First we need to config the git after installing:

git config --global user.name “Name”

git config --global user.email [some@gmail.com](mailto:some@gmail.com)

Now we can see the config file by

ls -la

also by,

git config --list

or cat .gitconfig

Now we have configured git for first time use.

For test editor we use Textmate

git config --global core.editor “mate –wl1”

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

hit enter:

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 nano text editor to edit file in .bashrc

nano .bashrc then write

if [ -f ~/.git-completion.bash]; then

source ~/.git-completion.bash

fi

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.

Initialising GIT Repository

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

* mkdir first\_git\_project
* git init

create some sample file like

echo “The first project” > some.txt

git add .

This will add all changes to this entire project.

And commit the changes

git commit –m “Initial commit”

To see the log of all the commit

* git log

git log –n 1

will limit the number of commit.

Or,

git log --since=2012-06-12

all logs after date

same until does same work.

git log –author=”Rijwan”

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

will tell the difference of all the working directory-> staging index -> repository

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

Status of these three.

Adding 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.

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 diiference of staging index we type:

git diff --staged

To remove file we use

git rm file.txt

It will delete all the file like shift+delete in unix model 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

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

git commit --am “Commit all”

--am means adding and committing

GIT undo the changes:

For undo in present working directory use --

git checkout -- index.html

the file name or folder name is any.

Now if we modified the file in staging stage then,

We can undo it using:

git reset HEAD resource.html

or any file name

and you can do checkout.

Now how to undo the commit changes.

git checkout ‘Hexdecimal no.’ resource.html

hex can be obtain in git log files

and we can commit those changes.

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

The hex code of commit

Will open vim edit and save close.

Undo multiple commits:

**.Gitignore file**

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

We use !,# or other expression to do that.

\*.txt

If we write in it then all txt file will get ignored in git.

Now if we put # sign

This will only comment the line and the file actually reflect in git written in #

**Branching**

git branch

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

git log --oneline

It will give the oneline commit statement of all.

For creating new branch type:

git branch branch\_name ->only one time for branch creation

then check branch using

git branch

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 doesn’t reflect.

**Create and switch branches at the same time.**

git show HEAD

will show the commit that we have changed.

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

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.

**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.**

Also 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

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

git diff --color-words new\_feature..shorten\_title

we can also extract the only important change part:

git diff –color-words new\_feature..shorter\_title^

Now which are the branches which have all the merge data

We can find it by:

git branch –merged

**Renaming the branches**

We can rename the branch by:  
git branch –m new\_feature SEO\_title

So now the name changed to SEO\_title

**Branch delete**

\*\* Use – when we use words, but for single word we use - one hyphen.

git branch -d branch\_name

or

git branch --delete branch\_name

The error message:

The branch 'branch\_delete' is not fully merged.

If you are sure you want to delete it, run 'git branch -D branch\_delete'.

It mean we have not merged the files in master branch so it shows warning but still we can delete it.

git branch -D branch\_delete

**Configuring the command prompt to show the branch name**

**For linux :**

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

For windows we can set the same but creating .git\_profile at the user directory and follow the same procedure.

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

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.

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

to see 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,

**Merger Conflicts**

Now if we change some content in any branches 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 get after opeining a file

<<<<<< HEAD

======

>>>>>>> branch\_name

So in this lines our merge conflicts have occurred and which lines we should have will be depend upon our choice, that is merge conflicts resolving we called.

Resolving:

1. Abort the merge
2. Resolve the conflicts manually -> mostly used.
3. 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,

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.

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**

1. Keep lines short.
2. Keep commits small and focused
3. Beware stray edits to whitespace -> spaces,tabs,line\_returns
4. Merge often
5. 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 dont 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

**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}

Like typically diff commands

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

1. git stash pop
2. 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

**Remote repository**

For uploading your branch we will type:

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

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

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.

**Push**

The git push command is used to upload local repository content to a remote repository. Pushing is how you transfer commits from your local repository to a remote repo.

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 ans 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,

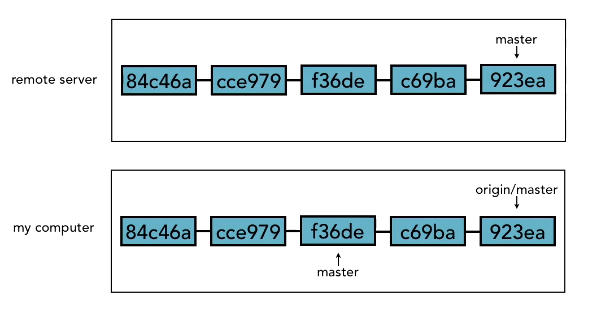
git branch –r

So the recommendation for working git with remote repository is:

1. Fetch before you work
2. Fetch before you push
3. 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 some-thing 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, anda 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.

For checking the all branches we have we will use:

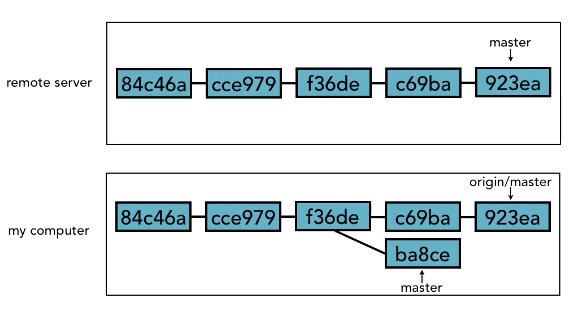
git branch –a

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



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 the branch we type:

git branch –d non\_tracking

Or in simply type:

git checkout –b non\_tracking origin/non\_tracking

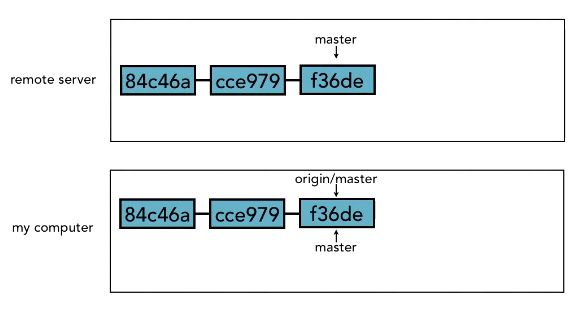
It will give the copy of non\_tracking in our computer.

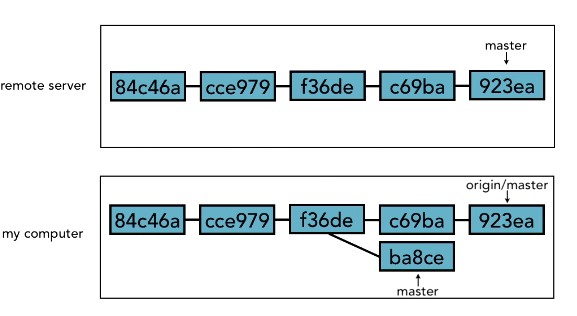
Now we can make changes and push in the non\_tracking remote branch.

For checkout in master type

git checkout master

**Pushing to updated remote branch**





If 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 it 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.

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