

The background features abstract, overlapping green geometric shapes, primarily triangles and polygons, in various shades of green, creating a modern and dynamic visual effect.

GitLab

Basic Fundamentals

Outlines

- ▶ What is GIT.
- ▶ Why is everyone using GIT.
- ▶ GIT Concept.
- ▶ GIT Commands.
- ▶ Brief Details of GIT and it's Commands.
- ▶ Cheat Sheet for GIT.
- ▶ *Appendix to study workflow in GITLab

What is GIT

- ▶ Git is a fast, scalable, distributed revision control system with an unusually rich command set that provides both high-level operations and full access to internals.
- ▶ Git is a version-control system for tracking changes in computer files and coordinating work on those files among multiple people. It is primarily used for source-code management in software development.

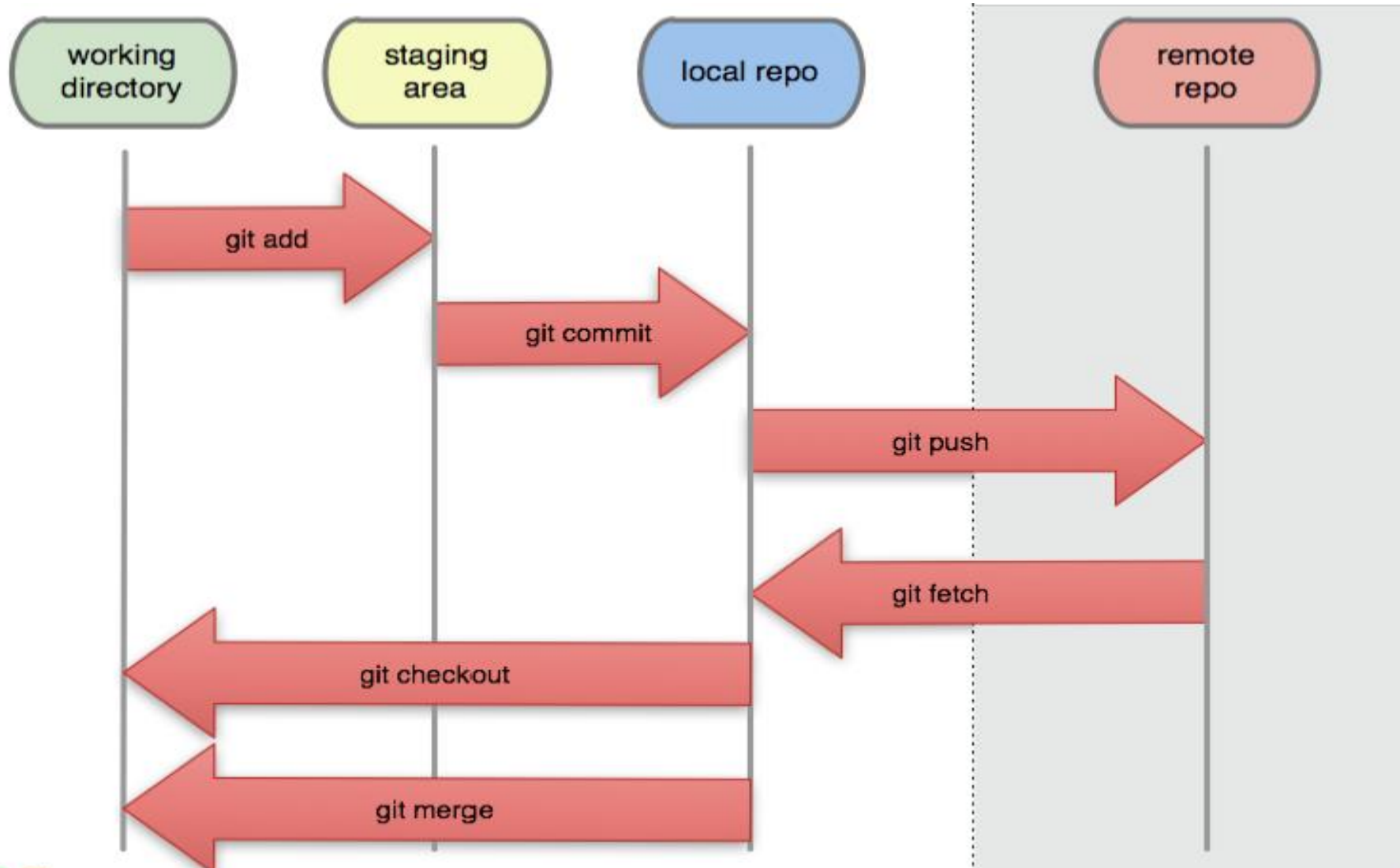
Why everyone is using GIT

- ▶ **Branching:** Branching means you diverge from the main line of development and continue to do work without messing with that main line.
- ▶ **Local:** A **local branch** is a branch that only you (the local user) can see. It exists only on your local machine.
- ▶ **Fast:** It is fast to manage the code in local machine and push, merge in master file when needed.
- ▶ **Distributed:** Several branches can work on single master repository.
- ▶ **Small:** Very small software to carry out tracking of our project.
- ▶ **Staging Area:** Staging is a step before the commit process in git. That is, a commit in git is performed in two steps: staging and actual commit. As long as a change set is in the staging area, git allows you to edit it as you like (replace staged files with other versions of staged files, remove changes from staging, etc.).
- ▶ **Workflows**
- ▶ **Popularity-Github, Bigbucket**

GIT Concept



GIT Concept(Continue..)



Common GIT Commands

<code>add</code>	Add file contents to the index	<code>merge</code>	Join two or more development histories together
<code>bisect</code>	Find by binary search the change that introduced a bug	<code>mv</code>	Move or rename a file, a directory, or a symlink
<code>branch</code>	List, create, or delete branches	<code>pull</code>	Fetch from and integrate with another repository or a local branch
<code>checkout</code>	Checkout a branch or paths to the working tree	<code>push</code>	Update remote refs along with associated objects
<code>clone</code>	Clone a repository into a new directory	<code>rebase</code>	Forward-port local commits to the updated upstream head
<code>commit</code>	Record changes to the repository	<code>reset</code>	Reset current HEAD to the specified state
<code>diff</code>	Show changes between commits, commit and working tree, etc	<code>rm</code>	Remove files from the working tree and from the index
<code>fetch</code>	Download objects and refs from another repository	<code>show</code>	Show various types of objects
<code>grep</code>	Print lines matching a pattern	<code>status</code>	Show the working tree status
<code>init</code>	Create an empty Git repository or reinitialize an existing one	<code>tag</code>	Create, list, delete or verify a tag object signed with GPG
<code>log</code>	Show commit logs		

Common GIT Command We will Cover

<code>add</code>	Add file contents to the index
<code>branch</code>	List, create, or delete branches
<code>checkout</code>	Checkout a branch or paths to the working tree
<code>clone</code>	Clone a repository into a new directory
<code>commit</code>	Record changes to the repository
<code>fetch</code>	Download objects and refs from another repository
<code>push</code>	Update remote refs along with associated objects
<code>status</code>	Show the working tree status

Brief Details of GIT and It's Commands.

Configuring GIT for first time

Now that you have Git installed and to prevent warnings, you should configure it with your information.

- ▶ -> `git config --global user.name "Your Name"`
- ▶ -> `git config --global user.email youremail@domain.com`

Now we can see the config file by

- ▶ -> `ls -la`

also by,

- ▶ -> `git config --list`
- ▶ or -> `cat .gitconfig`

~/gitconfig contents

```
[user]
  name = Your Name
  email = youremail@domain.com
```

Configuring GIT for first time(Continue..)

Now we have configured git for first time use.

For test editor we use Vim editor.

- ▶ -> `git config --global core.editor "vim"`

For color of output by git we use:

- ▶ -> `git config -global color.ui true`

And all the option can be seen by:

- ▶ -> `cat .gitconfig`

For installing auto-completion (Window already provides) for Linux or Mac we use:

- ▶ -> `curl -OL https://github.com/git/git/raw/ master/contrib/completion/git-completion.bash`

Configuring GIT for first time(Continue..)

And move the file to .git-completion.bash file

- ▶ -> mv ~/git-completion.bash ~/.git-completion.bash

press

- ▶ -> ls -la

To see the list of all files.

To activate autocompletion we will use vim text editor to edit file in .bashrc

- ▶ vi .bashrc then write
- ▶ if [-f ~/.git-completion.bash]; then
- ▶ source ~/.git-completion.bash
- ▶ fi

Press esc and shift+:wq to exit and save.

Work with GIT

We can type:

- ▶ -> git help
 ,to get all the GIT Help. Also if we press
- ▶ -> git help <command> It will open manual for the particular command
- ▶ -> like git help fetch

Press f- forward b-backward or space to walk through the page. Q to get out
or we can use

- ▶ -> man git-fetch or any ther command it will open exact same manual.

Initializing and committing in GIT Repository

Now we need to make a directory and initialize as git repository

- ▶ -> `mkdir first_git_project`

Now we have to inform git that the repository is git repository for that,

- ▶ -> `git init`

Create some sample file like

- ▶ -> `echo "The first project" > some.txt`

Now we want to put the files in staging state and after that commit.

- ▶ -> `git add .`

This will add all changes to this entire project.

And commit the changes

- ▶ -> `git commit -m "Initial commit"`

GIT Log

To see the log of all the commit

- ▶ -> git log

For limit the number of commit we type:

- ▶ -> git log -n 1

Or,

- ▶ -> git log --since=2012-06-12

will show all logs after date.

- ▶ -> git log --until=2018-12-06

Or if we want to see the commit made by user we type the command,

- ▶ -> git log -author="Name"

We can also see the changes based on commit like File with Init, java, bug fixes etc

- ▶ -> git log --grep="Init"

Init is for all the changes similarly other like fix.

Git Status

For observing the difference of all the working directory-> staging index -> repository, we type

- ▶ -> git status

Currently we are in master, that we have created so it will give working directory clean.

Let, we add another file, let suppose second.txt and third.txt

And let add to the directory

- ▶ -> echo "The second file" > second.txt

- ▶ -> echo "The third file" > third.txt

And type:

- ▶ -> git status

It will show the untracked both files.

Git Staging and Diff command

Now we will add the file to staging area

- ▶ -> `git add second.txt` or (`git add .` will add all the file in one go)

again `git status` will show all the tracked and untracked file, I.e `third.txt`

- ▶ -> `git commit -m "Second file to project"`

- ▶ -> `git log`

will tell all the commits

For getting what changes have took place in files we use in working directory,

- ▶ -> `git diff`

- ▶ -> `git diff some.txt`

will give the changes in single file.

For viewing the difference of staging index we type:

- ▶ -> `git diff --staged`

Remove and Renaming File

To remove file we use

- ▶ -> `git rm file.txt`

It will delete all the file mentioned from git directory.

For renaming we use:

- ▶ -> `git mv firstfile.txt first_file.txt`

and the file is renamed and we can check the status.

To move the file in directory:

- ▶ -> `mkdir first_folder`

- ▶ -> `git mv first.txt first_folder/first.txt`

For taking the difference in different color side by side we use:

- ▶ -> `git diff --color-words file.html`

GIT undo

Now if we want to change without going to staging state we can do:

- ▶ -> `git commit --am "Commit all"`

--am means adding and committing simultaneously.

For undo in present working directory use --

- ▶ -> `git checkout -- index.html`

In your case there could be any file name or folder name.

Now if we modified the file in staging stage then,

We can undo it using:

- ▶ -> `git reset HEAD resource.html`

or any file name, followed by checkout as we also need to do undo in working directory.

Undo Commit Changes

Now how to undo the particular file from our commit changes.

- ▶ -> `git checkout 'Hexdecimal no.' resource.html`

Hex can be obtain in git log files. Let suppose

- ▶ -> `git checkout 2907d12603 resource.html`

The first 10 digit is unique.

If we want just revert what we have done in last commit we can revert it in git:

- ▶ -> `git revert 2b6dnkd8sbsjw`

With the given hex code everything being tracked in file will be reverted from given hex commit.

After hitting enter it will open vim editor and give commit name, save and close.

Working of .gitignore file

It is used to ignore the files what we do not want to track in our working directory.

We use `!`, `#` or other expression to do that. Let's suppose we want to ignore all the text file's with extension, `*.txt`

- ▶ `*.txt`

If we write it in `.gitignore` file then all `txt` file will get ignored in git, so press

- ▶ `-> vi .gitignore`

And write the `*.txt`, `*.io` etc to ignore their tracking.

For writing comments we have `#` symbol before any comment.

Branching: Creation

- ▶ -> git branch

This command will show all the list of branch that we have in local machine.

*mark will show the current branch we presently are in.

We can track the git head pointer for branch and master

For oneline commit statement we write,

- ▶ -> git log --oneline

For creating new branch type:

- ▶ -> git branch branch_name

This command should be passed one time for branch creation.

then check branch using

- ▶ -> git branch

Branching: Switching

In this case HEAD point to the master, for switching we will choose checkout

- ▶ -> `git checkout branch_name`

We can create any change in one branch and observe that in different branch that change won't be reflected that is one of the most important feature of GIT.

- ▶ -> `git show HEAD`

Will show the commit that we have changed.

For creating branch that can create and switch at same time we type,

- ▶ -> `git checkout -b branch_name`

now if we type

- ▶ -> `git branch`

We see we have branch as well as we have switched to it also.

Branching: Other info

If we may make branch of new_feature, it will also have commit of new_feature that we have created not master because branch is made by it.

This below command will give all the commits in master as well as branches

► -> `git log --graph --oneline --decorate --all`

Also where the head is pointing to.

When the working directory is not clean or there is something in staging area then we won't be able to checkout of the branch.

In that case, you need to do first add in staging area and then commit changes.

But this doesn't tightly bound like we can add another file .txt but not change any existing file. So that no conflicts occurs.

Comparing the branches

The contents of two branches can be compared.

- ▶ -> `git diff master..branch_name`

Now if we want to see this in one line we can type

- ▶ -> `git diff --color-words new_feature..shorten_title`

We have taken example by taking two branch names.

We can also extract the only important change part:

- ▶ -> `git diff -color-words new_feature..shorter_title^`

Now if we are interested to know, which are the branches which have all the merge data

We can find it by:

- ▶ -> `git branch -merged`

Branching: Renaming and Deletion

We can rename the branch by:

- ▶ `git branch -m new_feature SEO_title`

So the branch name have been changed to `SEO_title`

For branch deletion we type:

- ▶ `-> git branch -d branch_name`

or

- ▶ `-> git branch --delete branch_name`

If you get any error message:

The branch 'branch_name' is not fully merged. It mean we have some changes to be merged in master that we haven't done.

If you are sure you want to delete it forcibly,

- ▶ `git branch -D branch_name`

Configuring the command prompt to show the branch name

Sometime we want that in Linux terminal, the main directory in which we are working on and the branch name then follow these steps(In windows GIT bash such feature is already there so no need to configure)

we need git completion file -> need to install `.git-completion.bash` and loaded in `.bashrc`

We need `__git_ps1`

- ▶ -> `echo $PS1` -> will return the name of computer
- ▶ -> `export PS1='-->'`

will return the terminal cursor default.

For formatting is such a way that it will shows current working directory as well as branch where we are in we type:

- ▶ -> `export PS1='\W$__git_ps1 "(%s)" > '`

But it is only active as long as we active the window.

So for permanent we need to write it down in `.git_profile` or `.bashrc` file

Merging

If any changes, we do in branch we need to merge in master so it is also available there.

First we should checkout to the receiver branch in our case it's master, let we need to merge seo_title branch.

- ▶ -> git checkout master
- ▶ -> git merge seo_title

The seo_title is the branch to merge in master branch

Now we can check the difference between two branches which will be null

- ▶ -> git diff master..seo_title
- ▶ -> git branch --merged

It will tell us that both branches is fully incorporated.

Merging: Fast forward Commit

If there is no changes in master and we merge its branch, then the merging is fast-forward and HEAD pointer simply move to branch commit. We can type

- ▶ -> `git log seo_title --oneline -3`
- ▶ -> `git log master --oneline -3`

Notice that top commit SHA is equal in both case.

If we don't want the fast -forward commit we will type

- ▶ -> `git merge --no-ff seo_title`

It will force to make commit and merge.

There are couple of merge option are:

- ▶ -> `git merge --ff-only seo_title` -> it will do fast forward only, if not do fast forward just abort the merge.

Otherwise the merge will happen in normal way i.e

- ▶ -> `git merge seo_title`

It will pop-up vim editor in which you give the commit message and save,

Merge Conflicts

Now in some cases if 2 developers in 2 different branch working on same file change some contents in the same lines simultaneously then the merge conflicts will arrive and git will show before merging that a conflict have arrived, the details of files can be seen on

► -> git status

We will open the conflicted file after opening a file we can see,

```
<<<<< HEAD
```

```
=====
```

```
>>>>> branch_name
```

So in this lines our merge conflicts have occurred and which lines we should have will be depend upon our choice.

Merge Resolving

Resolving:

- ▶ Abort the merge
- ▶ Resolve the conflicts manually -> mostly used.
- ▶ Merge tools

Now to abort the merge we use:

- ▶ -> `git merge --abort`

Now everything is clean.

But now if we want to do the edits manually:

- ▶ We type first -> `git merge branch_name`

Now we will be at merging state if any conflicts occur in that state check the status of the file. And open the conflicted file and you will see,

Merge Resolving (Continue..)

We need to choose same file between

<<<<HEAD

Or

>>>>Branch_name

Make all changes in any one and delete manually other also clear that << == >> marks.Hit commit

► -> git commit

a popup message will come delete the commit message and save.

Now everything is clean.

Now at end type:

► -> git log --graph --oneline --all --decorate

we will get nice representation of relation between branches, merges in visual format in terminal.

Merge Resolving with tools and strategy

For tools type:

- ▶ -> `git mergetool tool='NAME OF THE TOOL'`

The name of the tool will be given in help of the git.

For name just type

- ▶ `git mergetool --tool-help`

Strategy of merge conflicts

- ▶ Keep lines short.
- ▶ Keep commits small and focused
- ▶ Beware stray edits to whitespace -> spaces, tabs, line_returns
- ▶ Merge often
- ▶ Track changes to master -> The important one

Stashing Changes

The stash is a place where we can store changes temporarily without having to commit them to the repository. It's a lot like putting something into a drawer to save it for later. The stash is not part of the repository, the staging index or the working directory, it's a special fourth area in Git, separate from the others. And the things that we put into it aren't commits, but they're a lot like commits, they work in a very similar way. They're still a snapshot of the changes that we were in the process of making, just like a commit is. But they don't have a SHA associated with them.

For saving in stash make the changes in files and when you intend to checkout of the branch, it will popup to save or commit the changes.

Type for save in stash

► -> `git stash save "Changes in file"`

Check git log for the swap SHA

Stashing Changes (Continue..)

For view stash changes:

- ▶ -> `git stash list`

For all the stash list available.

This will give the available stash, where we can always pull from it.

For viewing or inspect the changes type:

- ▶ -> `git stash show stash@{0}`

For getting more details we type

- ▶ -> `git stash show -p stash@{0}`

Stash Retrieval

The retrieval of stash can happen in any branch, git ensure it in most optimized way

There are two ways to get the items out of stash

- ▶ -> `git stash pop`

or

- ▶ -> `git stash apply`

We need to specify which stash we need to pop out otherwise by default it will pops out last one.

- ▶ -> `git stash pop stash@{0}`

For different stash we have different number in stash so we can choose such like that.

Now after pop we type `git stash list` again, we won't get the details of popped out stash.

How to delete the stash Item

For delete we type:

- -> `git stash drop stash@{0}`

Or whatever number stash we have.

But if we want to clear everything in the stash including last recorded we can do it in:

- -> `git stash clear`

Working with Remote Repository

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. In this section, we'll cover some of these remote-management skills.

First we need to login in github and create a repository and in Download section copy the Link of your git let suppose <https://Repository-folder.git>.

For uploading your branch we need to add the remote url in our git project for that type,

► -> git remote add origin <https://Repository-folder.git>

We can call or rename the remote repository to origin or some other also.

Remote origin

For listout all the origin we type:

- ▶ -> `git remote`

or for the details of URL we type:

- ▶ -> `git remote -v`

We can view all the details in

- ▶ -> `cat .git/config`

For remove the origin we type

- ▶ -> `git remote rm origin`

We can add remote location again.

Pushing to Remote Repository

The git push command let you upload your local repository to the remote repository.

For pushing we use:

- -> `git push -u origin branch_name`

In our case it's master so,

- -> `git push -u origin master`

It will ask password and username for your account.

Cloning a repository

We can clone any repository in remote machine type:

- ▶ -> git clone <git http link>

like,

- ▶ -> git clone <https://github.com/mdrijwan123/Duval-s-Triangle-DGA-Diagnosis.git>

but sometime we already have that repository in our computer in that case we can give the name to our repository but all the files will remain same.

- ▶ -> git clone <https://github.com/mdrijwan123/Duval-s-Triangle-DGA-Diagnosis.git> Duval_fault

The duval fault is the name of repository we can give, in your case it will be different.

- ▶ By checking ls -la we can see the folder there.

Tracking remote branches

If we don't do git push with the -u option, it does not track any remote branch. All it does is push our code up there, and that's it. It doesn't keep any kind of reference that this is the branch there we're going to be working with in the future. The -u option says push it up there, and also make a note of the fact because we're going to be coming back and working with this branch frequently.

When we clone any repository it is also tracking the master branch,

Same can be checked with

- ▶ -> `cat .git/config`

For making the untracked file tracking with master on github we can change config as:

Lets we have our branch name as non_tracking and remote as origin:

- ▶ -> `git config branch.non_tracking.remote origin`

For merge:

- ▶ -> `git config branch.non_tracking.merge refs/heads/master`

This will give same tracking configuration in .git/config file.

Pushing changes to remote repository

We will apply that while we also push some more changes up to the remote repository that we created. Change the contents of file in working directory and commit. The change we have done is in local machine, we want to push the change to local repository. So,

We can track the commit as,

- ▶ -> `git log --oneline`

for remote link we do,

- ▶ -> `git log --oneline origin/master`

That is the copy of remote branch that is tracking with the GitHub or GitLab repository, We can compare the changes with,

- ▶ -> `git diff origin/master..master`

So we can track the changes. So for push the changes we type

- ▶ -> `git push`

Now change can be shown in

- ▶ -> `git log --oneline origin/master`

as well as in GitHub or GitLab

Fetching changes from remote Branch

If we independently working on one branch and push the changes on remote branch, in other branch we need to fetch the changes so that it should be synchronized master/origin with remote repository so that we can push the changes in that branch. We can do that in fetch command through git,

We can check the remote branches we have using

- ▶ `git branch -r`

Without sync it will not show us the branches pushing in git repo from other branch for that we need to pass command:

- ▶ `-> git fetch <remote-name>`

In our case remote name is origin, so

- ▶ `-> git fetch origin`

Now the branch is synched with the remote repository now we can see the changes in,

- ▶ `git branch -r`

Recommendation for working in GIT

Fetch, Merge etc.

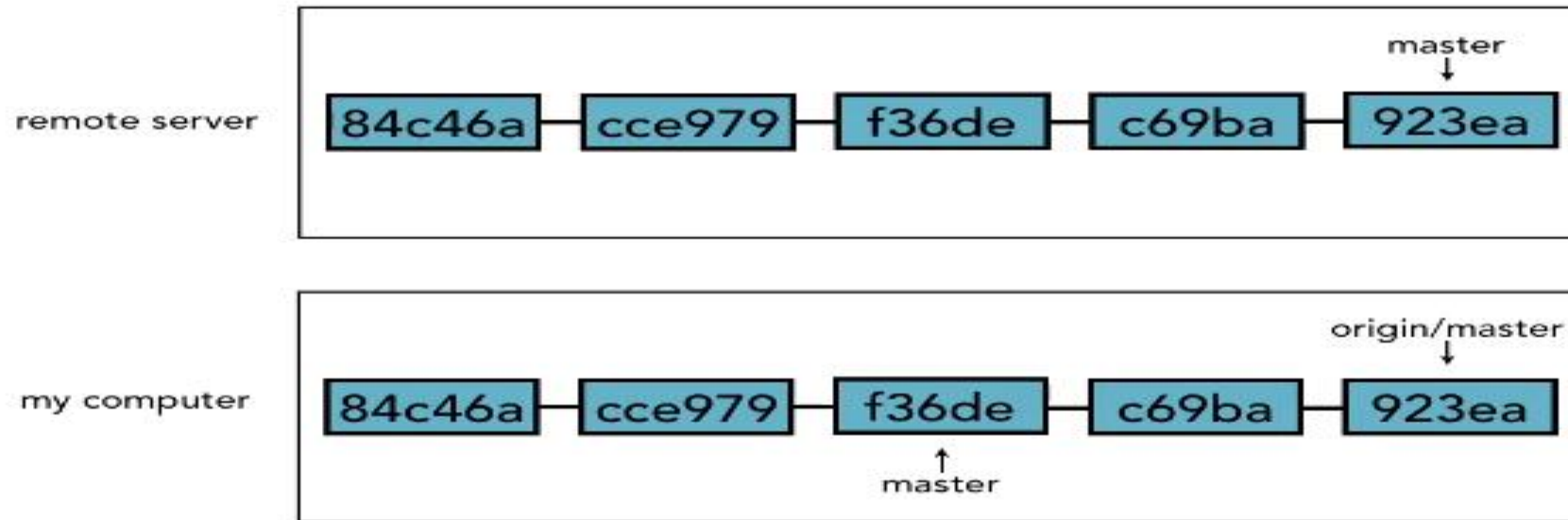
So the recommendation for working git with remote repository is:

- ▶ Fetch before you work
- ▶ Fetch before you push
- ▶ Fetch often

But until now we have only updated the origin/master synched with the remote repository we still not have updates on our working branch, the fetch is something like synching the changes in cache but not yet have been merged with the local repository, that we can do with merge.

Merging in Fetched changes

In this illustration, you can see that the remote server has five commits in it, and a fetch has taken place, because origin/master is perfectly in sync with what's on the remote server. Our master branch however is two commits behind, so it needs to have those two commits added to it, and the process that we do that is by merging.



Merging in Fetched changes (Continue..)

For checking the all branches we have we will use:

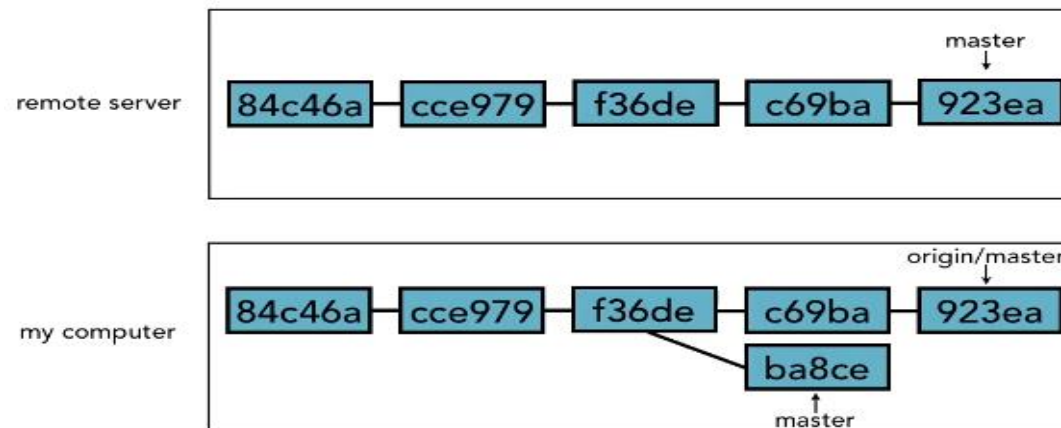
- -> `git branch -a`

We can see the difference by:

- -> `git diff origin/master..master`

Notice the changes, so if we want to go with merge type:

- -> `git merge origin/master`



Merging in Fetched changes

Some Recommendations

Our local version, that's where we are merging in, we are not going back up to GitHub or GitLab to see what's there. So when we do a merge, we always want to make sure we do a fetch first. Fetch, then merge. We do want to do our best to be as up-to-date as possible, `git fetch`, `git merge`. That's the process that we are going to go through.

Now git also provide a shortcut called `git pull`. `Git pull` is equal to `git fetch+git merge`.

The bad part is, that for beginners, it obscures the fact that we are actually doing this two step process, and lot of people who use `git pull` a lot, forget about doing `git fetch` and `git merge`. We don't really realize that what's it doing for them under the hood, and then sometimes when things go wrong with `git pull`, it's hard to understand what actually went wrong and what happened and how to fix it.

If we use `git fetch+git merge`, it's much harder for those kinds of problems to arise, because it's much clear what went wrong and how to fix it. So recommend is to try and use `git fetch` and `git merge` for a while, until we feel like we've got the hang of it and then if you know that what we really want is just `git fetch` and pull down the changes, then you can try using `git pull`.

Checkout Remote branches

To switch to Remote branch:

- ▶ -> `git branch <branch_name> origin/<branch_name>`

Let suppose we have non_tracking branch so,

- ▶ -> `git branch non_tracking origin/non_tracking`

It's a tracking branch that will track, origin/non_tracking We can check in

- ▶ -> `git branch`

or

- ▶ -> `cat .git/config`

To see the tracking information.

To delete branch

To delete the branch we type:

- ▶ -> `git branch -d non_tracking`

Or in simply type:

- ▶ `git checkout -b non_tracking origin/non_tracking`

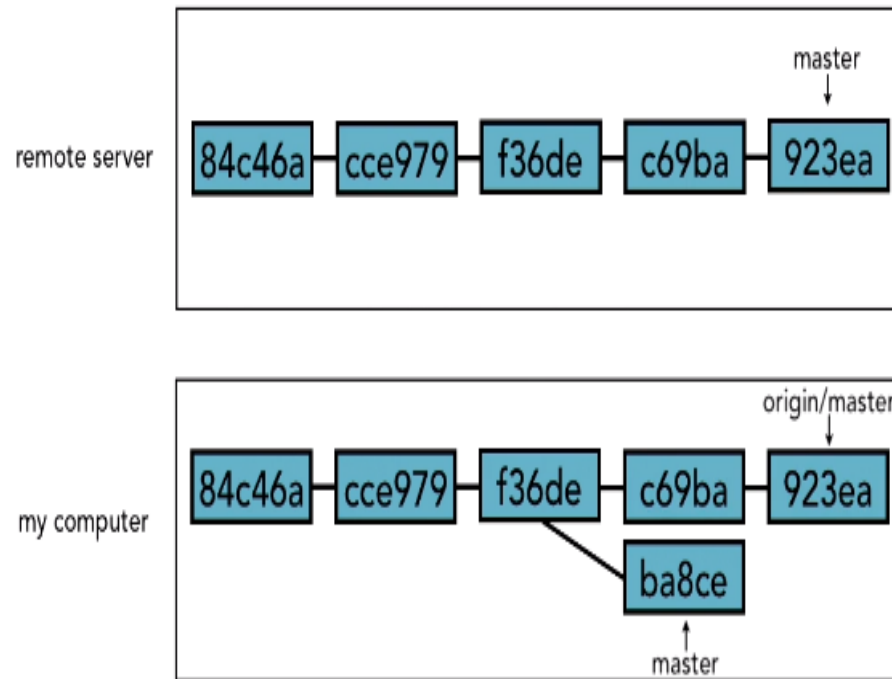
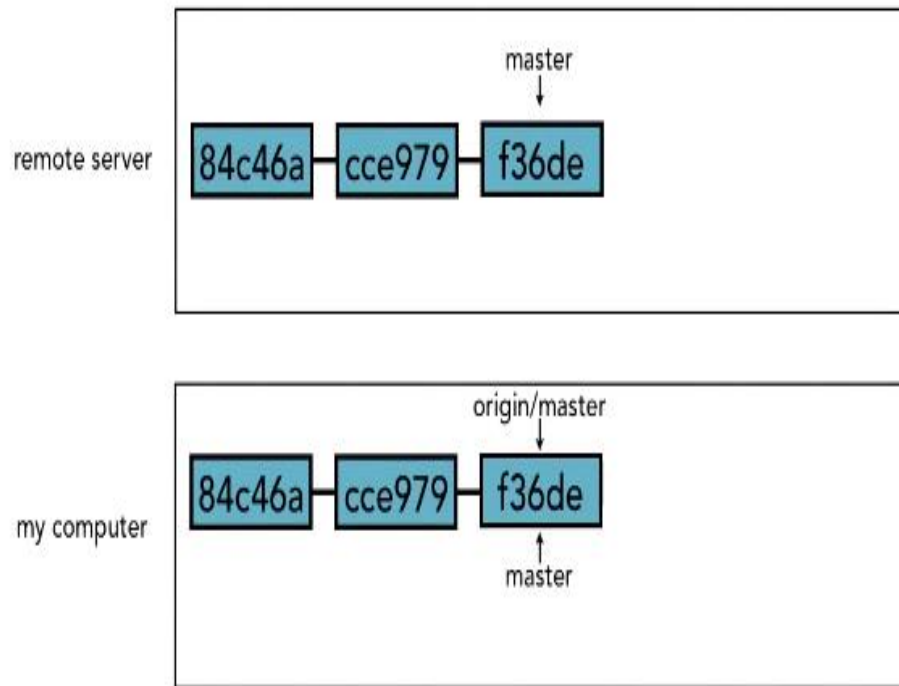
It will give the copy of non_tracking in our computer.

Now we can make changes and push in the non_tracking remote branch.

For checkout in master type

- ▶ -> `git checkout master`

Pushing to updated remote branch



If such situation occurs, then we can't push the remote server, you'll just need to fetch, then merge, then push again. That way we can make sure that everyone's changes are taken into account.

Delete a remote branch

The first way is to use `git push` and just like we did before when we are using `git push`, we push to the remote, and when we did it before, we did it like this, we said, push the contents of `non_tracking` up to the server. In order to delete it, you put a colon in front of it. `Git push origin`, and a `:non_tracking` will have the effect of deleting the branch on the remote server.

► -> `git push origin :non_tracking`

Now if we check `git branch -r`

The remote branch is gone.

Notice that if we do `git branch` for our local branches, it's still here. All I did was a push up to origin, I didn't do anything to my local branches, so I still have all that information. Why this awkward non-intuitive way of doing it? What's the colon all about? When we did our original push of the branch up to origin, we did it like this, `git push origin/non_tracking`. That's actually shorthand for `git push origin non_tracking:non_tracking`.

Delete a remote branch

What this is saying is push to origin my local branch `non_tracking`, to the remote branch called `non_tracking`. That's what it's doing. When we only have one, it assumes that they are the same, which they often are. But this colon divides those two. So when we are doing a delete, what you are actually doing is saying, push to origin nothing up to the branch `non_tracking`. So that's why that colon is there. That's where it comes from. But that's not very intuitive.

The other way is:

- -> `git push origin --delete non_tracking`

Git Cheat Sheet



Git Basics

<code>git init</code> <code><directory></code>	Create empty Git repo in specified directory. Run with no arguments to initialize the current directory as a git repository.
<code>git clone <repo></code>	Clone repo located at <code><repo></code> onto local machine. Original repo can be located on the local filesystem or on a remote machine via HTTP or SSH.
<code>git config</code> <code>user.name <name></code>	Define author name to be used for all commits in current repo. Devs commonly use <code>--global</code> flag to set config options for current user.
<code>git add</code> <code><directory></code>	Stage all changes in <code><directory></code> for the next commit. Replace <code><directory></code> with a <code><file></code> to change a specific file.
<code>git commit -m</code> <code>"<message>"</code>	Commit the staged snapshot, but instead of launching a text editor, use <code><message></code> as the commit message.
<code>git status</code>	List which files are staged, unstaged, and untracked.
<code>git log</code>	Display the entire commit history using the default format. For customization see additional options.
<code>git diff</code>	Show unstaged changes between your index and working directory.

Undoing Changes

<code>git revert</code> <code><commit></code>	Create new commit that undoes all of the changes made in <code><commit></code> , then apply it to the current branch.
<code>git reset <file></code>	Remove <code><file></code> from the staging area, but leave the working directory unchanged. This unstages a file without overwriting any changes.
<code>git clean -n</code>	Shows which files would be removed from working directory. Use the <code>-f</code> flag in place of the <code>-n</code> flag to execute the clean.

Rewriting Git History

<code>git commit</code> <code>--amend</code>	Replace the last commit with the staged changes and last commit combined. Use with nothing staged to edit the last commit's message.
<code>git rebase <base></code>	Rebase the current branch onto <code><base></code> . <code><base></code> can be a commit ID, a branch name, a tag, or a relative reference to HEAD.
<code>git reflog</code>	Show a log of changes to the local repository's HEAD. Add <code>--relative-date</code> flag to show date info or <code>--all</code> to show all refs.

Git Branches

<code>git branch</code>	List all of the branches in your repo. Add a <code><branch></code> argument to create a new branch with the name <code><branch></code> .
<code>git checkout -b</code> <code><branch></code>	Create and check out a new branch named <code><branch></code> . Drop the <code>-b</code> flag to checkout an existing branch.
<code>git merge <branch></code>	Merge <code><branch></code> into the current branch.

Remote Repositories

<code>git remote add</code> <code><name> <url></code>	Create a new connection to a remote repo. After adding a remote, you can use <code><name></code> as a shortcut for <code><url></code> in other commands.
<code>git fetch</code> <code><remote> <branch></code>	Fetches a specific <code><branch></code> , from the repo. Leave off <code><branch></code> to fetch all remote refs.
<code>git pull <remote></code>	Fetch the specified remote's copy of current branch and immediately merge it into the local copy.
<code>git push</code> <code><remote> <branch></code>	Push the branch to <code><remote></code> , along with necessary commits and objects. Creates named branch in the remote repo if it doesn't exist.

Additional Options +

git config

<code>git config --global user.name <name></code>	Define the author name to be used for all commits by the current user.
<code>git config --global user.email <email></code>	Define the author email to be used for all commits by the current user.
<code>git config --global alias.<alias-name> <git-command></code>	Create shortcut for a Git command. E.g. <code>alias.glog log --graph --oneline</code> will set <code>git glog</code> equivalent to <code>git log --graph --oneline</code> .
<code>git config --system core.editor <editor></code>	Set text editor used by commands for all users on the machine. <code><editor></code> arg should be the command that launches the desired editor (e.g., vi).
<code>git config --global --edit</code>	Open the global configuration file in a text editor for manual editing.

git log

<code>git log -<limit></code>	Limit number of commits by <code><limit></code> . E.g. <code>git log -5</code> will limit to 5 commits.
<code>git log --oneline</code>	Condense each commit to a single line.
<code>git log -p</code>	Display the full diff of each commit.
<code>git log --stat</code>	Include which files were altered and the relative number of lines that were added or deleted from each of them.
<code>git log --author="<pattern>"</code>	Search for commits by a particular author.
<code>git log --grep="<pattern>"</code>	Search for commits with a commit message that matches <code><pattern></code> .
<code>git log <since>..<until></code>	Show commits that occur between <code><since></code> and <code><until></code> . Args can be a commit ID, branch name, HEAD, or any other kind of revision reference.
<code>git log -- <file></code>	Only display commits that have the specified file.
<code>git log --graph --decorate</code>	<code>--graph</code> flag draws a text based graph of commits on left side of commit msgs. <code>--decorate</code> adds names of branches or tags of commits shown.

git diff

<code>git diff HEAD</code>	Show difference between working directory and last commit.
<code>git diff --cached</code>	Show difference between staged changes and last commit

git reset

<code>git reset</code>	Reset staging area to match most recent commit, but leave the working directory unchanged.
<code>git reset --hard</code>	Reset staging area and working directory to match most recent commit and overwrites all changes in the working directory.
<code>git reset <commit></code>	Move the current branch tip backward to <code><commit></code> , reset the staging area to match, but leave the working directory alone.
<code>git reset --hard <commit></code>	Same as previous, but resets both the staging area & working directory to match. Deletes uncommitted changes, and all commits after <code><commit></code> .

git rebase

<code>git rebase -i <base></code>	Interactively rebase current branch onto <code><base></code> . Launches editor to enter commands for how each commit will be transferred to the new base.
---	---

git pull

<code>git pull --rebase <remote></code>	Fetch the remote's copy of current branch and rebases it into the local copy. Uses git rebase instead of merge to integrate the branches.
---	---

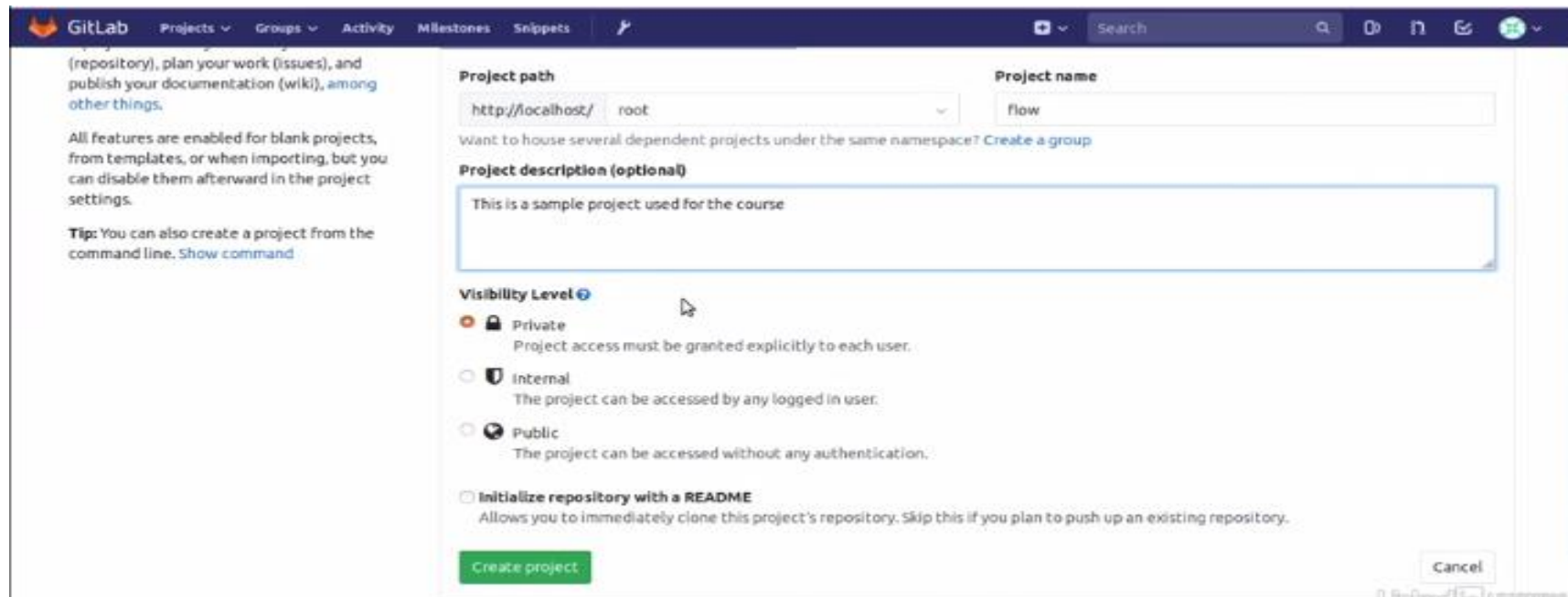
git push

<code>git push <remote> --force</code>	Forces the <code>git push</code> even if it results in a non-fast-forward merge. Do not use the <code>--force</code> flag unless you're absolutely sure you know what you're doing.
<code>git push <remote> --all</code>	Push all of your local branches to the specified remote.
<code>git push <remote> --tags</code>	Tags aren't automatically pushed when you push a branch or use the <code>--all</code> flag. The <code>--tags</code> flag sends all of your local tags to the remote repo.

Appendix: Hands On GitLab

Case1: Overview and first repository

- ▶ This slide will provide a high-level overview of GitLab and shows you where to find major features within the tool. Projects are a core concept within GitLab. Once you establish a project, GitLab provides a wide set of features such as a code repository, issue trackers, wikis, and CI/CD tooling to support the project.
- ▶ Click on the create project and the first thing that we'll need to do is provide a name for our project. I'm simply going to call this project flow.



The screenshot shows the GitLab web interface for creating a new project. The top navigation bar includes links for Projects, Groups, Activity, Milestones, and Snippets. The main form is titled 'Create project' and contains the following fields and options:

- Project path:** A dropdown menu showing 'http://localhost/' and 'root'.
- Project name:** A text input field containing 'flow'.
- Project description (optional):** A text area containing 'This is a sample project used for the course'.
- Visibility Level:** A section with three radio button options:
 - ☒ **Private**: Project access must be granted explicitly to each user.
 - ☐ **Internal**: The project can be accessed by any logged in user.
 - ☐ **Public**: The project can be accessed without any authentication.
- ☐ **Initialize repository with a README**: Allows you to immediately clone this project's repository. Skip this if you plan to push up an existing repository.

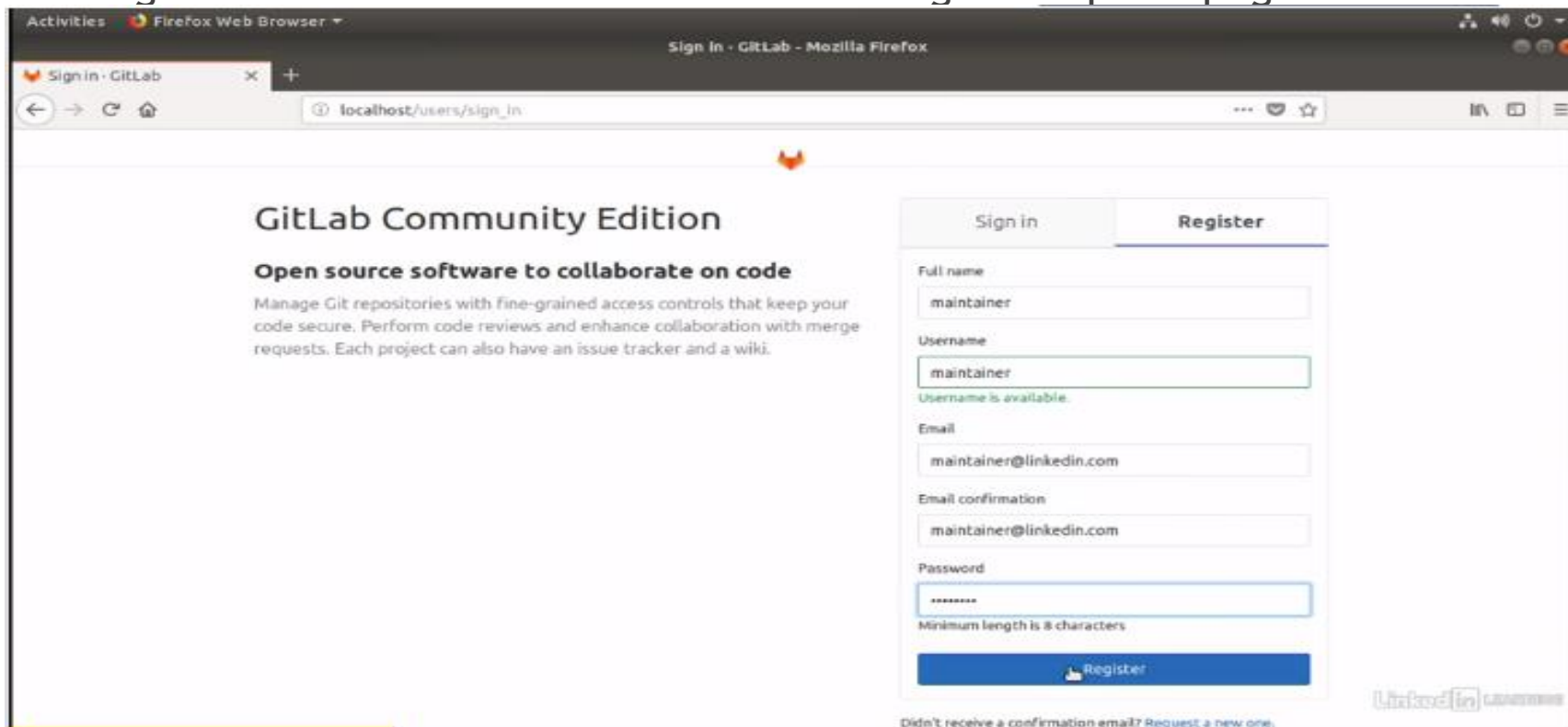
At the bottom of the form are two buttons: 'Create project' (green) and 'Cancel' (grey).

Case1: Overview and first repository

- ▶ We can specify a visibility level for the project. This will determine how exposed your project is to other users.
- ▶ We select private project that will only be available to users we grant access, explicitly, to the project. When we build out a project, we can also choose to initialize a README file within the repository. And create project.
- ▶ You'll notice the left-hand side bar contains a repository. This is where our code will be stored for the project. The interface allows you to look through the different branches found within the repository from the main screen, or you can find the branches on the side bar. Additionally, you can look at the code from various different aspects. You can see the latest commits that were provided to your project.

Case 3: Create user account set-up

- ▶ When starting out with GitLab, the first thing a team should do is setup user accounts. User accounts allow team members to log into the tool and are used to control their access and privileges on specific projects.
- ▶ Within GitLab there's two way to create an account. First, a team member can register for an account on their own through the splash page for the tool.



The screenshot shows the GitLab Community Edition splash page in a Mozilla Firefox browser window. The browser's address bar shows the URL `localhost/users/sign_in`. The page features the GitLab logo and the text "GitLab Community Edition" and "Open source software to collaborate on code". Below this, a brief description of GitLab's capabilities is provided. On the right side, there is a registration form with two tabs: "Sign in" and "Register". The "Register" tab is active, showing fields for "Full name", "Username", "Email", "Email confirmation", and "Password". The "Full name" field contains the text "maintainer". The "Username" field also contains "maintainer", and a green message "Username is available." is displayed below it. The "Email" field contains "maintainer@linkedin.com", and the "Email confirmation" field also contains "maintainer@linkedin.com". The "Password" field is masked with asterisks, and a note "Minimum length is 8 characters" is shown below it. A blue "Register" button is at the bottom of the form. At the very bottom of the page, there is a link: "Didn't receive a confirmation email? Request a new one."

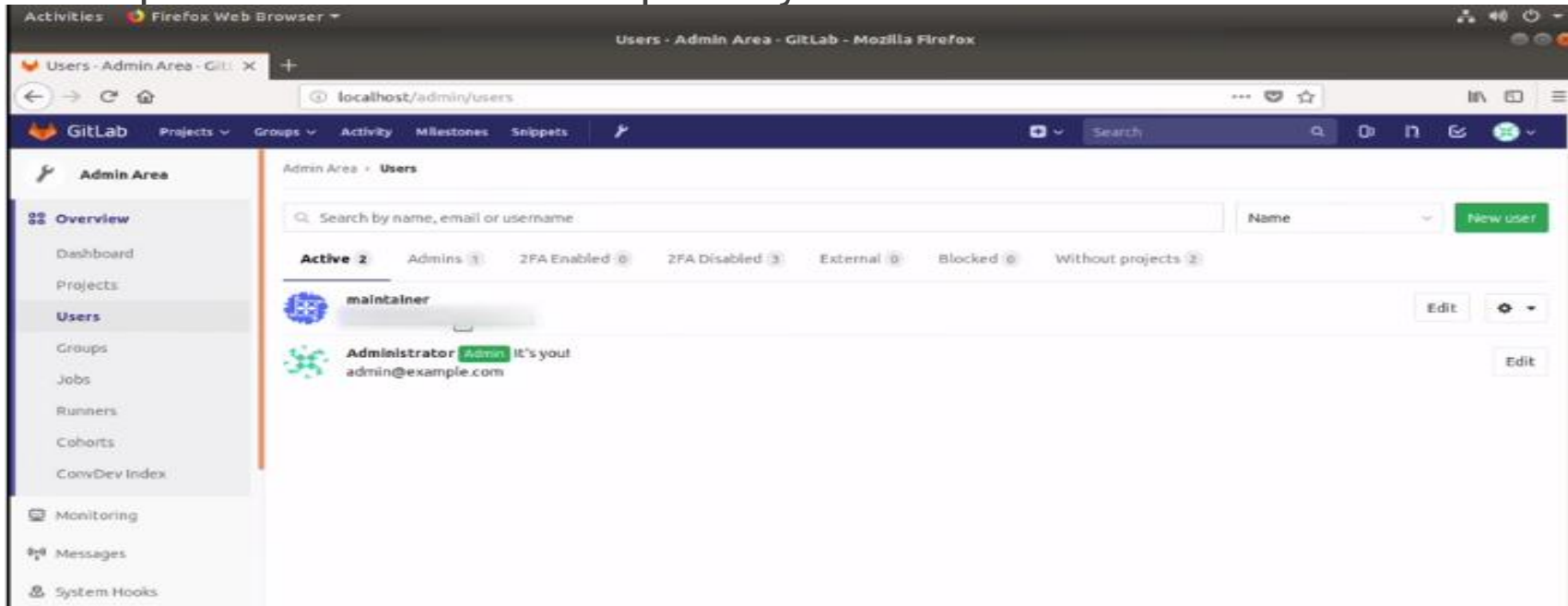
Case 3: Create user account set-up

- ▶ So what we're going to do is build a maintainer account.
- ▶ The maintainer account is going to represent our team member who has commit privileges on every branch within the repository. They're going to be the folks performing the merge request from the contributors. So we go ahead and set up our maintainer account.
- ▶ Another way is by administrator. So an administrator can create an account for a new user. So we're going to log in as root and once we're logged in we're going to access the administration area for GitLab by clicking the little wrench at the top of the screen.



Case 3: Create user account set-up

- If we take a look now at the sidebar, you're going to see the user's section appear. We already see the maintainer account that we've created, and we're going to go ahead and build a new user account, this time for a contributor. The contributor account is going to represent those developers on our team that lack commit access and need to work through a maintainer in order to have their feature branches integrated within the major development branches in our repository.



Case 3: Create user account set-up

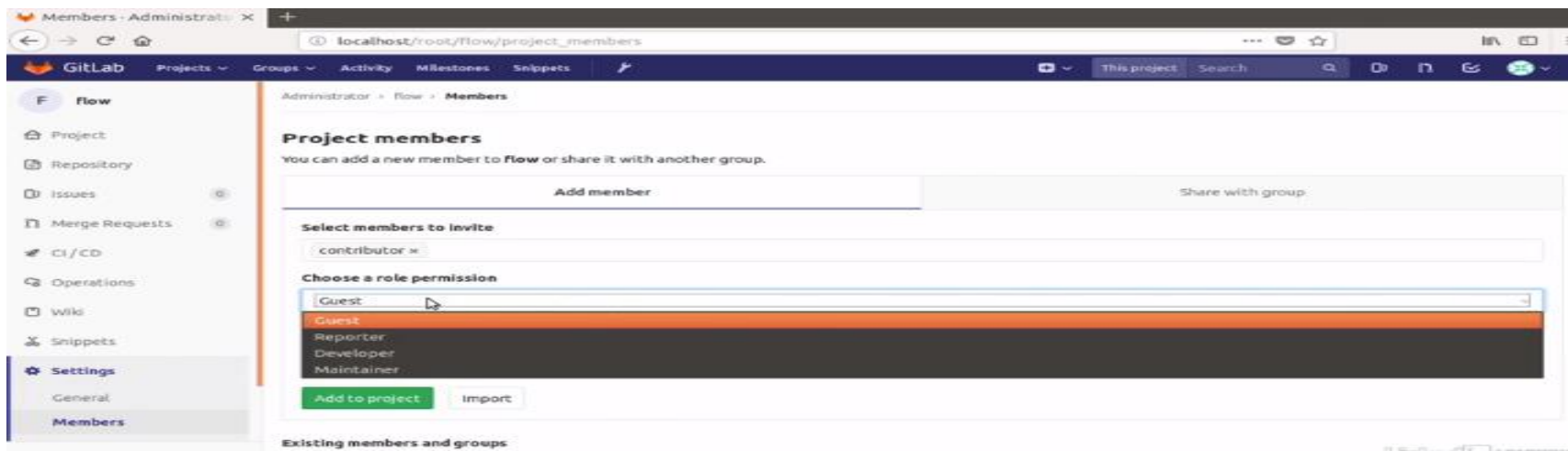
- So we just need once again to provide simple pieces of information. The name, the username and the email for contributor. From there if we'd like we can set up additional access for this user. We can make them an administrator if we want them to have higher level privileges, however for this user, they can stay as a regular user or one of developers on our project. Once we've completed the form, all we have to do is click the create user button at the bottom of the screen and we see that the new user is established.



The screenshot shows the GitLab Admin Area 'New User' form in a Firefox browser. The browser's address bar shows 'localhost/admin/users/new'. The left sidebar contains the 'Admin Area' menu with options: Overview, Dashboard, Projects, Users (selected), Groups, Jobs, Runners, Cohorts, and ConvDev Index. Below this are Monitoring, Messages, System Hooks, and a Collapse sidebar button. The main content area is titled 'New user' and contains three sections: 'Account', 'Password', and 'Access'. The 'Account' section has three required fields: 'Name' (containing 'contributor'), 'Username' (containing 'contributor'), and 'Email' (empty). The 'Password' section has a 'Password' label and a message: 'Reset link will be generated and sent to the user. User will be forced to set the password on first sign in.' The 'Access' section is partially visible at the bottom. The bottom right of the page shows social media links for GitHub, LinkedIn, and Twitter.

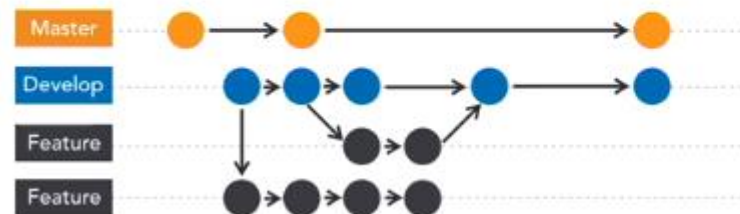
Case 3: Create user account set-up

- ▶ Within the flow project we can scroll down on the sidebar and under settings we find the member's area. This allows us to assign different users to the project.
- ▶ So let's first assign the contributor to the project and we need to select a permission level for this user on the project. We're going to go ahead and make our contributor a developer. This will give them push permissions and commit permissions on branches within the repository that are not protected. Once we've established their permission level, we can click on add to project and they're now assigned.
- ▶ Similarly we can assign the permission for maintainer as Maintainer.



Case 4: Branching Strategies

- ▶ Git's branching model is one of the most powerful available amongst version control systems. Its flexibility allows teams to easily create various types of branches that establish workflows.
- ▶ Using Git branches, we can quickly fork the code base to work on a new feature while maintaining the original source code in a separate line of code. So, this allows us to experiment on a new feature in a separate area. Depending on our workflow, we may need to leverage several types of branches. Let's talk about those now. First, let's discuss long-running branches. Long-running branches always remain open, storing the history of a particular line of development.



Feature Branches

- Branches created for a particular feature
- Stores work related to the feature for a short time

Thank you!