USING GIT TO KEEP TRACK OF YOUR CODE DEVELOPMENT

https://faun.pub/learn-git-in-13-words-part-1-of-3-45e83db145fd
https://faun.pub/learn-git-in-13-words-part-2-of-3-ef5147034995

Video tutorial: https://youtu.be/RGOj5yH7evk

- ➔ Use command line to learn! (GUI exist, and if you know the CLI use, you'll know what to look for in GUI)
- ➔ NB: Differences between git and GitHub.org!

Essential concepts

#1. Repository

Short version: A root folder for your project.
Collection of your project files.

Create an empty folder, go into it and type
$ mkdir learn-git
$ cd learn-git
$ git init
- Initialized empty Git repository in ~/learn-git/.git/

With that, learn-git is now a git repository (or "repo"). You can tell because it has a subfolder called .git; that's where git keeps all the private stuff it needs to do what it does.

#2. Commit

Short version: A diary entry in your repository's history. at some point, you can decide to commit the files you've edited. This will count as a single step in your repository's history. Committing doesn't affect the files themselves; nothing in your repository will visibly change. Git doesn't automatically commit all the files you've changed.
The -m flag indicates that the string after it is the commit message.

```
$ git add README.md myfile.txt
$ git commit -m "Create a README.md , myfile.txt files"
```

Committing serves as the basis for everything else you can do with your repository.

#3. Log

Short version: The sequence of commits in your repository.The string of numbers and letters in the first line are the hash, which can be used in certain git commands to identify the commit. Below that are the author and date, followed by the message.

- Make your commit message meaningful, but concise.
- Don't commit too many files with too many changes at once. This goes hand in hand with the above — if you find that your message needs to be too long to describe what you've done, chances are you're committing too many things.
- The one acceptable exception is when you run init in a folder you've already been working in. In such a case, git will count all existing files (which can easily be hundreds) as uncommitted changes. Most of the time, you can just use add . and commit them all with a message such as "Initial commit".

```
[nperlin@gcat2:~/HPC-stack/hpc-stack]$ git log
commit 255ef2c94eed6f4c42a4db0fcabf2a7bc4a26dbd (HEAD -> feature/linux-centos-intel,
origin/feature/linux-centos-intel, noaaepic/feature/linux-centos-intel)
Author: Natalie Perlin <68030316+natalie-perlin@users.noreply.github.com>
Date:   Thu Feb 24 10:32:32 2022 -0500

    Update build_libtiff.sh

    corrected the address for libtiff download, removed double "lib" in the address
```

#4 Branch

Short version: Parallel universes.

Wouldn't it be great if you could split into three parallel universes where you try each solution, and then pick one of those as your new reality?
Issue the following commands in your git repository:

```
$ git branch solution-a
$ git branch solution-b
$ git branch solution-c
$ git checkout solution-a
```

So far, nothing has changed. Modify a few files, make one or two commits, check the log — everything works as usual. Now try this:

```
$ git checkout solution-b
$ git status
```

Your repository will now be in the state it was when you issued all those branch commands. If you pull up the log, there will be no sign of the commits you just made; indeed, if you check the files themselves, they will be back to their old contents. You can probably see where this is

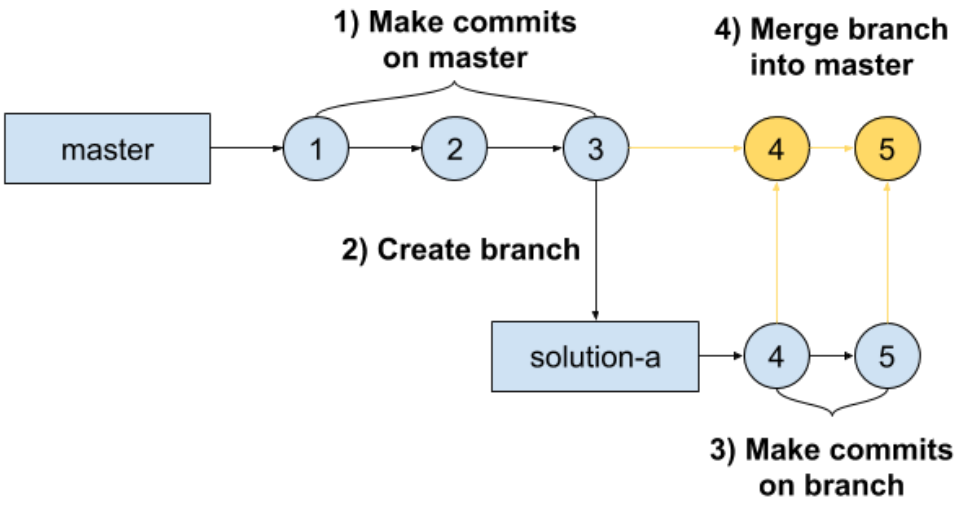
going, but just to drive it home, issue checkout solution-a once again, and take a look at your log: the commits are back, as are the files.

The branch command, unsurprisingly, creates branches in your repository; the checkout command allows you to switch to a different branch. When you make a commit, it's added to the history of the branch you're currently on, independently of all other branches. This way, you can make as many changes as you like, with no risk of breaking your existing code.
If you're not sure which universe you're in, just use branch without any parameters; this will give you a list of all branches, and mark the current one with an asterisk *:

```
$ git branch
  master
* solution-a
  solution-b
  solution-c
```

So, what do you do after you've decided on the ideal solution? Well, you could just keep working on that branch from then on, but this is very much not recommended. The principle of git is that the master branch should contain the definitive content of your repository, so you need to merge the commits from your chosen branch into that:

The merge command compares the histories of the current branch (here, master) and the target branch (here, solution-a), and copies all the commits from the target branch that were made after the latest commit of the current branch.



Note: For the time being, it's very important that you do not make any commits on the master branch while you're working on one of the alternate branches! If you do, you might not be able to merge them. We'll learn how to solve that later

Once you've merged your chosen solution, the alternate branches are no longer necessary. You can use the -d or -D flags to delete one or more branches:

```
$ git branch -d solution-a
Deleted branch solution-a (was 814621e).
$ git branch -D solution-b solution-c
Deleted branch solution-b (was 389ab8d).
Deleted branch solution-c (was 8f0374b).
```

The difference is that -d will only delete a branch if it has been merged into master, otherwise it errors out (in this case, you can use it with solution-a, which you've just merged). The -D flag deletes a branch no matter what, so be sure to use it carefully.

#5. Diff

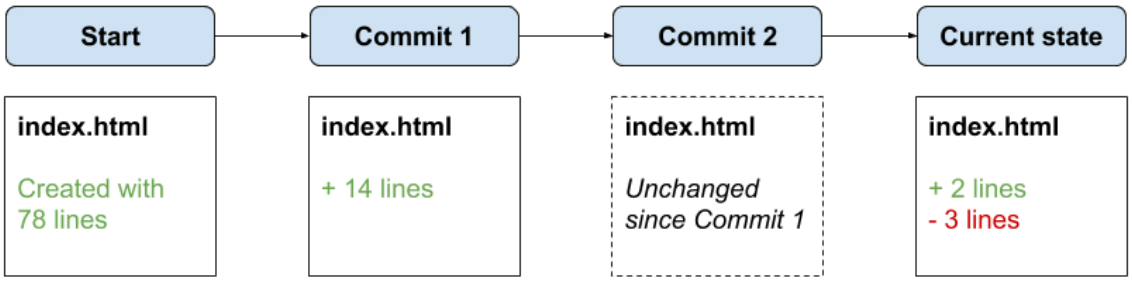
Short version: See which lines changed in which files.

So how exactly does git know what each file should contain when you change branches? Does it store actual copies of all files for each branch? That would quickly add up to a great deal of storage space. Fortunately, git's creator realized that as well, and used a much smarter solution. Make a simple change in your repository, such as adding a single line to a file, then issue the following command:

```
$ git diff
```

If you added a line to a file, you will see it highlighted in green (if your console supports it), and marked with a plus sign + before it. If you deleted a line, it will be in red with a minus sign -. If you changed a line, the old version will be marked as deleted, and the new version as added.

In grossly simplified terms, this is more or less what git uses to save space. When you make a commit, git creates a list of all the files in your repository, and uses the same algorithm to determine which ones have changed since the last commit. If a file's contents are the same, git marks it as unchanged; otherwise, git saves the differences — but only the differences. Meaning that, if you only changed two lines in a file that's thousands long, git will only save those two.



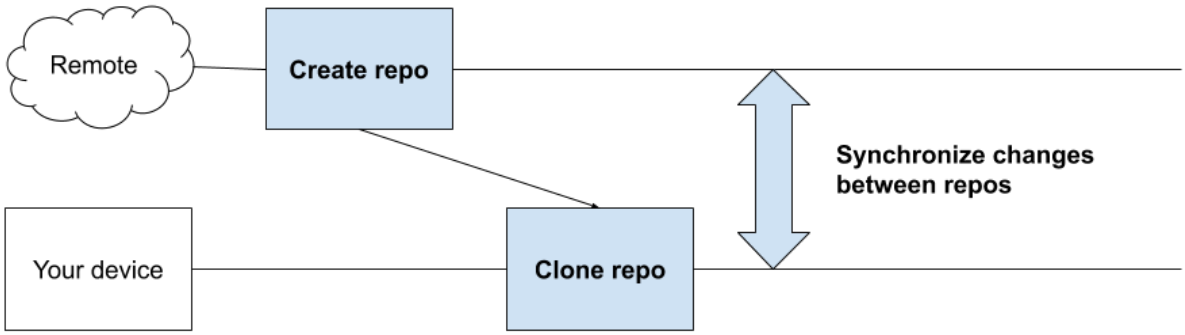
Using this method, git is able to perfectly recreate the state of any file at any commit, by starting from the beginning of your repository's history, and applying the changes of each commit in sequence. Obviously, storing the changes also adds some storage overhead, but it's negligible compared to creating a complete backup.

#6. Remote

Short version: A backup or master copy you can sync your repo with.
The most basic problem facing any team of developers is that they all need access to the same codebase.
Then, they each clone this repository onto their own computers:
`$ git clone https://github.com/our-team/our-project.git our-project`

The clone command is one of the few that you can (and should!) issue outside of a git repository. It's usually given two parameters: a URL to a repo, and a folder name. When issued, it does the following:

- Creates a new subfolder with the given name (here, our-project).
- Initializes the new folder as a git repo (just like if you went into it and issued init).
- Copies the contents of the repo at the given URL into the new folder, including git data such as logs and branches.
- Sets the repo at the URL as the remote origin of the newly created repo.



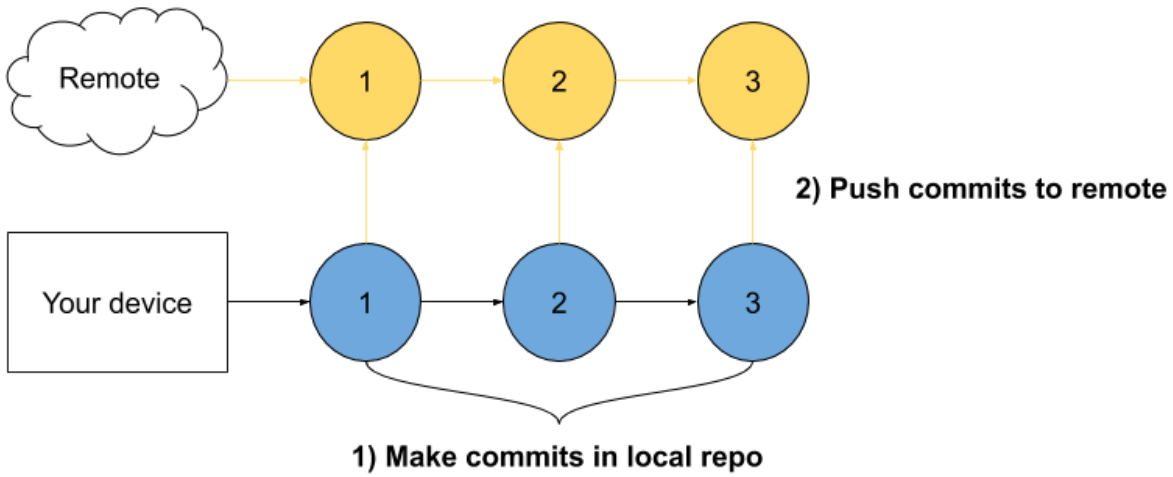
So, where exactly can you create this remote origin? If you've spent any time in the developer world, you've almost certainly heard of GitHub, GitLab, or BitBucket, to name a few. All of these services allow you to create repositories on their servers, from where you can clone them to any device you use.

#7. Push

Short version: Copy commits from your repo to the remote.

As mentioned above, your local repo will generally be considered a working copy of the remote origin. You work on the code, you create commits and branches, and at certain points (e.g. when you've finished working on a feature), you need to copy the current state of your repo to the remote. This is what git calls a push.

At its core, a push is similar to a merge. When you issue the command, git compares the history of your branch with the history of the remote branch, and copies all the new commits.



One question you might be asking at this point is, exactly which remote branch does git compare your local branch with?
Creating a branch in your local repo does not automatically create a matching branch in the remote origin. You have to explicitly tell git to do that when you make your push:
`$ git push -u origin solution-a`
Let's dissect this back to front:

- solution-a is the name of the branch we want to push (as used in the previous article)
- origin means you want to push to the remote origin. Theoretically, a git repo can have multiple remote repos attached to it, and you could push to any of them. However, this is extremely rare, so we'll just stick to origin throughout these notes.
- -u is a flag that tells git to create a new branch in the remote repo with the same name as the branch we're pushing (here, solution-a), and set it as the upstream of the local branch. You can think of the upstream as the branch equivalent of a remote; it's the branch on the remote that your local branch is meant to be synced with.

This only needs to be done when you push a branch for the first time. Once the remote branch is created, you can just use push with no parameters:

```
$ git push
```

When issued this way, git will assume you want to push the branch you're currently on to its upstream (so make sure you're always on the correct branch when you push!). If there is no upstream set, it will error out.
Note: You cannot perform a push if you have uncommitted changes in your repo. For the time being, you'll have to commit them before pushing. We will learn a more convenient solution later.

#8. Pull

Short version: Copy commits from the remote to your repo.
Let's say your teammate just pushed a new branch to the remote, and asked you to review their code. Since they've done the work on their own computer, you will not have it in your repository; but, as you may expect, you can use git to copy it from the remote origin.

First, you need to create a new branch on your own computer. Strictly speaking, it doesn't have to have the same name as the remote branch, but it makes good sense to have them match:
`$ git branch solution-a`
`$ git checkout solution-a`
Now, you can pull the new commits from the remote branch to your local one:
`$ git pull origin solution-a`

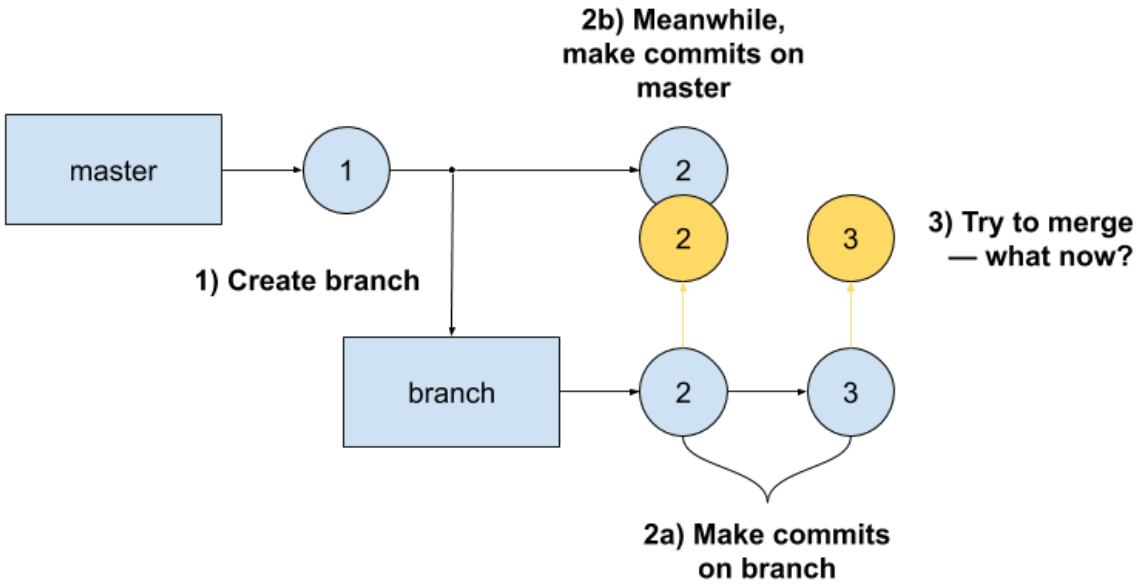
When you pull a branch, you have to specify the remote name (origin) and the branch name (here, solution-a) when issuing the command. If you push to a remote branch and set the upstream, you can afterward use pull with no parameters as well:

```
$ git pull
```

Note: Also just like with push, you cannot pull a branch if you have uncommitted changes.

#9. Conflict

Short version: Commits on different branches modified the same lines in the same file. Remember how, in part one, I said you shouldn't make any commits to the master branch before merging your alternate branch? Well, let's imagine you did.



Potential conflict. Both branches have commits that were made after the new branch was created.

In a situation like this, git does the best it can to figure things out on its own. As we already discussed, it keeps track of which lines of which file changed in each commit. If it can confidently determine that the commits on the two branches affect either different files, or unrelated lines in the same file, it will just go ahead and apply the merged commits on top of the ones on the master branch.

If, however, it appears that two commits affect the same lines in the same file, git will halt the merging process, and signal a conflict:
Automatic merge failed; fix conflicts and then commit the result.

When this happens, you can use the status command to see which files are involved in the conflict. The git status command displays the state of the working directory and the staging area. It lets you see which changes have been staged, which haven't, and which files aren't being tracked by Git. Status output does not show you any information regarding the committed project history.

```
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)
Unmerged paths:
  (use "git add <file>..." to mark resolution)
both modified:   index.css
```

If you open the files in question (in this example, index.css), you will find that git has inserted both versions of the affected lines, and marked each with the branch it's coming from:

```
<<<<<<< HEAD
body { background-color: red; }
=======
body { background-color: blue; }
>>>>>>> solution-a
```

This may look somewhat confusing, so let's take it line by line and decipher what each one means:

- <<<<<<< HEAD — This line indicates that the lines below are the changes made on the branch that you're merging into (in git terminology, HEAD refers to the branch you're currently on).
- Afterward come the actual lines that had been edited. In this case, a commit was made on the master branch to set the background color to red.
- ======= — This acts as a separator between the two blocks of lines.
- Below the separator are the lines that had been edited on the branch you're merging from. In this case, a commit was made on the solution-a branch that set the background color to blue.
- >>>>>>> solution-a — Finally, the last line indicates the name of the branch you're merging.
-

Now that we understand what's happening, here's what you need to do:
Go through the affected files, resolve each conflict, and remove the conflict marker lines (i.e. the ones beginning with <<<<<<<, >>>>>>>, and =======).
Once all conflicts are resolved, stage and commit the affected files to complete the merge process:

```
$ git add index.css
$ git commit
```

If you’re not sure how to resolve the conflict, you can abort the merge at any point:
`$ git merge --abort`

This will safely revert all files to the state they were in before you started the merge.
When resolving conflicts, you will usually keep one of the changes and discard the other. You should know, though, that this is by no means mandatory. You can edit the file however you like, keep both changes, or neither, or some amalgam of the two. As long as you remove the conflict markers, git will assume you know what you’re doing.
Conflict resolution is one of those situations where a graphical git client can work wonders. They can highlight the clashing changes, let you compare them side by side, and pick one or the other version with a single click. Most developers prefer to use them, and in practice I encourage you to do the same. Still, it never hurts to know what’s actually happening.