

Module Introduction

Get started using Git for your project

Objective

Get started with Git for version control of your projects

Background/Recap

Ever been in a situation where

- You badly needed your old code but it's lost beyond the reach of `Ctrl-Z`
- You take backups of your work in progress into different folders like **project-x-01082020**, **project-x-02082020** etc.
- You need to integrate changes made to a project independently by your team members without losing any code
- You need to keep track of metadata on a project worked on by multiple developers like - what/who/when/why for any changes made

Version control systems help in these situations. They help us keep track of changes to our files by taking snapshots of them. We can look at earlier snapshots and also restore our code to an earlier snapshot. This lets us work on our projects without worrying about introducing errors to partly/completely working versions and not being able to go back. Git is one of them, probably the most popular one.

Git projects are called Repositories and contain Git related files apart from the project files we add. These git files contain info about all the changes made - who made the changes, what changes were made, when was it made etc. Similar to having Google Drive for file sharing, we have version control providers like GitLab, GitHub etc. to facilitate the use of Git repositories.

Primary goals

1. Understand how Git works
2. Learn the basic Git commands

Setup - Crio users

- These are the links to your GitLab repository. We'll use these in the next milestone

```
git@gitlab.crio.do:COHORT_ME_GIT_BASICS_ENROLL_1596802014715/somesh-jpn-ME_GIT_BASICS.git
```

```
https://gitlab.crio.do/COHORT_ME_GIT_BASICS_ENROLL_1596802014715/somesh-jpn-ME_GIT_BASICS.git
```

- Goto the above **https** link to view your Git repository in Gitlab. You'll need to sign in with Google using the account registered with Crio.

Setup - Gitpod

- These are the links to your GitLab repository. We'll use these in the next milestone

```
git@gitlab.crio.do:COHORT_ME_GIT_BASICS_ENROLL_1596802014715/somesh-jpn-ME_GIT_BASICS.git
```

```
https://gitlab.crio.do/COHORT_ME_GIT_BASICS_ENROLL_1596802014715/somesh-jpn-ME_GIT_BASICS.git
```

- Goto the Gitpod workspace and open the terminal from within
- To download/upload contents from/to the above Gitlab repositories you'll have to prove you're having access to it. Execute the below command to generate an SSH key. **SSH** protocol allows you to remotely access machines using the SSH key.

```
ssh-keygen -t ed25519 -C "crio-git-byte"
```

- You'll be asked to enter the file to save the generated key and passphrase to secure it. Hit enter for both of these, you don't have to type anything.

```
gitpod /workspace/me_git_basics $ ssh-keygen -t ed25519 -C "crio-git-byte"
Generating public/private ed25519 key pair.
Enter file in which to save the key (/home/gitpod/.ssh/id_ed25519):
Created directory '/home/gitpod/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/gitpod/.ssh/id_ed25519
Your public key has been saved in /home/gitpod/.ssh/id_ed25519.pub
The key fingerprint is:
SHA256:w00zRPXAQfnvTw56SYmFfD3FRKQun/K5docfYRXwihk crio-git-byte
The key's randomart image is:
+--[ED25519 256]--+
|      .+=0 ..*+|
|      . 0 00 ..+|
|      o o  E...00|
|      +   B.+..0|
|      S  o.*.+..|
|      o  .0=..  |
|      .0++..  |
|      +*==+   |
|      .0+===  |
+-----[SHA256]-----+
gitpod /workspace/me_git_basics $
```

- Did you notice you were given two links to your Gitlab repository earlier? The one which starts with `git@gitlab.com` is to be used to authenticate using SSH. The `https` link authenticates using your `gitlab.crio.do` username and password
- Execute `cat ~/.ssh/id_ed25519.pub` and **copy** the public key printed. You'll add this to <https://gitlab.crio.do/> in the next step

```
gitpod /workspace/me_git_basics $ cat ~/.ssh/id_ed25519.pub
ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAINTtXwpi/NrK9aC+rZ0acRwAksX4cYdq925UDNdxCqYa crio-git-byte
gitpod /workspace/me_git_basics $
```

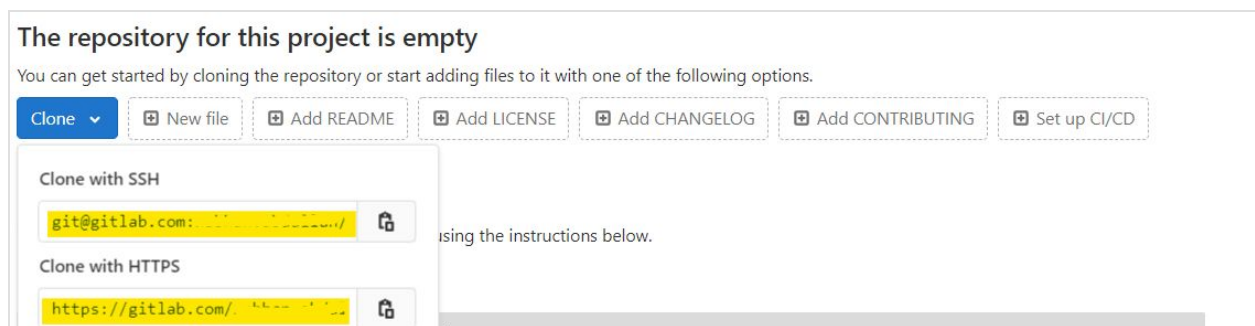
- Goto <https://gitlab.crio.do/> and copy-paste this key by following the instructions in the below video. (Note: You have to visit <https://gitlab.crio.do/> and not gitlab.com) <https://youtu.be/mNtQ55quG9M>
- Execute the below commands to setup the directory structure

```
mkdir -p ~/workspace/bytes
```

```
cd ~/workspace/bytes
```

Setup - Others

- An empty Git repository is required, follow the steps [here](#) to create a blank project on [GitLab](#). (**You won't have permission to create one on [gitlab.crio.do](#)**) Nb: Don't select the **Initialize repository with a README** option
- Download and install Git client from [here](#)
- Setup your Git client as explained in [this](#) article
- Create & add an SSH key to GitLab
 - Create SSH Key - [Ubuntu Windows](#)
 - [Add SSH key to Gitlab](#)
- Get the GitLab repository links which we'll use in the next milestone



- Execute the below commands to setup the directory structure for uniformity

```
mkdir -p ~/workspace/bytes
```

```
cd ~/workspace/bytes
```

Getting a copy of the code to your system

All Git related commands start with the `git` keyword. One of the frequent commands you'll use will be `git status` which gets you the current *status* of the Git repository. Try executing it.

Git would've told you something like this

```
fatal: Not a git repository (or any of the parent directories): .git
```

Not the best of welcome message you might've anticipated from Git, still it's something for us to get started with.

The output from a Git command, if successful, tells us that it succeeded with some additional information, else it prints out details which can tell us why exactly the command failed. These details are very useful to address the error and run the command again.

The above message means that you are not in a directory which is part of a git repository.

You'd have your Git repository links from the setup task. Go to the **HTTPS** link (starting with `https://gitlab`) of the repository in your browser. GitLab would tell you it's empty. Let's download the "empty" repository to `~/workspace/bytes/` directory in our local system. Use the below command with the **SSH** link (starting with `git@gitlab`) to your repo to do this.

```
mkdir -p ~/workspace/bytes
```

```
cd ~/workspace/bytes
```

```
# Ensure you clone using the ssh link and not the https link
```

```
git clone <add-ssh-link-here>
```

You'd be able to see a new folder. Does it's name have any correlation to the repository name? Is the new folder empty as GitLab told us or was it all a lie? (Hint: Hidden files)

Now that we have the same Git repository on our local system as on GitLab, the former will be referred to as the `local` repo and the GitLab one, `remote` repo.

Remotes in Git are usually places like GitLab & GitHub where we can share our code with others. Once you `cd` to the local repo folder, execute `git remote -v` to see something like this

```
$ cd ~/workspace/bytes/somesh-jpn-ME GIT BASICS
```

```
$ git remote -v
```

```
origin  
git@gitlab.crio.do:COHORT_ME_GIT_BASICS_ENROLL_1596802014715/somesh-jpn-ME_GIT_BASICS.git (fetch)
```

```
origin  
git@gitlab.crio.do:COHORT_ME_GIT_BASICS_ENROLL_1596802014715/somesh-jpn-ME_GIT_BASICS.git (push)
```

This just means that whenever Git says **origin**, it's referring to the repository denoted by

```
git@gitlab.crio.do:COHORT_ME_GIT_BASICS_ENROLL_1596802014715/somesh-jpn-ME_GIT_BASICS.git
```

Try `git status` again from inside the repo folder. Assuming you weren't eager enough to have made any changes to the files or folders after cloning the repo yet, you'll get this output

```
On branch master
```

```
Initial commit
```

```
nothing to commit (create/copy files and use "git add" to track)
```

- Line 1: Mentions that we're currently in a branch named **master**. What is a Git branch? Why do we need them?
- Line 2: Mentions that we are yet to make our initial commit
- Line 3: Mentions that you haven't made any changes that need to be committed

Wait, wait! What does a **commit** mean?

We'll see in the next milestone :)

For now, you have cloned a remote GitHub repository and you can access it on your machine just like a folder. You might also observe that there is a `.git` folder inside the repo. See if you can find out what this contains.

References

1. [Git Cheatsheet](#)
2. [A Short History of Git](#)
3. [Branches in Git](#)
4. [What is the git folder?](#)

Curious Cats

- When would you need multiple remotes?
- Is there any difference between using the `https` link and the `ssh` link to clone a repository?
- What if you need to use a different name for the download folder rather than the name of the repository with `git clone`?
- Where does `git remote` command get the mapping of remote name & the corresponding links?

Saving changes locally

Let's get started with adding new files. I'll create a file, **first.txt** and add some text to it. Feel free to create your own file(s). A file goes through different states in a Git repo - any changes we make are said to be in the *working directory*. Ok, try executing `git status` again - No, you won't get the earlier message or rather you shouldn't - Things don't happen the same way twice in life :)

```
On branch master
```

```
Initial commit
```

```
Untracked files:
```

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

```
first.txt
```

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


We have some familiar lines in there and some newcomers as well!

It's trying to tell us that **first.txt** is an *Untracked* file. What does it mean for a file to be tracked in Git?

Line 4 is Git being a good Samaritan. It's kind enough to tell us that the `git add` command can be used to add files to the next commit. A *commit* is like a checkpoint in Git where we save the current state of our files.

Use the `git add` command to mark your files for the next commit. This pushes the changes from the *working directory* to the *staging area*. Use `man git add` to see the manual page. How does `git status` look now?

We've added new files, do they reflect on GitLab? Login and check.

We're one step away from making our very first commit. We do that using `git commit`. The thing is, commits are like checkpoints for us to navigate back and forth (if required). It's better to add a message to our commit. Don't worry, our good friend Git will remind you (or should I say, force) to add a message to your commit.

If you have a default text editor configured, it will open up, enter some info about what your commit has/is and close it. In case you were wondering, I added "Add first.txt file".

```
crio-user@crio-demo:$ git commit
[master 201c556] Add first.txt file
1 file changed, 3 insertions(+)
create mode 100644 first.txt

##### NOTE FROM CRIO #####

Submit your code for assessment using the instructions in the last milestone task.
crio-user@crio-demo:$
```

Now that we've committed our changes, it should reflect on GitLab, right? Go and check.

We can use `git log` to view all our commits. Each commit can be uniquely recognized by an ID. In this case, the commit ID starts with 869634. How do you think the ID is calculated, is it just a random number?

```
commit 869634735e48ad5b08c48239a9be84287bf15216 (HEAD -> master, origin/master)
Author: CrioUser <crio-user@criodo.com>
Date: Wed Aug 5 01:47:43 2020 +0530

    Add first.txt file
```

Make changes to one of the files you just committed. Follow the same procedure to create a second commit with changes. See the output of `git status` after each command. Does it differ from what you saw earlier?

Let the world know!

We committed our ground-breaking updates, but they still haven't shown up in the GitLab UI. Let's see how to do this.

Pushing is the art of sending updates from your local repo to the remote repo, which in our case is the GitLab repo we initially cloned from. One or more commits form a push. Use `git push -u origin master` for that. We specify the remote name & its branch we're pushing our code to.

```
crio-user@crio-demo:~$ git push -u origin master
Counting objects: 3, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 294 bytes | 26.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To gitlab.crio.do:COHORT_ME [redacted].git
 91a3b87..201c556 master -> master
Branch 'master' set up to track remote branch 'master' from 'origin'.

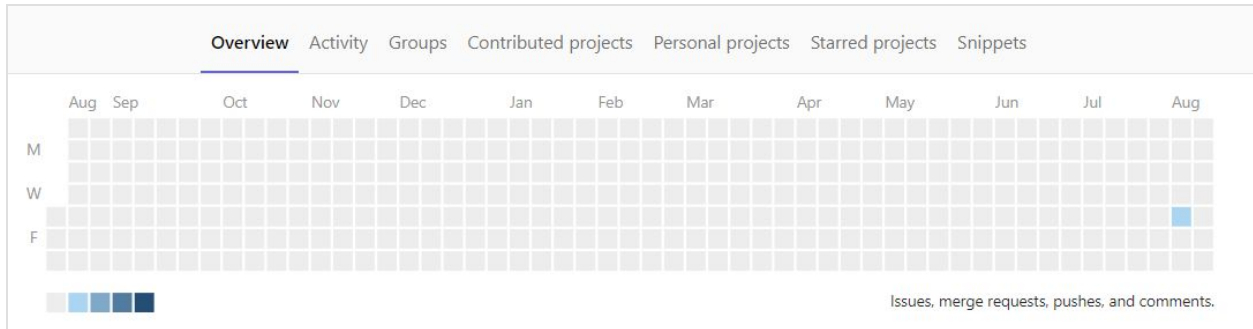
##### NOTE FROM CRIO #####

Great Work! Your assessment report will be available shortly in the Assessment Tab and through email
crio-user@crio-demo:~$
```

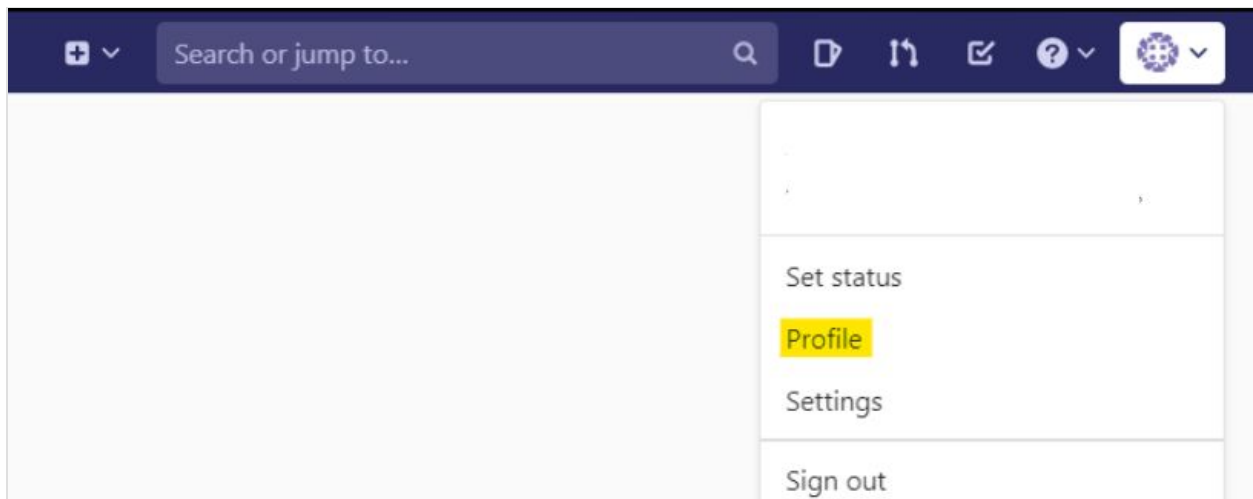
Finally! You'll be able to see the changes we've made locally in the GitLab repo. Get familiar with the details shown there, like

1. When was a file last updated in the remote repo?
2. Who made the commits and when?
3. See changes to files in each commit

Congratulations! You've successfully pushed your changes to Gitlab and have made your mark on the Contribution graph.



Go to your Gitlab profile to see it



The contribution graph is a quick indicator to recruiters that you've been working on projects & following industry best practices like using version control (Git) for it.

References

1. [Git: Add All Files to a Repo](#)
2. [Commit message style guide](#)
3. [The Git Commit Hash](#)

Curious Cats

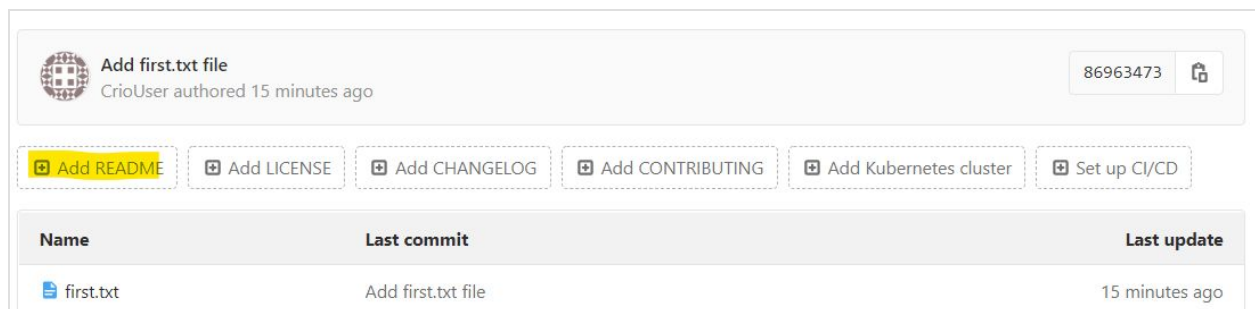
- How would you mark all changes for committing using `git add` instead of listing out each of the file names?
- How can we move changes to a modified file from the staging area back to the working directory? What if we need to get the file back to the previous committed state?
- What if we don't want the text editor to show up its glorious face when `git commit` is run? Is there a workaround to provide the commit message with the command itself?
- Now that we have only a handful of files, it's easy to remember what changes were newly made. How'd you check the changes made to the local repo since the previous commit?
- Rename a file and make a commit. Now, make some edits to the file and commit again. `git log --follow <filename>` lists all commits that changed this file. You'll be able to find commits related to the original file name are also listed out. How does Git detect renames?

Getting updates from the remote

Most of the time, you'll be working on projects as part of a team. Each of you will be pushing new updates to the remote repository. How can you get the latest version of the code, with changes made by others, from the remote repo to your local repo?

Let's experience one such scenario. Readme files are usually helpful to provide instructions to users about the repository - what it is, how to set up the project, etc. Let's create one for our git repo!

Use the GitLab UI to create a new Readme file.



Add the below text to the Readme file

```
Learning Git Basics [Added via GitLab]
```

You'll be able to see the file added in GitLab. Did a new commit get auto-created?

Now we need to get these changes reflected in our local repo as well. How do we do that?

Let's take a step back and think about how we did the opposite i.e., get changes from local repo to the remote. That was `git push`, right? So, this should be ___?

Once you've pulled the changes from the remote repo to your local repo, verify the changes by checking the Readme file. Can you see the new commits (Hint: Use `git log`)?

Can you quickly answer the first question below and comeback :)

You first! No, you first!

I have a couple of tasks for you

1. Create/edit another file in the local repo and commit changes
2. GitLab lets you edit files from the UI itself. Add "Learnt about git clone/status/log/add/commit commands" to the README file via GitLab and save the changes.

What happens if you try to push your local changes now?

Hmm, Git doesn't seem to be happy with that :(

```
To git@gitlab.com:nabhan.abdullah/git-basics-byte.git
! [rejected]        master -> master (fetch first)

error: failed to push some refs to
'git@gitlab.crio.do:COHORT ME GIT BASICS ENROLL 1596802014715/somesh-jpn-
ME GIT BASICS.git'

hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository
pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for
details.
```

Why could that be?

Git has presented us with intel on why the error occurred as always. It's trying to tell us something about the updates being rejected due to the remote having work that we don't have locally. Ah, there you go! We had made some changes to the README earlier via GitLab UI, right?

We'll need to use the `git pull` command first to integrate the remote changes locally & then push. Do both of these.

But, why did we have to do a `git pull` followed by a `git push`? Why wouldn't it work the other way around? Think about a scenario where multiple team members could be changing the same file as well. We'll get to this in the next milestone.

References

1. [Git Pull Explained](#)
2. ["Remote contains work that you don't have locally" error explained](#)
3. [Other ways to resolve the "Remote contains work that you don't have locally" error](#)

Curious Cats

- Both `git clone ...` & `git pull ...` were used to download files from remote to local, how are they different?
- There's another command, `git fetch ...`. How's it different from a `git pull ...`?

Time to make a choice!

Ready for more tasks that are common during a software cycle?

1. Update the Readme title to **Learning lots of Git Basics [Added via GitLab]** via GitLab UI
2. Update the Readme title to **Learning more Git Basics [Added via local repo]** from the local repo and commit the changes

Push the changes. You know what happens next, right? Yep, Git will ask you to pull first. Is the situation under control once you Pull?

You'd be shown a message like this (if you are on Crio workspace, you'll be asked to enter **y** or **n** based on if you know how to resolve a merge conflict. Enter **y**, we'll go through how to resolve the merge-conflict below. Entering **n** will remove any changes caused by the pull command. Do a pull again if you accidentally hit **n**)

```
Auto-merging README.md
```

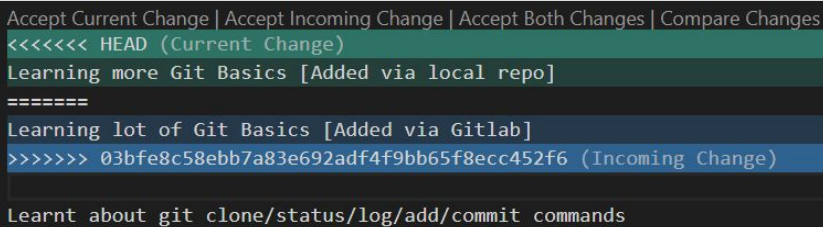
```
CONFLICT (content): Merge conflict in README.md
```

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

It's hinting towards some conflict in the README.md file. But, why didn't this happen the last time we did the same process?

On pulling, Git checks the changes coming from the remote, sees that the same file has changed locally as well & automatically tries to include both sets of changes. In this case, we've made changes to the same line of the README file both locally & remotely. Git wouldn't know which one to prioritise and needs our help to resolve this `merge conflict`.

Open the README file in your editor



```
Accept Current Change | Accept Incoming Change | Accept Both Changes | Compare Changes
<<<<<< HEAD (Current Change)
Learning more Git Basics [Added via local repo]
=====
Learning lot of Git Basics [Added via Gitlab]
>>>>>> 03bfe8c58ebb7a83e692adf4f9bb65f8ecc452f6 (Incoming Change)
Learnt about git clone/status/log/add/commit commands
```

Current Change is what you had locally & **Incoming Change** is that from the remote.

Make up your mind and choose one. You can also choose both sets of changes and edit them further if that is required. After you've made the changes according to your choice, do a `git add README.md` and `git commit` to complete the merge.

References

1. [About Merge Conflict](#)

Note

If you are using Gtipod, it will be a good idea to remove the SSH key you added to <https://gitlab.crio.do/> after you complete the Byte tasks for security reasons.

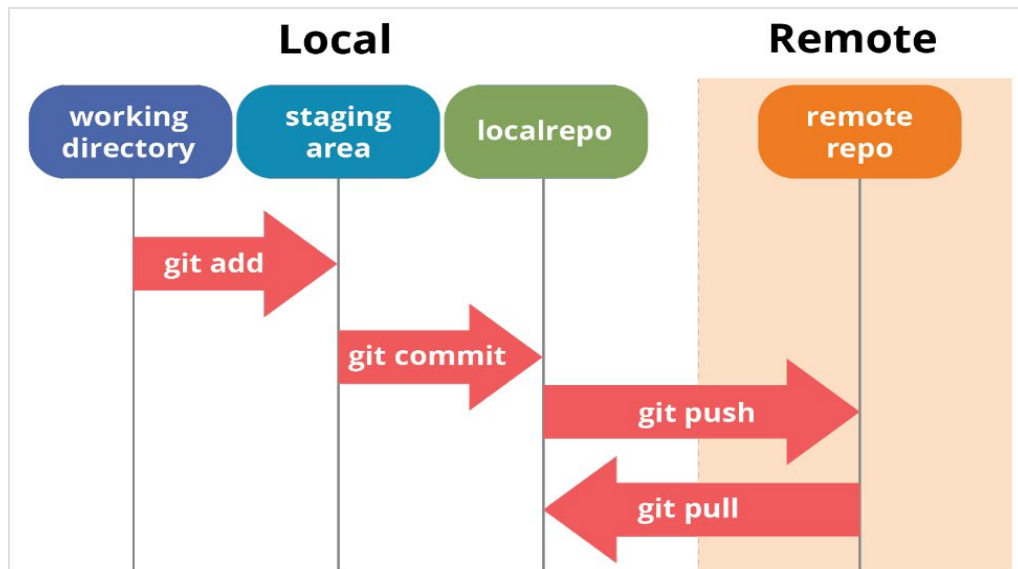
Curious Cats

- What kinds of merges can Git do automatically?

Summary

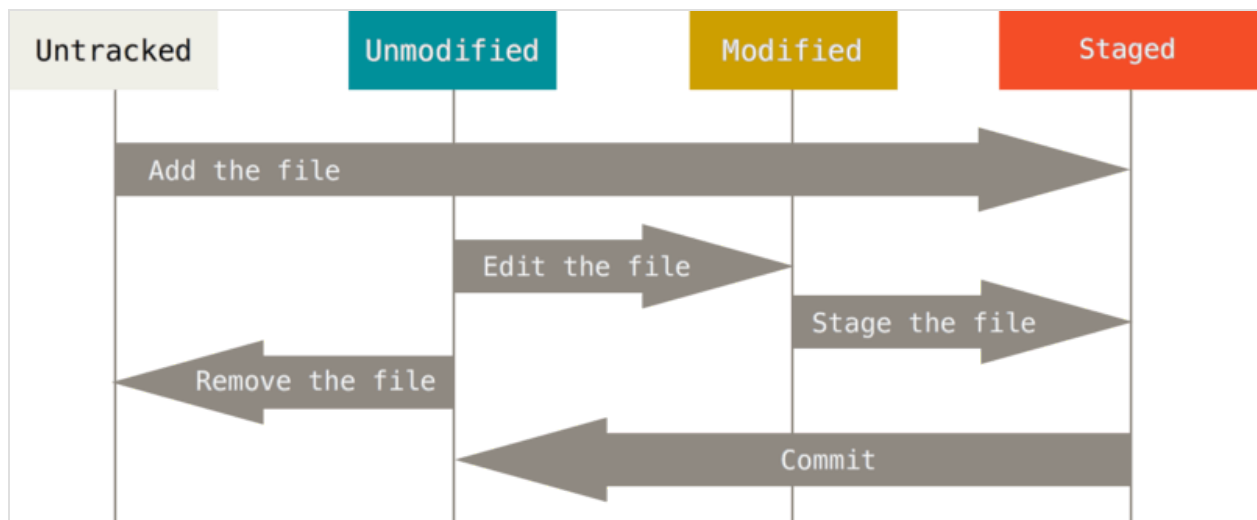
Basic workflow in Git

1. You modify files in your working directory
2. You stage the files, adding snapshots of them to your staging area (`git add file`)
3. You do a commit, which takes the files as they are in the staging area and stores that snapshot permanently to your local Git repository (`git commit file`)
4. You push the changes from the local repository to the remote repository (`git push`)
5. You pull the changes from the remote repository to the local repository (`git pull`)



[Source](#)

- State of the file according to the actions



[Source](#)

- Find
 - Quiz answers [here](#)
 - Pointers to the *Curious Cats* questions [here](#)

Further reading

Git offers several more features which help in maintaining code and collaborating with team members during software development. For example - config, branch, stash, squash etc. You can explore these as needed.

- [What is Version Control?](#)
- [Oh Shit, Git!?!](#)
- [The Advanced Git Guide](#)

Excited? Dig deeper to Git internals for answering the below questions

- [What actually happens within Git when we do a commit?](#)
- [Each commit gets a unique ID, how is it created?](#)

Newfound Superpowers

- Ability to use Git for your projects and never lose older versions.

Now you can

- Download code from hosting services like GitLab/GitHub & start working on them locally
- Make changes locally and push them back to the GitLab/GitHub
- Use Git to create checkpoints (sets of changes tracked together) in your project
- Share code and resolve merge conflicts while working in a team