423b047faefe55abb3fb2911d556fba195
Author: Paolo "Nusco" Perrotta <paolo.nusco.perrotta@gmail.com>
Date:   Wed May 13 18:45:03 2015 +0200

    First commit!
```

In this case first commit is a parent of the "Add cake" commit. Most of the commits have a parent, the first one is an exception.

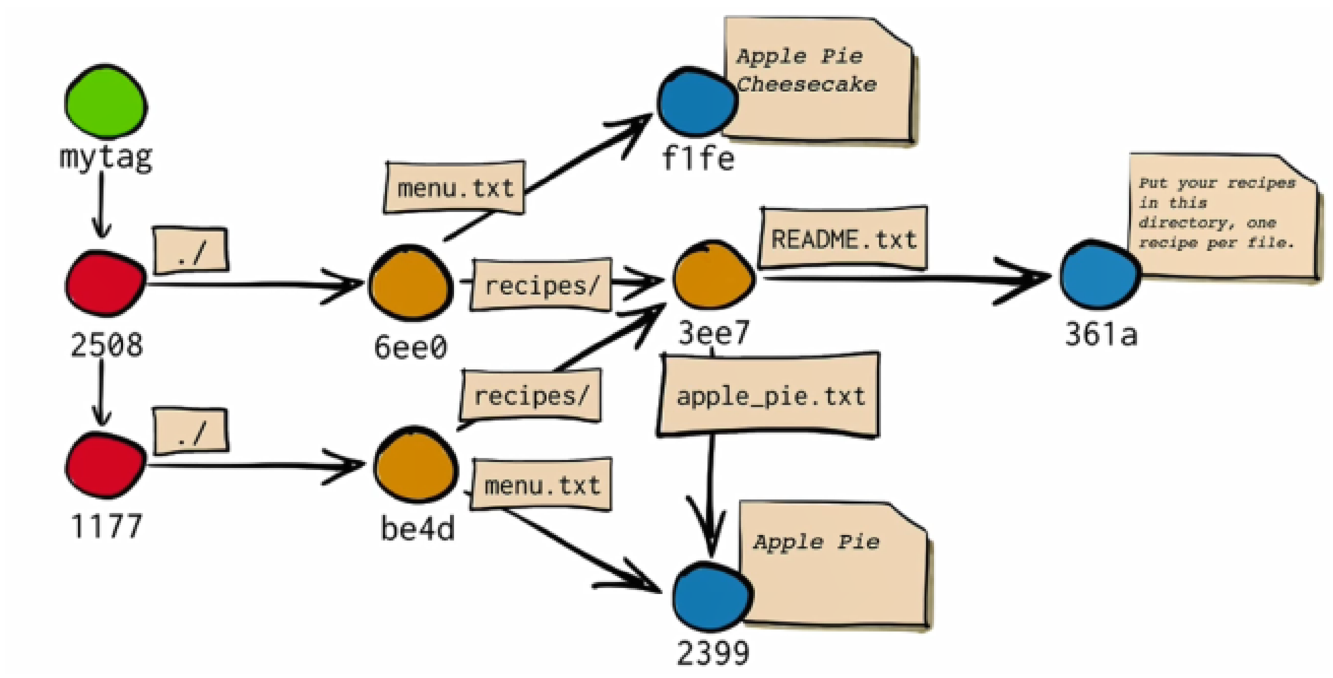


Git creates a new blob for the modified file menu.txt and it uses the same recipes tree because nothing has changed inside of it.

Tags: A simple object attached to an object



```
cookbook> git tag -a mytag -m "I love cheesecake"
cookbook> git tag
mytag
cookbook> git cat-file -p mytag
object 25080f99c0965f07f9c9a5369d34fe6673ebeb91
type commit
tag mytag
tagger Paolo "Nusco" Perrotta <paolo.nusco.perrotta@gmail.com> 1431601375 +0200

I love cheesecake
cookbook> 
```



Recap: Git Objects

- Blobs (files content)
- Trees (equivalent to directories)
- Commits
- Annotated Tags

Git is ... a Revision Control System

Git Is..

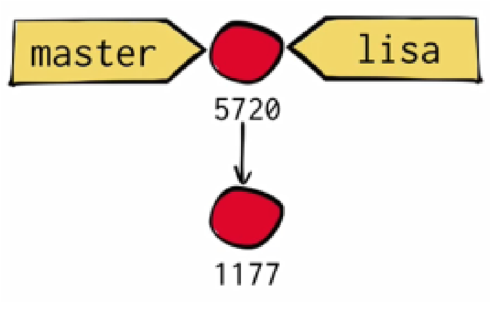
…a Revision Control System

Git creates for us at our first commit a default branch, the master branch.

The folder where you can find the branches is ".git/refs/heads".

A branch is just a reference to a commit.

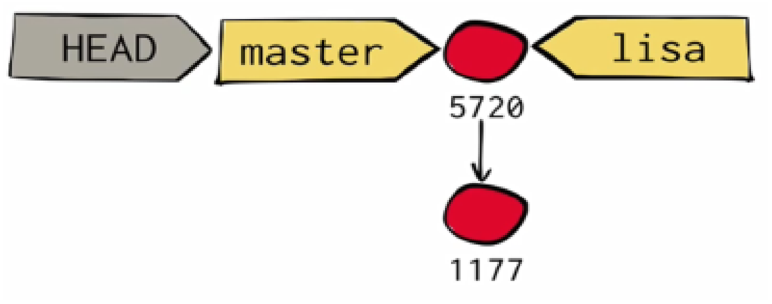
- git branch name -> creates a new branch
- git branch -> shows all branches



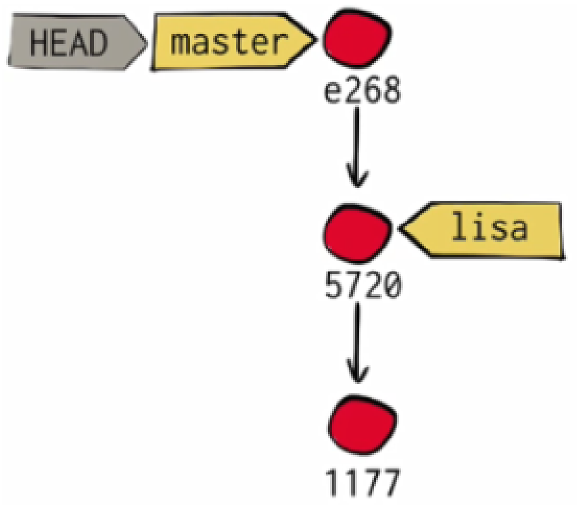
Git branch command shows all branches. The branch with * is the current working branch.

How git knows the branch that we are currently working on ?

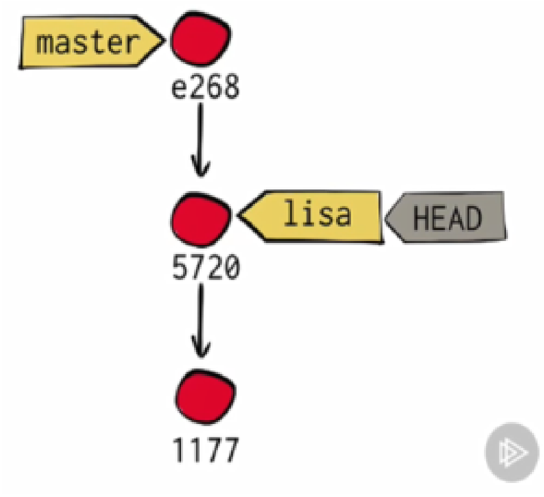
Well, there is a file ".git/HEAD" which tells him that. This file contains a reference to another file from "refs/heads/". Also, the current working branch is called HEAD.



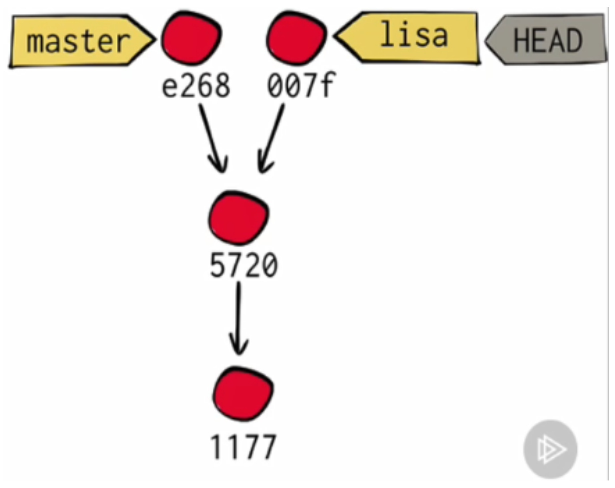
After a new commit:



- git checkout lisa -> Head will reference lisa, so lisa becomes the new currently working branch



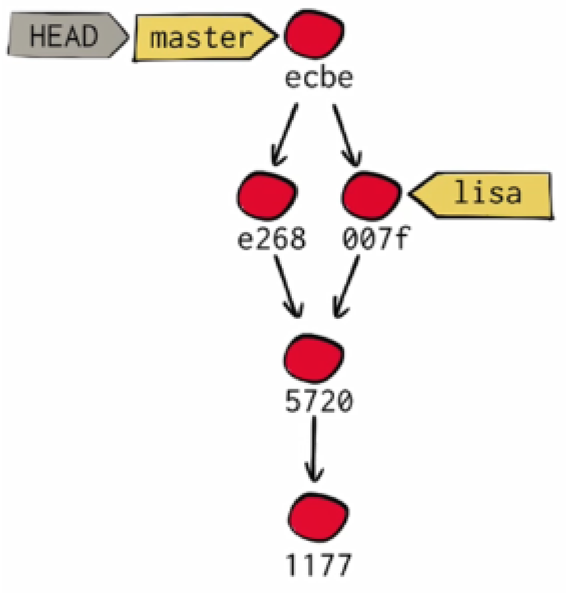
After new commit on lisa branch the scheme will look like this:



Let's merge!

You can merge lisa branch into master by having the master as HEAD and executing the command:

- git merge lisa



References

We already know that there are 4 types of objects in git:

- Blobs (content of a file)
- Trees (equivalent to directories)
- Commits
- Tags

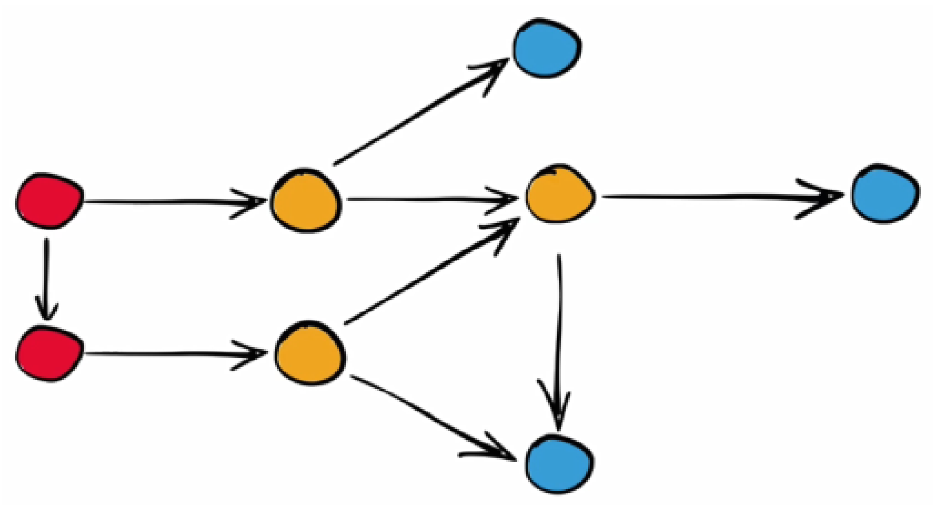
There are two types of references in git.

- References to track history
- References to track content

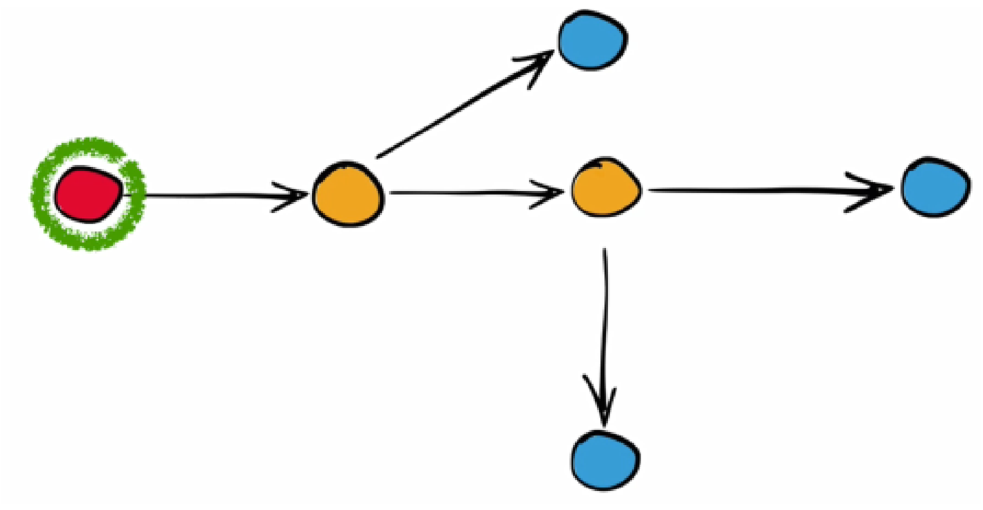
The references between two commits are used to track history, all the other references are used to track content.

When you checkout something, git doesn’t care about history. It doesn’t look at the way that commits connect to eachother, it just cares about trees and blobs.

For example:



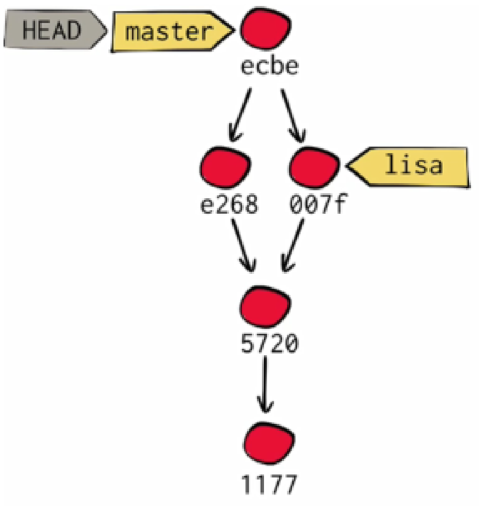
If look at one commit, it just look for the trees and blobs that can be reached by it.



This is the entire state of the project, and this is how you can travel though time with git.

Fast-forward - Merging without merging

For example:

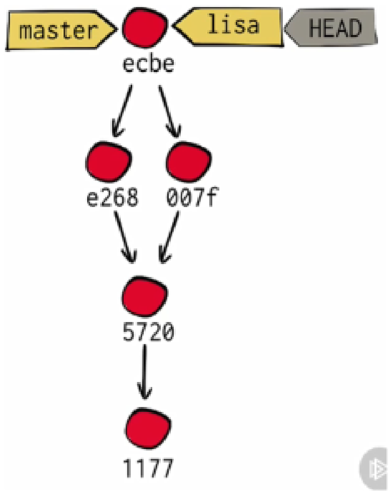


Lets say we want to merge master to lisa.

First we have to git checkout lisa and then git merge master.

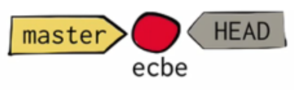
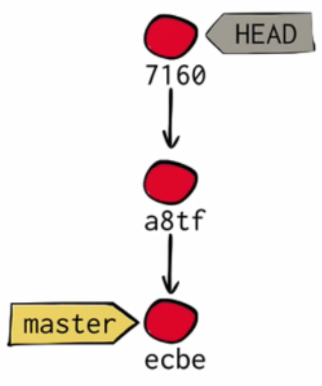
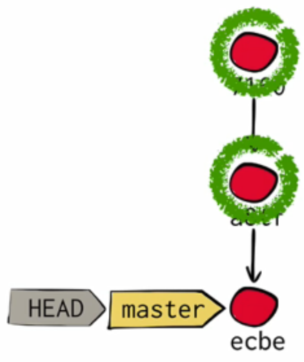
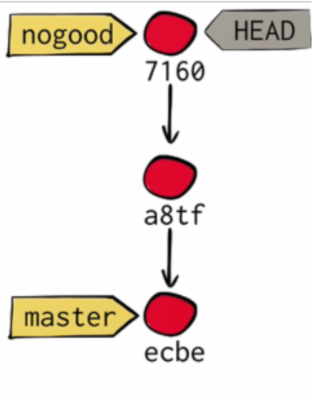
So what are we trying to achieve is to have a branch that has the latest files of master and the latest files of lisa. But we already have a branch like that, the branch that master points at.

When we will execute this command, git will figure it out and will not create a new commit for the merge. It will simply move lisa to point at the same branch as master, this way we will not have two commits that references to the same data (trees and blobs).



Loosing your head state

You can be in 'detached HEAD" state if you git checkout a commit. If you do that, there will be no branch active, the HEAD will point at the commit only.
In this 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.

			
• git checkout ecbe • git commit • git commit	• git checkout master	After this checkout, git will see that those commits are unreachable and it will consider them garbage, so it will delete them.	If you want you can save these commits if you want by making a new branch: • git branch nogood Or • git checkout -b nogood

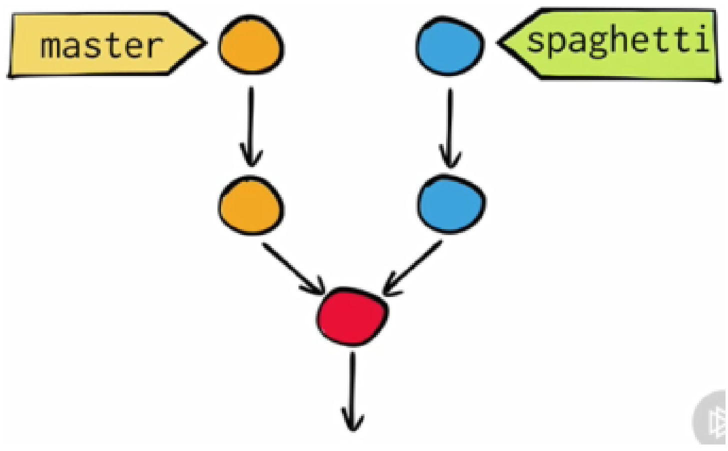
Three Rules

1 . The current branch tracks new commits

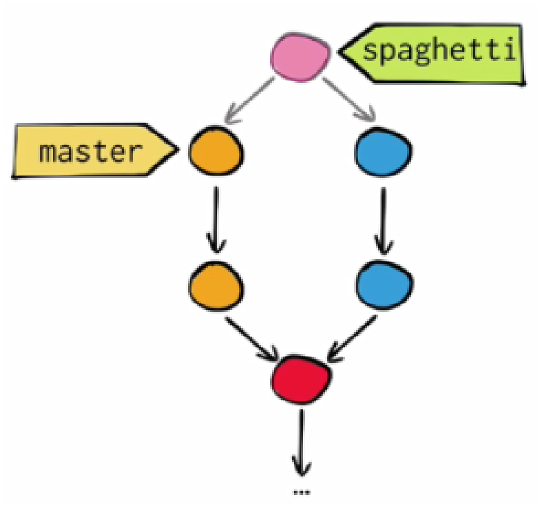
2. When you move to another commit, Git updates your working directory

3. Unreachable objects are garbage collected

Rebasing Made Simple

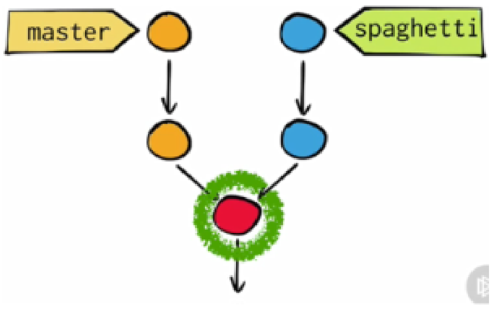


Let' say we want to put the content of the spaghetti and master branches together. We already know how to do that, merging.

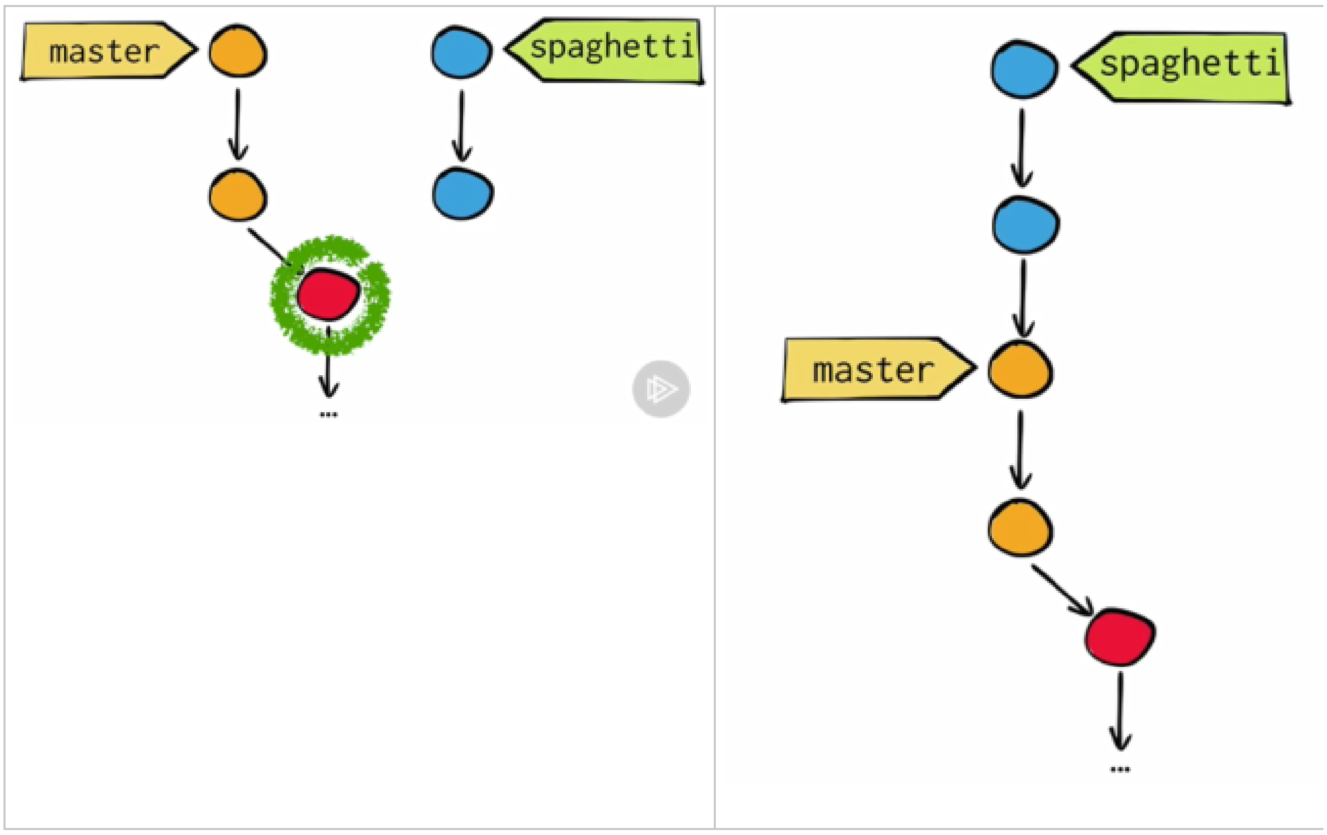


But there is another way to do that: Rebase

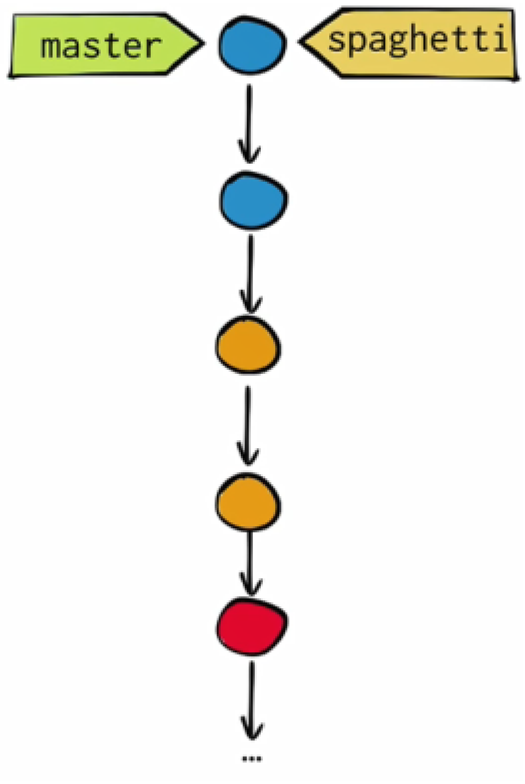
If we git rebase master git will look to the first commit in spaghetti that is also commit in master. So it's this commit here:



All the data after that commit its shared so its not relevant here for rebasing. So git is detaching the entire spaghetti branch from this commit and it moves it on top of master.



If we checkout master and rebase or merge it now there will be no difference. And our diagram will look like this:

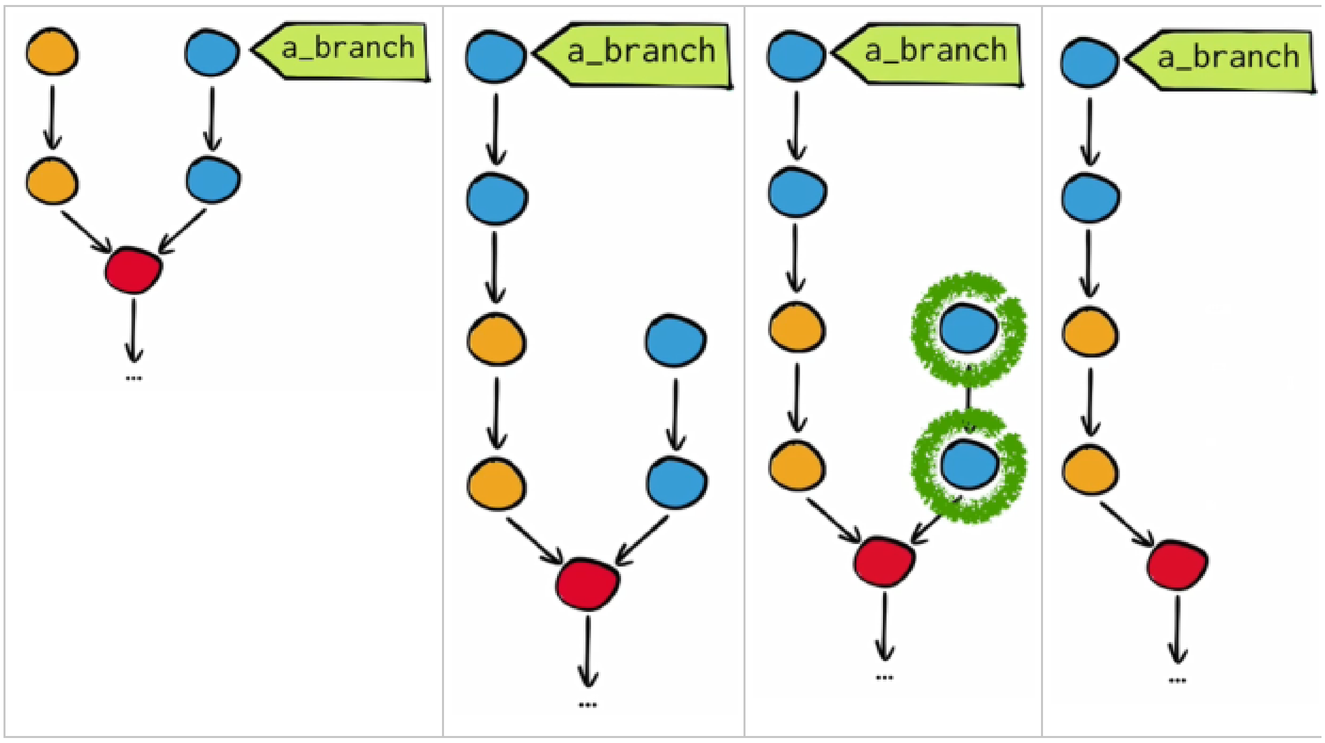


An illusion of movement

The whole story about rebases is not so simple. The truth is that git doesn't "move" the branch, this is impossible in git because, remember, those are database elements.

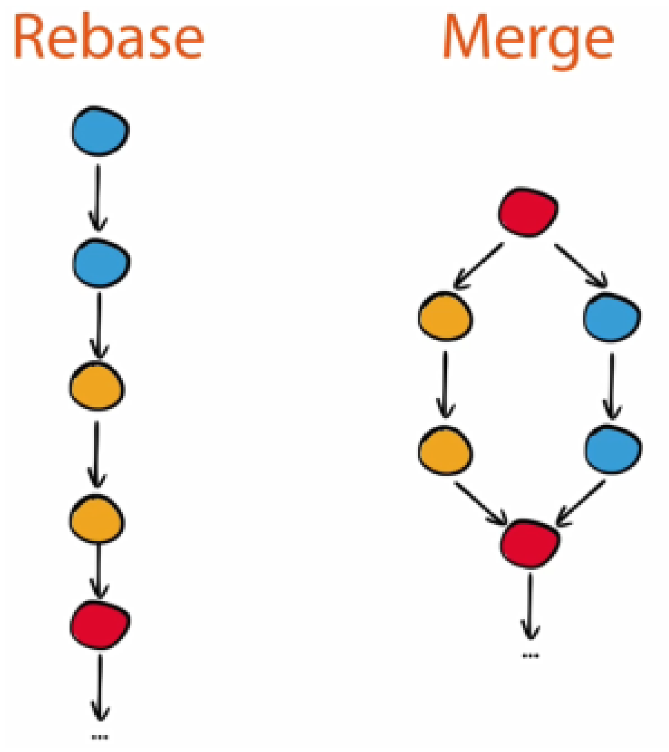
What git is really doing when you rebase is copying the uncommon commits on the top of the branch like new files, with new SHA1 in the database directory. After that, git looks for unreachable objects in the database, so it doesn't waste disk memory, and deletes them.

So rebasing is creating new commits.



The Trade-offs of Merges & Rebases

Why do rebase even exists ? Why do we have both of them if they are doing something similar? Well, the difference is that they have different trade-offs.



Merge: Preserves history exactly how it happened.

Rebase: Refactor history. So that it is always nice to look at.

At the rebase, it looks like the yellow commits were made first and then the blue ones were build up on top of them. But it's a lie. They were committed in parallel. So, rebasing is changing history.

When in doubt, just merge!

Tags in Brief

Git has two kinds of tags:

- git tag <tag_name>   -> Annotated tags
- git tag -a <tag_name> -> Non-Annotated tags (lightweight tag)

The tags can be found in ".git/refs/tags". A tag is a reference to an object.

What's the difference between a branch and a tag? Well, branches move, tags don't.

Recap:

Branches, Merges, Rebases, Tags - these are the main features that turn git from "A Stupid Content Tracker" into a "Revision Control System".

Git is ... a Distributed Revision Control System

Git Is..

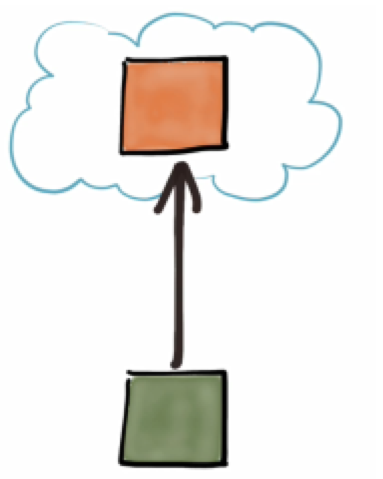
…a Distributed Revision Control System

So far we imagined that there is only one computer in the world. Let's see what happens if we use Git the way it's used in practice to share project across multiple computers.

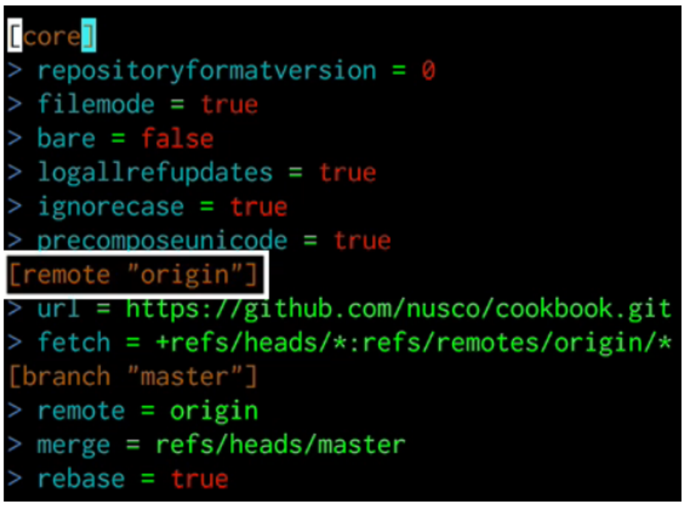
To get a project from git we can use:

- git clone <link_to_project>

This command will copy the project on your local machine with a single branch, master. If you want to work with the other branches you need to write different commands.



In ".git/config" we can find some useful information. Each inter-repository can remember information about other copies of the same repository, each other copy is called a remote. You can define as many remotes as you want but when you clone a project git immediately defines a default remote.



```
[core]
> repositoryformatversion = 0
> filemode = true
> bare = false
> logallrefupdates = true
> ignorecase = true
> precomposeunicode = true
[remote "origin"]
> url = https://github.com/nusco/cookbook.git
> fetch = +refs/heads/*:refs/remotes/origin/*
[branch "master"]
> remote = origin
> merge = refs/heads/master
> rebase = true
```

Here is the remote "origin" and it points to the URL that we cloned the project from.

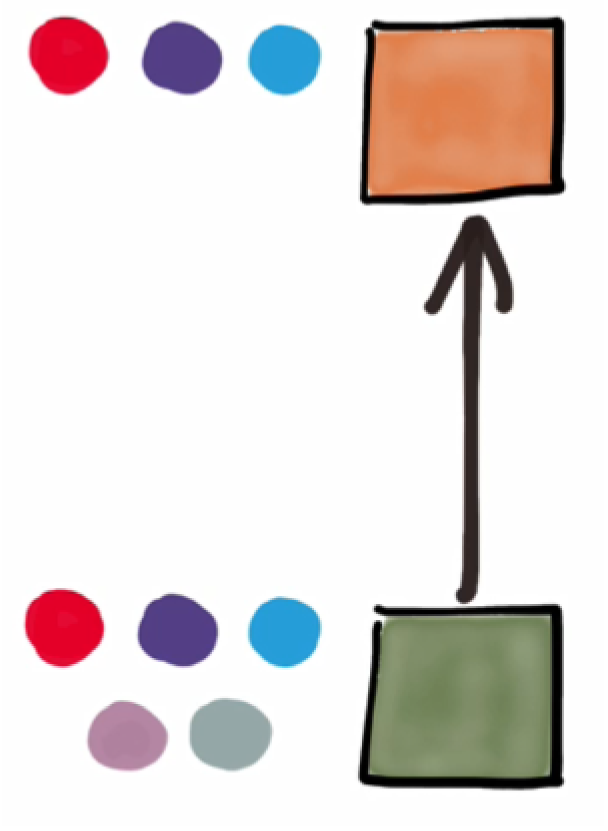
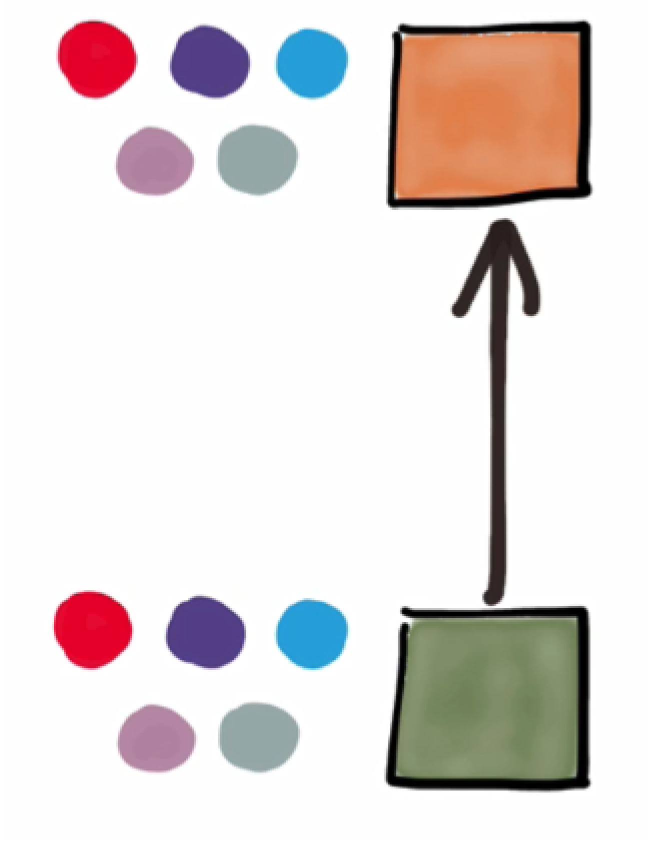
- git branch -> will show only the local branches
- git branch --all -> will show all the branches, including the ones from the remote, and the current position of HEAD on the remote

These can be found at ".git/refs/remotes/origin". Branches that are not in this folder are packed in the file ".git/packed-refs".

- git show-ref master -> will list all master branches SHA1, and their location.

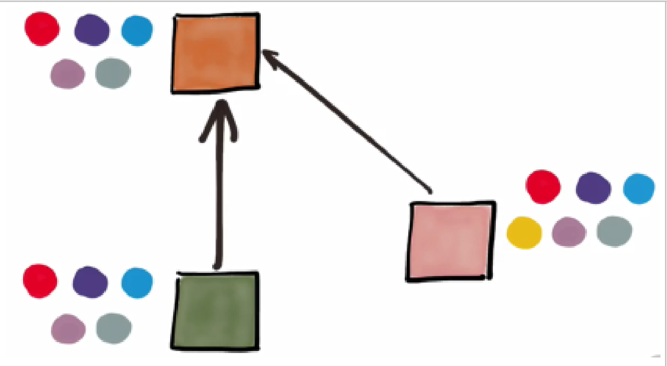
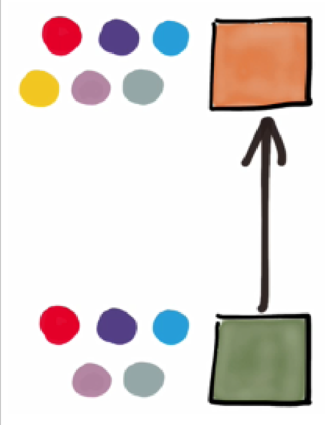
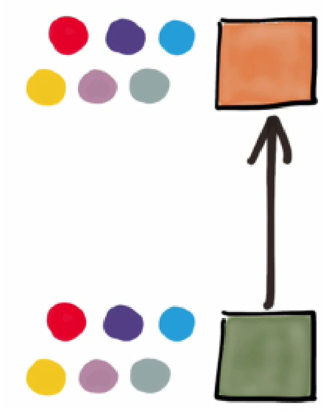
The Joy of Pushing

To synchronize the projects you can use push command to upload the changes you've made in the project.

	
• git push	

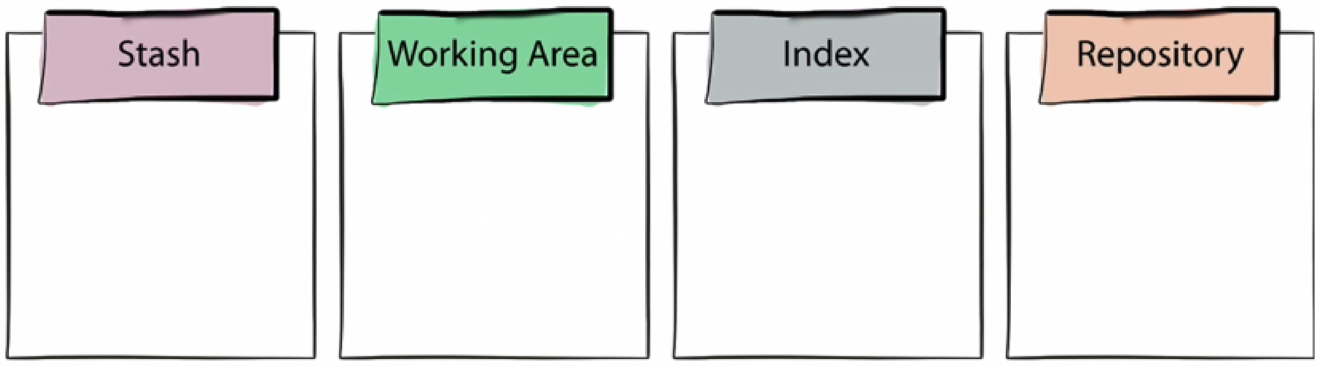
The Chore of Pulling

Let's say that someone else is pushing something to the git project. You can get the new version of the project with pull command. This command will synchronize the repos.

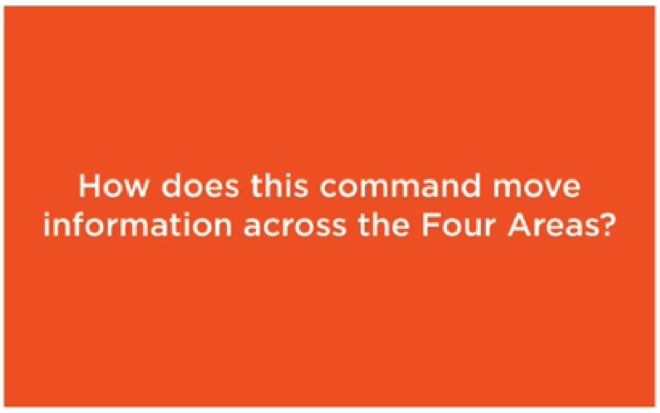
		
• git push	• git pull	

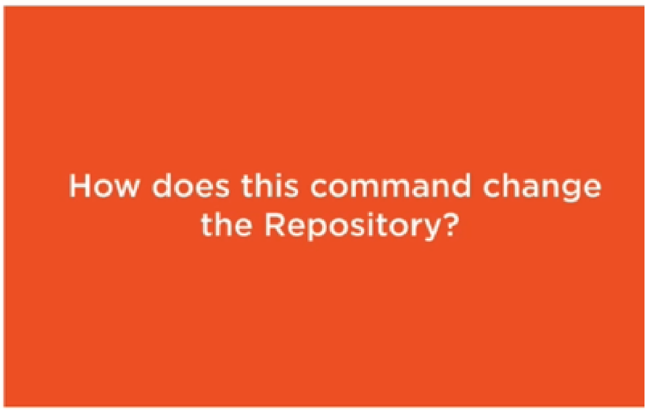
The Four Areas:

- Working Area (your project directory, file system)
- Repository (contains the entire history of the project)
- Index (the place where you put your files before a commit)
- Stash (temporary storage area)



The Two Questions:


How does this command move information across the Four Areas?


How does this command change the Repository?

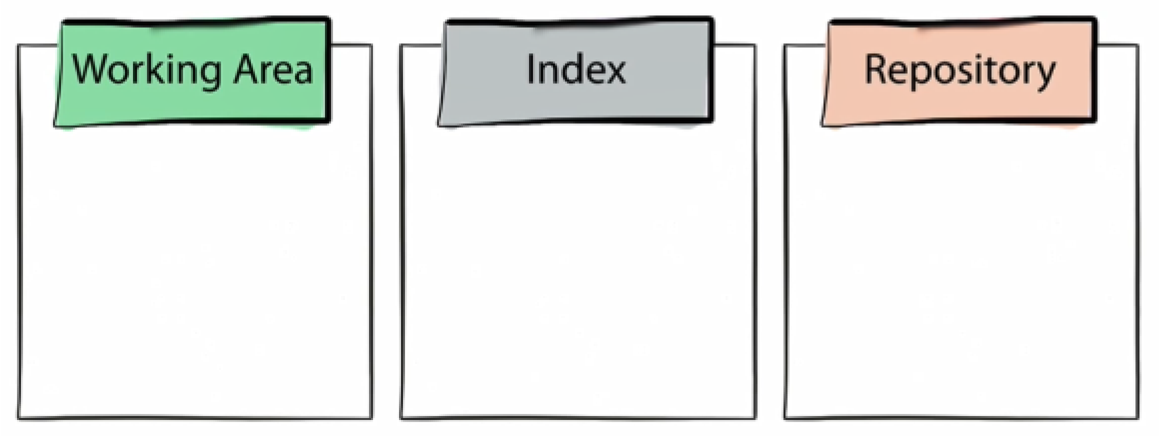
The Working Area

Is the project directory of your file system. This is where you work, you edit your files and test your code, etc.

The working area is important to you, git doesn't care as much about it. For git, the working area is a very temporary place. Git will avoid destroying your data, but we will see a few commands that does that. Don't assume that your data is safe until you have committed it. Once you commit, the git stores data in what it considers the really important area, the Repository.

The Index

Pretty much every versioning system out there has a Working Area and a Repository, but the Index is unique to git. Or, at least git is the only versioning system that allows you to modify it directly. You can imagine the Index as something that stands between the Working Area and the Repository. You generally don't move the data from the Working Area to the Repository directly. You go through the index. That's why the index is also called the Staging Area, because you stage your changes by adding them from the working area to the index, and then you commit the changes from the index to the repository.



If git status says that there is nothing to commit, then you are in clean status.

Helpful Git commands

Usual git commands

git log -2	*Shows the last two commits*
git log -2 --oneline	*Shows the last two commits, just one line for each commit*
git log -2 --oneline | wc	*Shows the last two commits, word count*
git log filename	*Shows the commits regarding the filename*
git show commit-id	*Shows the content of that commit*
git reset filename	*Downstage file*
git checkout filename	*Fetch the file with the one from git repository*
Git rev-list --left-right --count master…refs/changes/36/etc..	*Shows the "Ahead","Behind" count, visible in gitk --all*
git add	*Add to staged area*
git commit	*Create a new commit*
git commit filename	*Adds the file in the staged area and then creates a new commit*
git checkout -b <branch_name>	*Creates an new branch and checkout it*
git branch -d <branch_name>	*Deletes branch*

git push origin --delete <deleted_branch_name>	*Push of the deletion of a branch*
git cherry-pick	*Just tries to apply the changes to the current files from another position*
git rebase	*Put the changes on top of the current commits*
gitdir=$(git rev-parse --git-dir); scp -p -P 29418 uidk5856@gerrit.cip01.conti.de:hooks/commit-msg {gitdir}/hooks/	*In case that the change ID is not generated*
git submodule update --recursive --init	*Updates all submodules recursive , initialize and clone if there are new submodules*
git diff --name-status <branch_name or commit id> <branch_name or commit_it>	*It will return only the files that are different between those two branches or commits*
git reset --hard HEAD~<number_of_commits>	*It will delete the last <number_of_commits> commits including the changes.*
git reset --soft HEAD~<number_of_commits>	*It will delete the last <number_of_commits> commits WITHOUT deleting the changes.*
git config -f .gitmodules submodule.<SubmoduleName>.branch <branch>	*Sets the branch of a child submodule.*

Config: ssh key generation, setting the user and general settings

ssh-keygen -t rsa -b 4096 -C "<your.email>@continental-corporation.com"	*Generating the ssh key*
whoami	*Returns the username*
git config --global user.name "<Firstname> <Lastname>"	*Set username*
git config --global user.email "<your.email>@continental-corporation.com"	*Set email*
git config --global core.autocrlf true	*Set auto-converting CRLF line endings in LF when you add a file to index, and vice-versa when it checks out code onto your system.*
git config --global core.longpaths true	*Use long paths*
git config --global fetch.prune true	*Any git fetch or git pull will automatically prune*
git config --global core.editor notepad	*Set core editor as notepad*
git config --global format.commitMessageColumns 7	*Set commit message columns to 7*
git config --global core.editor "'C:/Program Files (x86)/Notepad++/notepad++.exe' -multiInst -notabbar -nosession -noPlugin"	*Set notepad++ as core editor (if this doesn't work check notepad++ path)*
git config --global --list	*Shows the current global configuration list*
git submodule add -b <branch> --name <name> -- <repository> <path>	Add <repository> as a submodule of the project at a specific <path>, with a specific <name> and follows a specific <branch>.

Config all in one command

git config --global user.name "<Firstname> <Lastname>" && git config --global user.email "<your.email>@continental-corporation.com" && git config --global core.autocrlf true && git config --global core.longpaths true && git config --global fetch.prune true && git config --global format.commitMessageColumns 7 && git config --global core.editor "'C:\LegacyApp\Notepad++\notepad++.exe' -multiInst -notabbar -nosession -noPlugin"

Other helpful commands

Ls -l	*List all files with details*
./gka	*Opens gitk -- all &*
~/bin	*Is "/c/Users/uidk4083/bin" where "/c/Users/uidk4083" = Home = ~*
mkdir ~/bin	*Creates bin folder*
ls ~/bin	*Shows all files in the bin folder*
start .	*Opens Windows Explorer in the current path*
touch t.txt	*Creates an empty file*
rm filename	*Delete file*
cd path	*Change Directory to path*
rm -rf	*Remove recursive and force*

Shell shortcuts

CTRL+A	*Move the cursor at the beginning of the command*
CTRL+E	*Move the cursor at the end of the command*
CTRL+R	*Search in commands history (press CTRL+R for looping though commands)*
cd ~	*change directory to Home path*
./	*Current directory*
cd -	*Change directory to previous path*
CTRL+W	*Deletes the last word*

Aliases: gitk, gg, xt, Dat

Step 1: Download the archive: *bin.7z*

Step 2: Open a git bash and run: *cd - && start .*

Step 3: Extract the content of bin.7z in the location of the explorer window that opened in step 2.