

1 Initial-Git

Initialize git on your machine:

```
1 git init
```

2 Basic-Git

Some basic git commands:

```
1 git add . # add unstaged file to staged
2 git restore <filename> # restore staged file to unstaged
3 git commit -m "Your_commit_message" # commit the staged file
4 git push # push files
```

3 Git Config

Configure commit history:

```
1 # global
2 git config --global user.name "your_name"
3 git config --global user.email "your_email"
4 # local
5 git config --local user.name "your_name"
6 git config --local user.email "your_email"
7 # check config lists
8 git config --global --list
9 git config --local --list
```

Global configs are located here : ~/.gitconfig and the Local configs are located at the current directories: .git/config

4 Some Common Git Command

There are some common git commands which we use regularly

```
1 git branch #to check branches
2 git status #to check staged and unstaged
3 git checkout #changing branch
4 git log #to check the commit history
5 git checkout -b "branch_name" #to create a new branch from main branch
```

5 SSH Keygen

SSH keygen is most important to connect the local server to the remote server. There are few steps to create SSH key and add it to the remote server .

5.1 Create SSH key

```
1 ssh-keygen -t rsa -b 2048 -C "comment"
```

This will show you a location where to save the ssh key .To save it on the default location `~/.ssh/id_rsa.pub` press ENTER, ENTER and ENTER . Here the `id_rsa.pub` is your SSH public key. Open the key, copy it and add it to github settings SSH key section .

5.2 Add To Remote

After adding the SSH key to remote , You can now connect your local machine to remote server.

5.2.1 Cloning Github/Gitlab

Go to a particular directory where you want to clone the remote git

```
1 git clone git@gitlab.com:group.project.git
```

5.2.2 Add Remote and Push

On the other hand, you work in your local machine and now you want to add your project to remote

```
1 git remote add origin git@gitlab.com:group.project.git
```

5.3 Now The Big Problem Arise

Suppose I have two GitHub accounts:

- `git@github.com:user1/project1.git`
- `git@github.com:user2/project2.git`

Earlier you saw in the previous command that i also have a GitLab account.

- `git@gitlab.com:group.projetc.git`

which i already connected to my local machine using SSH, and its public key is added in `id_rsa.pub` . Now i want to connect these two new GitHub account with SSH without any problem **Procedure**

```
1 #key generate for user1
2 ssh-keygen -t rsa -b 2048 -C "user1" -f ~/.ssh/id_rsa_user1
3 #key generate for user2
4 ssh-keygen -t rsa -b 2048 -C "user2" -f ~/.ssh.id_rsa_user2
```

Now there will be three key which are

- `~/.ssh/id_rsa.pub` which is for GitLab
- `~/.ssh/id_rsa_user1.pub` which is for GitHub user1
- `~/.ssh/id_rsa_user2.pub` which is for GitHub user2

Open a config file and add these line in it:

```
1 nano ~/.ssh/config #open config file
```

```

1 # GitLab account
2 Host gitlab
3   HostName gitlab.com
4   User git
5   IdentityFile ~/.ssh/id_rsa
6
7 # GitHub account user1
8 Host github-user1
9   HostName github.com
10  User git
11  IdentityFile ~/.ssh/id_rsa_github_user1
12
13 # GitHub account user2
14 Host github-user2
15   HostName github.com
16   User git
17  IdentityFile ~/.ssh/id_rsa_github_user2

```

Check the HostName carefully, cause for git clone and git remote add you need to add this hostname properly. Now open and copy the each SSH key and add them to their account

```

1 cat ~/.ssh/id_rsa.pub #add it to GitLab account
2 cat ~/.ssh/id_rsa_github_user1.pub #add it to GitHub user1 account
3 cat ~/.ssh/id_rsa_github_user2.pub #add it to GitHub user2 account

```

Now you can clone them singly and also add them in remote . You will see the HostName of config file will add after git@ part .

```

1 #for GitLab the HosName was gitlab
2 git clone git@gitlab:group/project.git
3
4 #for GitHub user1 the HostName was github-user1
5 git clone git@github-user1:user1/project1.git
6
7 #for GitHub user2 the HostName was github-user2
8 git clone git@github-user2:user2/project2.git

```

```

1 #for GitLab
2 git remote add origin git@gitlab:group/project.git
3
4 #for GitHub user1
5 git remote add origin git@github-user1:user1/project1.git
6
7 #for GitHub user2
8 git remote add origin git@github-user2:user2/project2.git

```

6 Fork Git Branch, Edit and Merge Request

Fork the main repository

```

1 git@gitlab.com/main/merge.git #suppose its the main repository
2 git@gitlab.com/fork/merge.git #forked repository

```

Clone your fork which you added to your gitlab:

```
1 git clone git@gitlab.com/fork/merge.git
2 cd merge
```

Here origin is automatically set to your fork (git@gitlab.com/fork/merge.git).

Add the main repository to upstream:

```
1 git remote add upstream git@gitlab.com/main/merge.git #upstream add
2 git remote -v #check origin and upstream set
```

Create a new branch for your feature

```
1 git checkout -b new-branch
```

Always work on your feature branch , never directly on master.

After completing edit, add, commit, push your feature branch to your fork(origin)

```
1 git push origin new-branch
```

It will create the branch in your remote fork. Then you need to create Merge Request to your fork master. Whenever merging the branch, it will automatically delete the branch from your remote fork repository. Then Create a Merge Request to the main repository from where you forked the repo.

Now there are some point:

- The commit history of main repository is ahead of your fork repository
- The commit history of your fork repository is behind of your main repository
- If you without doing anything try to push , it will shows conflict

Sync fork with main repository after Merge Request

```
1 git checkout master #comeback to the master
2 git branch -d new-branch #delete feature branch
3 git fetch upstream #fetch latest upstream change
4 git rebase upstream/master #rebase
5 git push #push updated master to your fork
```

Now the commit history of your fork repository will be same as your main repository from where you forked it .

7 Resolve Conflict when two or more person changing the same line

Its from the initial phase, you can also work on a existance directory. Always create a feature branch and work on that. The scenerio is given below: In main repo there is a file1 and into the file the text is:

```
1 Hello ,
2 My Name is Git.
```

And you fork the repo and clone it your local machine and make a branch named **feature** edit the file like:

```
1 Hello ,
2 My Name is Git.
3 And You are HTML
```

So you add it and commit it. On the same time someone changed the main repo file1 like this:

```
1 Hello ,
2 My Name is Git.
3 And You are HTML
4 And you love javascript
```

Now if you want to push the file to remote fork as well as merge to the main. you will see some conflict issue. There is the procedure to resolve the conflict manually. **Fetch upstream**

```
1 git fetch upstream
```

Checkout feature branch

```
1 git checkout feature
```

rebase your branch

```
1 git rebase upstream/master
```

You will see error like it:

```
1 CONFLICT (content): Merge conflict in file1
```

Now open file1 and manually changed it:

```
1 #from this
2 <<<<<< HEAD
3 Hello,
4 My Name is Git.
5 And You are HTML
6 -----
7 -----
8 And you love javascript
9 >>>>> feature
10
11 #to that
12 Hello,
13 My Name is Git.
14 And You are HTML
15 And you love javascript #or however you want to do it
```

Now make conflict resolved and continue rebase .

```
1 git add file1
2 git rebase --continue
```

Now You can push it

```
1 git push origin feature -f
```

Now from your fork feature branch you can add merge request and then delete them. And after doing these you should again do these things for make your local, fork and main repos commit history similar so that if you want to do something there will be no problem occurs. Also do these things everytime before you work. All the procedure here.

```
1 git checkout feature
2 git fetch upstream
3 git fetch upstream/master
4 git checkout master
5 git push
```

```
6 git branch -d feature
7 git push origin --delete feature
```