# Install GIT

Git is a distributed versions control system, that log every working history of a repository (storage).

Install git on the terminal run:

* sudo apt-get update
* sudo apt-get install git

After we finish installing git, check if the installation is working:

* git --version

Output:

git version x.x.x, that means git version that installed in local is x.x.x.

After that now we need to configure git username and email. On terminal run:

* git config --global user.name <full\_name>
* git config --global user.email <email\_address>

For example:

* git config --global user.name "Jack Runter"
* git config --global user.email jack@gmail.com

To check configuration, on terminal run:

* git config --list

Output:

    user.name=jack runter

    user.email=jack@gmail.com

\* Configuration will be saved in *~/.gitconfig* file*.*

# Local Implementation

after successfully install and configuring git now we will be using git locally, let's create an empty folder in home directory:

* mkdir ~/learn\_git\_local

Now change directory to that directory:

* cd learn\_git\_local

After that, now we need to initialize 'learn\_git\_local' as our local repository, on terminal run:

* git init

Output:

Initialized empty Git repository in /home/username/learn\_git\_local/.git/

then check current state repository, run:

* git status

Output:

    On branch master

    Initial commit

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

it means repository is in branch '*master*', and nothing to be commit (save our working to local repository), now add a new file:

* touch learn\_git.txt

then check current state repository, run:

* git status

Output:

    On branch master

    Initial commit

    Untracked files:

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

    learn\_git.txt

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

That means git detect new file that need to add to our repository, git will give suggestion for what action that we going to do, from that status git suggest for add new file learn\_git.txt, run:

* git add learn\_git.txt

Then if we check status repository, run:

* git status

That will result:

    On branch master

    Initial commit

    Changes to be committed:

    (use "git rm --cached <file>..." to unstage)

    new file: learn\_git.txt

That means file learn\_git.txt has been added to index of working tree and prepare for next commit, so every change from this file will be logged. Now try add content to learn\_git.txt, run:

* gedit learn\_git.txt

Add this text:

    Learn git

    in local repository

Save the file and close the gedit. Now if we check:

* git status

That will result:

    On branch master

    Initial commit

    Changes to be committed:

    (use "git rm --cached <file>..." to unstage)

    new file: learn\_git.txt

    Changes not staged for commit:

    (use "git add <file>..." to update what will be committed)

    (use "git checkout -- <file>..." to discard changes in working directory)

    modified: learn\_git.txt

That means file *learn\_git.txt* has changes, if we notice git will give suggestion for git add or git checkout, if we use git checkout *learn\_git.txt*, that will remove all changes back to first add that file or when the last commit. To see what changes has been added to the file use:

* git diff learn\_git.txt

Output:

    diff --git a/learn\_git.txt b/learn\_git.txt

    index e69de29..e16794a 100644

    --- a/learn\_git.txt

    +++ b/learn\_git.txt

    @@ -0,0 +1,2 @@

    +Learn git

    +in local repository

If we see @@ -0,0 +1,2 @@, it means in the file has add from line 1 until line 2. And +Learn git +in local repository, that means the file has an added text “Learn git in local repository”. After that use git add, run:

* git add learn\_git.txt

Then if we check:

* git status

Output:

    On branch master

    Initial commit

    Changes to be committed:

    (use "git rm --cached <file>..." to unstage)

    new file: learn\_git.txt

now we need commit our changes, run:

* git commit learn\_git.txt -m “First Commit”

That command means that we commit our changes to the local repository with a message “First Commit”. Now if we check:

* git status

Output:

    On branch master

    nothing to commit, working directory clean

It means in branch master nothing changes. Now we want see logs of commit for the last created:

* git log -1

Output:

    commit 785186a77cecace02b9cd413444dd6084972b6e0

    Author: jack ranter <jack@kiranatama.com>

    Date: Thu Feb 18 14:13:36 2017 +0700

After successfully commit for the first time, let’s say we want work on new branch so the base project will not change. Let's create new branch using:

* git branch development\_1

\* if we want delete branch use: git branch -d development\_1

Check the all branch that exist for repository, run:

* git branch

Output:

    development\_1

    \* master

the mark '\*' means the repository is in branch master, if we want move to branch '*development\_1'*, run:

* git checkout development\_1

Output:

Switched to branch 'development\_1'

Now if we check the current branch:

* git branch

Output:

    \* development\_1

    master

That means the repository has move to branch '*development\_1'*.

\* *git checkout,* can be used to remove changes in working file or switching between branches.

## Undoing Changes

### git checkout

The git checkout command serves three distinct functions: checking out files, checking out commits, and checking out branches. In this module, we’re only concerned with the first two configurations.

Checking out a commit makes the entire working directory match that commit. This can be used to view an old state of your project without altering your current state in any way. Checking out a file lets you see an old version of that particular file, leaving the rest of your working directory untouched.

#### Usage

git checkout master

Return to the master branch. Branches are covered in depth in the next module, but for now, you can just think of this as a way to get back to the “current” state of the project.

git checkout <commit> <file>

Check out a previous version of a file. This turns the <file> that resides in the working directory into an exact copy of the one from <commit> and adds it to the staging area.

git checkout <commit>

Update all files in the working directory to match the specified commit. You can use either a commit hash or a tag as the <commit> argument. This will put you in a detached HEAD state.

#### Example

**Viewing an Old Revision**

This example assumes that you’ve started developing a crazy experiment, but you’re not sure if you want to keep it or not. To help you decide, you want to take a look at the state of the project before you started your experiment. First, you’ll need to find the ID of the revision you want to see.

git log --oneline

Let’s say your project history looks something like the following:

b7119f2 Continue doing crazy things

872fa7e Try something crazy

a1e8fb5 Make some important changes to hello.py

435b61d Create hello.py

9773e52 Initial import

You can use git checkout to view the “Make some import changes to hello.py” commit as follows:

git checkout a1e8fb5

This makes your working directory match the exact state of the a1e8fb5 commit. You can look at files, compile the project, run tests, and even edit files without worrying about losing the current state of the project. *Nothing* you do in here will be saved in your repository. To continue developing, you need to get back to the “current” state of your project:

git checkout master

This assumes that you're developing on the default master branch, which will be thoroughly discussed in the Branches Module.

Once you’re back in the master branch, you can use either git revert or git reset to undo any undesired changes.

**Checking Out a File**

If you’re only interested in a single file, you can also use git checkout to fetch an old version of it. For example, if you only wanted to see the hello.py file from the old commit, you could use the following command:

git checkout a1e8fb5 hello.py

Remember, unlike checking out a commit, this *does* affect the current state of your project. The old file revision will show up as a “Change to be committed,” giving you the opportunity to revert back to the previous version of the file. If you decide you don’t want to keep the old version, you can check out the most recent version with the following:

git checkout HEAD hello.py

### git revert

The git revert command undoes a committed snapshot. But, instead of removing the commit from the project history, it figures out how to undo the changes introduced by the commit and appends a *new* commit with the resulting content. This prevents Git from losing history, which is important for the integrity of your revision history and for reliable collaboration.

#### Usage

git revert <commit>

Generate a new commit that undoes all of the changes introduced in <commit>, then apply it to the current branch.

#### Example

The following example is a simple demonstration of git revert. It commits a snapshot, then immediately undoes it with a revert.

# Edit some tracked files

# Commit a snapshot

git commit -m "Make some changes that will be undone"

# Revert the commit we just created

git revert HEAD

### git reset

If git revert is a “safe” way to undo changes, you can think of git reset as the *dangerous* method. When you undo with git reset(and the commits are no longer referenced by any ref or the reflog), there is no way to retrieve the original copy—it is a *permanent* undo. Care must be taken when using this tool, as it’s one of the only Git commands that has the potential to lose your work.

Like [git checkout](https://www.atlassian.com/git/tutorials/undoing-changes/git-checkout), git reset is a versatile command with many configurations. It can be used to remove committed snapshots, although it’s more often used to undo changes in the staging area and the working directory. In either case, it should only be used to undo *local* changes—you should never reset snapshots that have been shared with other developers.

#### Usage

git reset <file>

Remove the specified file from the staging area, but leave the working directory unchanged. This unstages a file without overwriting any changes.

git reset

Reset the staging area to match the most recent commit, but leave the working directory unchanged. This unstages *all* files without overwriting any changes, giving you the opportunity to re-build the staged snapshot from scratch.

git reset --hard

Reset the staging area and the working directory to match the most recent commit. In addition to unstaging changes, the --hardflag tells Git to overwrite all changes in the working directory, too. Put another way: this *obliterates* all uncommitted changes, so make sure you really want to throw away your local developments before using it.

git reset <commit>

Move the current branch tip backward to <commit>, reset the staging area to match, but leave the working directory alone. All changes made since <commit> will reside in the working directory, which lets you re-commit the project history using cleaner, more atomic snapshots.

git reset --hard <commit>

Move the current branch tip backward to <commit> and reset both the staging area and the working directory to match. This obliterates not only the uncommitted changes, but all commits after <commit>, as well.

#### Examples

**Unstaging a File**

The git reset command is frequently encountered while preparing the staged snapshot. The next example assumes you have two files called hello.py and main.py that you’ve already added to the repository.

# Edit both hello.py and main.py

# Stage everything in the current directory

git add .

# Realize that the changes in hello.py and main.py

# should be committed in different snapshots

# Unstage main.py

git reset main.py

# Commit only hello.py

git commit -m "Make some changes to hello.py"

# Commit main.py in a separate snapshot

git add main.py

git commit -m "Edit main.py"

As you can see, git reset helps you keep your commits highly-focused by letting you unstage changes that aren’t related to the next commit.

**Removing Local Commits**

The next example shows a more advanced use case. It demonstrates what happens when you’ve been working on a new experiment for a while, but decide to completely throw it away after committing a few snapshots.

# Create a new file called `foo.py` and add some code to it

# Commit it to the project history

git add foo.py

git commit -m "Start developing a crazy feature"

# Edit `foo.py` again and change some other tracked files, too

# Commit another snapshot

git commit -a -m "Continue my crazy feature"

# Decide to scrap the feature and remove the associated commits

git reset --hard HEAD~2

The git reset HEAD~2 command moves the current branch backward by two commits, effectively removing the two snapshots we just created from the project history. Remember that this kind of reset should only be used on *unpublished* commits. Never perform the above operation if you’ve already pushed your commits to a shared repository.

### git clean

The git clean command removes untracked files from your working directory. This is really more of a convenience command, since it’s trivial to see which files are untracked with git statusand remove them manually. Like an ordinary rm command, git clean is *not* undoable, so make sure you really want to delete the untracked files before you run it.

The git clean command is often executed in conjunction with git reset --hard. Remember that resetting only affects tracked files, so a separate command is required for cleaning up untracked ones. Combined, these two commands let you return the working directory to the exact state of a particular commit.

#### Usage

git clean -n

Perform a “dry run” of git clean. This will show you which files are going to be removed without actually doing it.

git clean -f

Remove untracked files from the current directory. The -f (force) flag is required unless the clean.requireForce configuration option is set to false (it's true by default). This will *not* remove untracked folders or files specified by .gitignore.

git clean -f <path>

Remove untracked files, but limit the operation to the specified path.

git clean -df

Remove untracked files *and* untracked directories from the current directory.

git clean -xf

Remove untracked files from the current directory as well as any files that Git usually ignores.

#### Example

The following example obliterates all changes in the working directory, including new files that have been added. It assumes you’ve already committed a few snapshots and are experimenting with some new developments.

# Edit some existing files

# Add some new files

# Realize you have no idea what you're doing

# Undo changes in tracked files

git reset --hard

# Remove untracked files

git clean -df

After running this reset/clean sequence, the working directory and the staging area will look exactly like the most recent commit, and git status will report a clean working directory. You're now ready to begin again.

Note that, unlike the second example in git reset, the new files were \_not \_added to the repository. As a result, they could not be affected by git reset --hard, and git clean was required to delete them.

## Stashing

Often, when you’ve been working on part of your project, things are in a messy state and you want to switch branches for a bit to work on something else. The problem is, you don’t want to do a commit of half-done work just so you can get back to this point later. The answer to this issue is the git stash command.

Stashing takes the dirty state of your working directory — that is, your modified tracked files and staged changes — and saves it on a stack of unfinished changes that you can reapply at any time.

### Stashing Your Work

To demonstrate, you’ll go into your project and start working on a couple of files and possibly stage one of the changes. If you run git status, you can see your dirty state:

$ git status

# On branch master

# Changes to be committed:

# (use "git reset HEAD <file>..." to unstage)

#

# modified: index.html

#

# Changes not staged for commit:

# (use "git add <file>..." to update what will be committed)

#

# modified: lib/simplegit.rb

#

Now you want to switch branches, but you don’t want to commit what you’ve been working on yet; so you’ll stash the changes. To push a new stash onto your stack, run git stash:

$ git stash

Saved working directory and index state \

"WIP on master: 049d078 added the index file"

HEAD is now at 049d078 added the index file

(To restore them type "git stash apply")

Your working directory is clean:

$ git status

# On branch master

nothing to commit, working directory clean

At this point, you can easily switch branches and do work elsewhere; your changes are stored on your stack. To see which stashes you’ve stored, you can use git stash list:

$ git stash list

stash@{0}: WIP on master: 049d078 added the index file

stash@{1}: WIP on master: c264051 Revert "added file\_size"

stash@{2}: WIP on master: 21d80a5 added number to log

In this case, two stashes were done previously, so you have access to three different stashed works. You can reapply the one you just stashed by using the command shown in the help output of the original stash command: git stash apply. If you want to apply one of the older stashes, you can specify it by naming it, like this: git stash apply stash@{2}. If you don’t specify a stash, Git assumes the most recent stash and tries to apply it:

$ git stash apply

# On branch master

# Changes not staged for commit:

# (use "git add <file>..." to update what will be committed)

#

# modified: index.html

# modified: lib/simplegit.rb

#

You can see that Git re-modifies the files you uncommitted when you saved the stash. In this case, you had a clean working directory when you tried to apply the stash, and you tried to apply it on the same branch you saved it from; but having a clean working directory and applying it on the same branch aren’t necessary to successfully apply a stash. You can save a stash on one branch, switch to another branch later, and try to reapply the changes. You can also have modified and uncommitted files in your working directory when you apply a stash — Git gives you merge conflicts if anything no longer applies cleanly.

The changes to your files were reapplied, but the file you staged before wasn’t restaged. To do that, you must run the git stash apply command with a --index option to tell the command to try to reapply the staged changes. If you had run that instead, you’d have gotten back to your original position:

$ git stash apply --index

# On branch master

# Changes to be committed:

# (use "git reset HEAD <file>..." to unstage)

#

# modified: index.html

#

# Changes not staged for commit:

# (use "git add <file>..." to update what will be committed)

#

# modified: lib/simplegit.rb

#

The apply option only tries to apply the stashed work — you continue to have it on your stack. To remove it, you can run git stash drop with the name of the stash to remove:

$ git stash list

stash@{0}: WIP on master: 049d078 added the index file

stash@{1}: WIP on master: c264051 Revert "added file\_size"

stash@{2}: WIP on master: 21d80a5 added number to log

$ git stash drop stash@{0}

Dropped stash@{0} (364e91f3f268f0900bc3ee613f9f733e82aaed43)

You can also run git stash pop to apply the stash and then immediately drop it from your stack.

### Un-applying a Stash

In some use case scenarios, you might want to apply stashed changes, do some work, but then un-apply those changes that originally came from the stash. Git does not provide such a stash unapply command, but it is possible to achieve the effect by simply retrieving the patch associated with a stash and applying it in reverse:

$ git stash show -p stash@{0} | git apply -R

Again, if you don’t specify a stash, Git assumes the most recent stash:

$ git stash show -p | git apply -R

You may want to create an alias and effectively add a stash-unapply command to your Git. For example:

$ git config --global alias.stash-unapply '!git stash show -p | git apply -R'

$ git stash apply

$ #... work work work

$ git stash-unapply

### Creating a Branch from a Stash

If you stash some work, leave it there for a while, and continue on the branch from which you stashed the work, you may have a problem reapplying the work. If the apply tries to modify a file that you’ve since modified, you’ll get a merge conflict and will have to try to resolve it. If you want an easier way to test the stashed changes again, you can run git stash branch, which creates a new branch for you, checks out the commit you were on when you stashed your work, reapplies your work there, and then drops the stash if it applies successfully:

$ git stash branch testchanges

Switched to a new branch "testchanges"

# On branch testchanges

# Changes to be committed:

# (use "git reset HEAD <file>..." to unstage)

#

# modified: index.html

#

# Changes not staged for commit:

# (use "git add <file>..." to update what will be committed)

#

# modified: lib/simplegit.rb

#

Dropped refs/stash@{0} (f0dfc4d5dc332d1cee34a634182e168c4efc3359)

This is a nice shortcut to recover stashed work easily and work on it in a new branch.

# Remote Server Implementation

There many remote servers for git in market like github.com, bitbucket.org. The difference between those is github cannot create private repository for free account, that means github focus on open source project, whereas bitbucket allow create private repository for free account, that means bitbucket it more focus for enterprise.

## Generate SSH key

SSH is a cryptographic network protocol for secure data communication, remote command-line login, remote command execution, and other secure network services between two networked computers.

To create new ssh key in ubuntu, on terminal run:

* ssh-keygen -t rsa -C <wer@email.address>

For example:

* cd ~
* ssh-keygen -t rsa -C "someone@gmail.com"

After we run that command, we will be asked for file destination, and passphrase.

For destination leave the default value, where it will be put in ~/.ssh folder

passphrase leave it empty.

After finish, it will generate:

    - .ssh/

    -- id\_rsa

    -- id\_rsa.pub

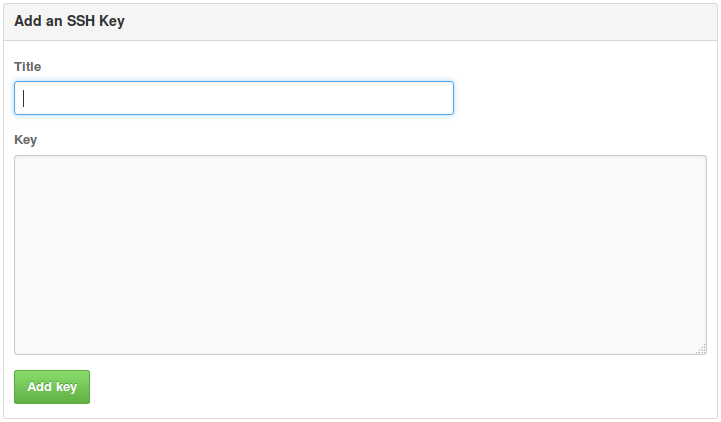
    -- known\_hosts

'id\_rsa.pub' file contain public key, that we will use for communication between remote server and local.

## Create Github account

Here we will use github as remote repository, first create account in: [*https://github.com/join*](https://github.com/join)

After success creating new account, now we need to add SSH Key to github, so communication between local repository and remote server repository in github can be established, go to: *https://github.com/settings/ssh*, click button “Add SSH key”, that will show input form:



* In input Title, insert email address.
* In input Key, insert public key from '.ssh/id\_rsa.pub'. For example on terminal run:
* cat ~/.ssh/id\_rsa.pub

Output:

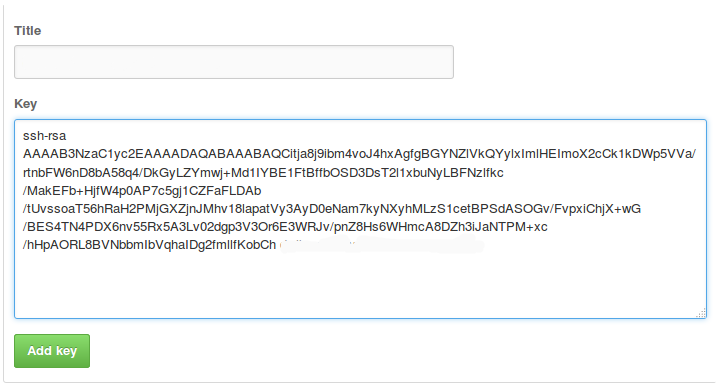
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQCitja8j9ibm4voJ4hxAgfgBGYNZlVkQYylxImlHEImoX2cCk1

kDWp5VVa/rtnbFW6nD8bA58q4/DkGyLZYmwj+Md1IYBE1FtBffbOSD3DsT2l1xbuNyLBFNzlfkc/MakEFb

+HjfW4p0AP7c5gj1CZFaFLDAb/tUvssoaT56hRaH2PMjGXZjnJMhv18lapatVy3AyD0eNam7kyNXyhMLzS1

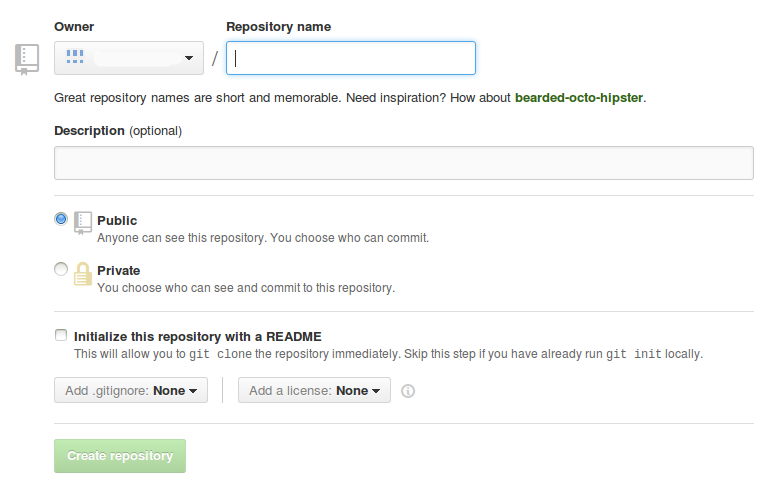
cetBPSdASOGv/FvpxiChjX+wG/BES4TN4PDX6nv55Rx5A3Lv02dgp3V3Or6E3WRJv/pnZ8Hs6WHmcA8D Zh3iJaNTPM+xc/hHpAORL8BVNbbmIbVqhaIDg2fmllfKobCh jack@gmail.com

Copy that public key to insert field, so it will look like:

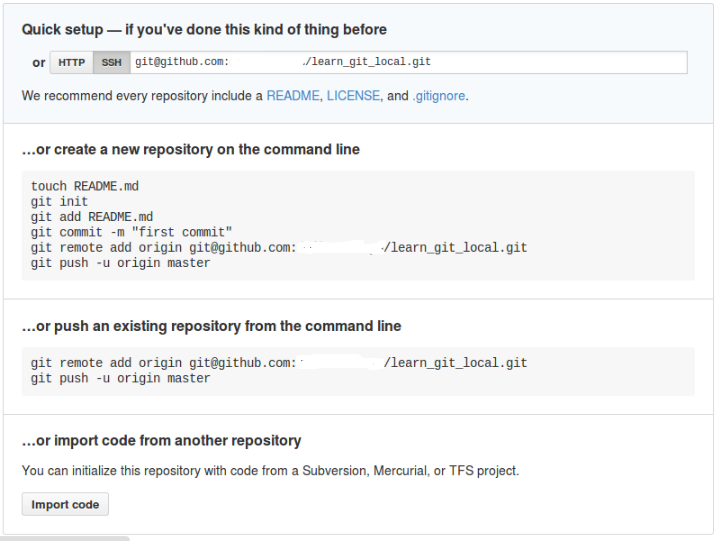


Click button “Add key”.

Now create new repository in: *https://github.com/new*



Fill Repository Name the same as Local Repository name '*learn\_git\_local*', choose public so everyone can see. After that click button '*Create repository*', and then we will be redirected to:



The description above is how to prepare local repository, so we can sent all commit in local into remote server github. This is example to setup in local, open console:

* cd learn\_git\_local

After that run:

* git remote add origin git@github.com:jack/learn\_git\_local.git

\*If we want modify remote repository use git remote set-url origin <link\_to\_remote>

example: git remote set-url origin https://jack@github.com/jack/learn\_git\_local.git

That git command is for setup remote repository to github, if we want see detail can be found in file '*learn\_git\_local/.git/config*'. 'Origin' is default name repository github.

## Implement Github

After successfully configuring, now we need to push (send local repository to github) the files. On terminal, and make sure we’re in '*learn\_git\_local*' directory, run:

* git push -u origin master

If we setup passphrase when generate ssh key, it will be asking passphrase.

If the process finished successfully, it will show message:

    Counting objects: 5, done.

    Writing objects: 100% (3/3), 303 bytes | 0 bytes/s, done.

    Total 3 (delta 0), reused 0 (delta 0)

    To git@github.com:jack/learn\_git\_local.git

    785186a..29e0386 master -> master

now if we access *https://github.com/user\_name/learn\_git\_local/commits/master,*we can see lists of commited files in our local that logs date, author, then the detail changes.

Next if we want to push to github again use:

* git push

\*if '*git push*' not work use this: '*git push origin master*'

That is because we use '-u' option the first time we pushed to github (this command allow us to not include origin and master).

After successfully push to github, if other person modified our work and push it to github, that means our local work need to be updated with changes from the remote repository, inside '*learn\_git\_local*' run:

* git pull origin master

The 'git pull' will successfully run when our local repository is the same as the remote one, if not git will give suggestion what we must do before pull.  
Make sure every commit that has push to remote server is never rollback, because it will result bad behavior, so make sure our work tested or create new branch for backup before pushing to remote server.   
  
Make sure we’re not modifying the same file with other members, because it will cause conflict (redundancy in repository that will cause confusion to git engine to choose which one the valid changes), to prevent this we must pull first before push to git server, but if conflict still happens we must manually modify the changes and choose the updated or both changes that we will commit.

## Gitignore

Gitignore is a file where we list of all file and folder that won’t be commited, this file resides in root project folder or root repository folder. This is an example where to put the file gitignore:

    -- latihan /

    ---- app/

    ---- db/

    ---- config/

    ---- .git/

    ---- .gitignore

    ---- tmp/

    ---- public/

    ---- log/

for the case above we won’t commit all the folders listed. Now let’s try to add some changes to .gitignore, and then add:

#ignore this file when git commit

.version

tmp/

public/

log/

config/

Save that file and when we check with git status command, the file or folder that we added in gitignore will not be detected as modification.

For complete resources:[*http://git-scm.com/docs*]( http:/git-scm.com/docs)

## Changing a remote's URL

The git remote set-url command changes an existing remote repository URL

The git remote set-url command takes two arguments:

* An existing remote name. For example, origin or upstream are two common choices.
* A new URL for the remote. For example:
  + If you're updating to use HTTPS, your URL might look like:

https://[hostname]/USERNAME/REPOSITORY.git

* + If you're updating to use SSH, your URL might look like:

git@hostname:USERNAME/REPOSITORY.git

### Example

List your existing remotes in order to get the name of the remote you want to change.

$ git remote -v

origin git@hostname:USERNAME/REPOSITORY.git (fetch)

origin git@hostname:USERNAME/REPOSITORY.git (push)

Change your remote's URL from SSH to HTTPS with the git remote set-url command.

$git remote set-url origin https://hostname/USERNAME/REPOSITORY.git

Verify that the remote URL has changed.

$ git remote -v

# Verify new remote URL

origin https://hostname/USERNAME/REPOSITORY.git (fetch)

origin https://hostname/USERNAME/REPOSITORY.git (push)