

# FUNDAMENTALS OF GIT

Git is a version control system for monitoring changes in computer files and for coordinating the work of multiple people on those files. It is primarily used for versioning code in software development, but can be used to track changes to any files. Git help us to:

- create remote and local repositories
- manage different versions of the same artifact
- manage changes in artifact
- see comparison of the different versions
- make collaboration between developers
- make requests for new updates and changes of specific artifact
- move quickly to many different stages of the same process, in case we mess up something
- quickly remove specific branch or feature or restart checkpoint of our process

## Download for Linux and Unix

For Ubuntu/Mint, this PPA provides the latest stable upstream Git version(execute commands inside of terminal):

```
a413@masterNode ~ $ sudo add-apt-repository ppa:git-core/ppa
a413@masterNode ~ $ sudo apt-get update && sudo apt install git
a413@masterNode ~ $ git help
```

These are common Git commands used in various situations:

### start a working area (see also: git help tutorial)

clone	Clone a repository into a new directory
init	Create an empty Git repository or reinitialize an existing one

### work on the current change (see also: git help everyday)

add	Add file contents to the index
mv	Move or rename a file, a directory, or a symlink
reset	Reset current HEAD to the specified state
rm	Remove files from the working tree and from the index

### examine the history and state (see also: git help revisions)

bisect	Use binary search to find the commit that introduced a bug
grep	Print lines matching a pattern
log	Show commit logs
show	Show various types of objects
status	Show the working tree status

### grow, mark and tweak your common history

branch	List, create, or delete branches
checkout	Switch branches or restore working tree files
commit	Record changes to the repository
diff	Show changes between commits, commit and working tree, etc
merge	Join two or more development histories together
rebase	Forward-port local commits to the updated upstream head
tag	Create, list, delete or verify a tag object signed with GPG

### collaborate (see also: git help workflows)

fetch	Download objects and refs from another repository
pull	Fetch from and integrate with another repository or a local branch
push	Update remote refs along with associated objects

'git help -a' and 'git help -g' list available subcommands and some concept guides.

Now we will create "gitLib" directory in our home path:

```
a413@mN:~$ mkdir gitLib && cd gitLib
```

Then we will configure our git user. First we will check our config global list:

```
a413@mN:~/gitLib$ git config --global --list
fatal: unable to read config file '/home/a413/.gitconfig': No such file or directory
```

System show us that there is not config file for git. We will create him manually.

```
a413@mN:~/gitLib$ touch ~/.gitconfig
```

After we create config file, we will add git username and email. In my case that would be "a413" user and "nikola\_djurdjevic@hotmail.co.uk" email. You add whatever you like:

```
a413@mN:~/gitLib$ git config --global user.name "a413"
a413@mN:~/gitLib$ git config --global user.email "nikola_djurdjevic@hotmail.co.uk"
```

How we can check if everything is fine? See content of ~/.gitconfig file. Here will be our last modification.

```
sudo nano ~/.gitconfig
```

## INIT GIT REPOSITORY

GIT INIT is command with which we define that our specific directory become GIT repository.

All we need to do is to create new empty directory and inside of him execute **GIT INIT** command.

```
a413@mN:~/gitLib$ mkdir first_repo && cd first_repo
```

```
a413@mN:~/gitLib/first_repo$ git init
```

Initialized empty Git repository in /home/a413/gitLib/first\_repo/.git/

```
a413@mN:~/gitLib/first_repo$ ls -la
```

```
drwxr-xr-x 3 a413 a413 4096 Oct 28 16:09 .
```

```
drwxr-xr-x 3 a413 a413 4096 Oct 28 16:08 ..
```

```
drwxr-xr-x 7 a413 a413 4096 Oct 28 16:09 .git
```

>> We created .git!

Checkpoints of the GIT cycle are:

- initialize git repository
- check status of git repository
- add new changes to stage area
- commit changes from stage area to new checkpoint
- check git status and log
- if we need push that checkpoint to remote git repository (GITHUB)
- then we can delete our local repository (directory on our computer)
- pull from remote repository to local repository, to see our last change
- for new modification, create new branch, switch to that branch, and start modification

In next sector we will explain every step and every command that has been involved in cycle.

## COMMITTING CHANGES IN GIT

Firstly we will check our git status for our new empty git repository.

```
a413@mN:~/gitLib/first_repo$ git status
On branch master
No commits yet
nothing to commit (create/copy files and use "git add" to track)
```

As we can see there is no changes in our git repository, and there is nothing that could be committed. Now, we will create new file. And then execute again git status.

```
a413@mN:~/gitLib/first_repo$ touch message1.txt
a413@mN:~/gitLib/first_repo$ git status
On branch master
No commits yet
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    message1.txt
nothing added to commit but untracked files present (use "git add" to track)
```

Now, we can't commit anything, but there has been change in our git repository. Still, this change is red. [Why?](#) Because we have change, but we didn't add that change on stage. How we can do that?

We will do that with git add. We can add all files with "[git add .](#)" or specific file with "[git add file\\_name](#)". In this case we will execute:

```
a413@mN:~/gitLib/first_repo$ git add message1.txt
```

Then we will again check git status:

```
a413@mN:~/gitLib/first_repo$ git status
On branch master - No commits yet
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   message1.txt
```

Now our status is green, and we can commit this change because it's on stage area. If we want to remove from stage area, we will execute:

```
a413@mN:~/gitLib/first_repo$ git rm --cached message1.txt
rm 'message1.txt'
```

If we check again with git status, we will see that file isn't anymore on stage area.

## COMMIT

Commit help us to identify and understand what we done, in that specific moment. Look on commit like checkpoint. We added with "git add message1.txt" our new file to stage. Now we want to commit that as checkpoint.

```
a413@mN:~/gitLib/first_repo$ git commit -m "Our first system message!"
[master (root-commit) 0ef0242] Our first system message!
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 message1.txt
a413@mN:~/gitLib/first_repo$ git status
On branch master
nothing to commit, working tree clean
```

Statuses of changes:

```
-modified    > Here we make modification to repository artifacts
-staged      > Here we add the changes to staging area
-committed   > Here we commit the changes to Git Repository

-command "git status"    > Tells us the status of the working directory and staging area
-command "git log"       > Displays committed snapshots or commit history

-git status --long > Default behavior!
-git status -s      > List of Untracked files or files ready for commit! ("A" or "??")
```

If we have "M" in function > git status -s > That means that that file has been modified!

If "M" is red, that means that our file, isn't on the stage area!

If we add him, with > git add fileName.txt > Now "M" will be green, because it is in stage area!

If we rename file, in git repository, with "git status -s" now we will get character "D", our file has been deleted!

With git commit --help command we can see all possible incomes what we can do while we commit.

With git status --help command we can see what kind of status we can get from git repository.

With git log --help command we can see what kind of logs we can get from git repository.

Execute these command in Linux Terminal.

## COMMIT LOG

We use log to remind us, what we have done, in specific commit, in case we won't to switch to some other checkpoint, or if we want to remove something.

```
a413@mN:~/gitLib/first_repo$ git log
commit 0ef02429e767d5d02526862cee4bf670f72c8072 (HEAD -> master)
Author: a413 <nikola_djurdjevic@hotmail.co.uk>
Date: Mon Oct 28 17:41:13 2019 +0100
```

Our first system message!

We also can have shorter version of log:

```
a413@mN:~/gitLib/first_repo$ git log --oneline
0ef0242 (HEAD -> master) Our first system message!
```

We can have logs between two ID Commits, like this:

```
git log <since>..<until>
```

Two value for this functions are IDs of commits, in “git log --oneline” mode!

**Example:** git log d110129..74d91c5 --oneline

We can write how many last commits we want to see:

```
git log -n 3 --oneline >> Last three commits!
```

## BRANCH

Branch is like one line of process in our development. We can divide the branches according to purpose. Cycle of branch is to:

- create new branch
- commit changes to new branch
- test changes
- if everything is fine, merge side branch with master branch

We will have many different branches for many different purposes. Every jump from one branch to another is merge. That means that changes of first branch become state of second branch. So for example is everything is fine with our feature, our feature become master branch. First we will see the list of our branches:

```
a413@mN:~/gitLib/first_repo$ git branch
* master
```

Now we will create new branch in our git repository:

```
a413@mN:~/gitLib/first_repo$ git branch feature/task2
```

If we look again on our list we will see:

```
a413@mN:~/gitLib/first_repo$ git branch
feature/task2
* master
```

Great, but how we switch to another branch? With command **checkout**.

```
a413@mN:~/gitLib/first_repo$ git checkout feature/task2
Switched to branch 'feature/task2'
```

```
a413@mN:~/gitLib/first_repo$ git branch
* feature/task2
master
```

Now, we will modified **message1.txt** file, and add new line into file. After that we will check status, then add to stage area changes, and then commit changes from stage area. Then we will explore **git log** to see what we done.

```
a413@mN:~/gitLib/first_repo$ echo "This is the first system message!" >> message1.txt
```

```
a413@mN:~/gitLib/first_repo$ git status
```

On branch feature/task2

Changes not staged for commit:

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

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

**modified: message1.txt**

no changes added to commit (use "git add" and/or "git commit -a")

```
a413@mN:~/gitLib/first_repo$ git add .
```

```
a413@mN:~/gitLib/first_repo$ git status
```

On branch feature/task2

Changes to be committed:

(use "git restore --staged <file>..." to unstage)

**modified: message1.txt**

```
a413@mN:~/gitLib/first_repo$ git commit -m "First system message has been modified!"
```

[feature/task2 650bf2c] First system message has been modified!

1 file changed, 1 insertion(+)

```
a413@mN:~/gitLib/first_repo$ git status
```

On branch feature/task2

nothing to commit, working tree clean

```
a413@mN:~/gitLib/first_repo$ git log --oneline
```

**650bf2c (HEAD -> feature/task2) First system message has been modified!**

0ef0242 (master) Our first system message!

=====

To rename branch use:

```
git branch -m small-feature quick-feature
```

Create new branch based on other branch use checkout with -b flag and two parameters:

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

## MERGE BRANCH

From last log we have seen this message:

```
a413@mN:~/gitLib/first_repo$ git log --oneline
650bf2c (HEAD -> feature/task2) First system message has been modified!
0ef0242 (master) Our first system message!
```

Here, we are still on feature/task2 branch and that is our HEAD checkpoint. Now we want to merge that with master.

```
a413@mN:~/gitLib/first_repo$ git merge feature/task2
Already up to date.
```

We get the message that everything is already updated because we are still on feature/task2 branch. We must switch to master branch, and from there ask for merge request. Before that we can see the difference with git diff.

```
a413@mN:~/gitLib/first_repo$ git checkout master
Switched to branch 'master'

a413@mN:~/gitLib/first_repo$ git diff master feature/task2
diff --git a/message1.txt b/message1.txt
index e69de29..68d6809 100644
--- a/message1.txt
+++ b/message1.txt
@@ -0,0 +1 @@
+This is the first system message!
```

Now we execute merge with feature/task2 branch:

```
a413@mN:~/gitLib/first_repo$ git merge feature/task2
Updating 0ef0242..650bf2c
Fast-forward
 message1.txt | 1 +
 1 file changed, 1 insertion(+)
```

If we see log, we will get that our HEAD is now master, but in same time branch feature/task2 is equal to master.

```
a413@mN:~/gitLib/first_repo$ git log --oneline --decorate --graph
* 650bf2c (HEAD -> master, feature/task2) First system message has been modified!
* 0ef0242 Our first system message!
```

Can we remove that feature/task2 branch if we don't need it anymore? Yes, we can.

```
a413@mN:~/gitLib/first_repo$ git branch -d feature/task2
Deleted branch feature/task2 (was 650bf2c).
```

Now, when we execute git log, we won't see feature/task2 branch:

```
a413@mN:~/gitLib/first_repo$ git log --oneline --decorate --graph
* 650bf2c (HEAD -> master) First system message has been modified!
* 0ef0242 Our first system message!
```

A git <branch> represents an independent line of development. We create a branch when we want to create a new feature or do a bug fix. Unstable code is never committed to main or production code base. A new branch get it's own working directory, index or staging area and commit history.

## COMMIT WITH CHANGES – CHECKOUT SPECIFIC COMMIT

Now we will add 3 more changes to our file, and every change will be defined by another commit.

```
a413@mN:~/gitLib/first_repo$ echo "Second system message:Welcome!" >> message1.txt
```

```
a413@mN:~/gitLib/first_repo$ git status
```

On branch master

Changes not staged for commit:

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

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

modified: message1.txt

no changes added to commit (use "git add" and/or "git commit -a")

```
a413@mN:~/gitLib/first_repo$ git commit -m "Second system message has been added!"
```

On branch master

Changes not staged for commit:

modified: message1.txt

no changes added to commit

Here, we can't commit because we didn't put our change on stage area, so before git commit we will add on stage:

```
a413@mN:~/gitLib/first_repo$ git add .
```

```
a413@mN:~/gitLib/first_repo$ git commit -m "Second system message has been added!"
```

[master aba3bd9] Second system message has been added!

1 file changed, 1 insertion(+)

And we will do the same thing two more times, with third and fourth system message:

```
a413@mN:~/gitLib/first_repo$ echo "Third system message:Analyze!" >> message1.txt
```

```
a413@mN:~/gitLib/first_repo$ git add .
```

```
a413@mN:~/gitLib/first_repo$ git commit -m "Third system message has been added!"
```

[master 24af2fa] Third system message has been added!

1 file changed, 1 insertion(+)

```
a413@mN:~/gitLib/first_repo$ echo "Fourth system message:Proceed!" >> message1.txt
```

```
a413@mN:~/gitLib/first_repo$ git commit -am "Fourth system message has been added!"
```

[master e0c20e6] Fourth system message has been added!

1 file changed, 1 insertion(+)

At last message we didn't execute **git add .** end yet we didn't get an error while we execute git commit. **How?**

Because in git commit command our attribute was **-am**, which means that we will in the same time, add to stage area our changes, and after that immediately execute commit for that change.

```
a413@mN:~/gitLib/first_repo$ git log --oneline
```

e0c20e6 (HEAD -> master) Fourth system message has been added!

24af2fa Third system message has been added!

aba3bd9 Second system message has been added!

650bf2c First system message has been modified!

0ef0242 Our first system message!



Now we have 5 different commits for our file. Remember that numbers before commit message are ID of that commit. Whole message in Message1 file is:

```
This is the first system message!  
Second system message:Welcome!  
Third system message:Analyze!  
Fourth system message:Proceed!
```

Let's say we want to get state of directory where was 2<sup>nd</sup> commit. Find ID of the second commit. In this case that would be **aba3bd9**. We will execute checkout command to that ID commit.

```
a413@mN:~/gitLib/first_repo$ git checkout aba3bd9  
Note: switching to 'aba3bd9'.
```

You are in 'detached HEAD' state. You can look around, make experimental changes and commit them, and you can discard any commits you make in this state without impacting any branches by switching back to a branch. If you want to create a new branch to retain commits you create, you may do so (now or later) by using -c with the switch command. Example:

```
git switch -c <new-branch-name>
```

Or undo this operation with:

```
git switch -
```

Turn off this advice by setting config variable advice.detachedHead to false

HEAD is now at aba3bd9 Second system message has been added!

```
=====
a413@mN:~/gitLib/first_repo$ git status
HEAD detached at aba3bd9
nothing to commit, working tree clean
a413@mN:~/gitLib/first_repo$ git log --online
aba3bd9 (HEAD) Second system message has been added!
650bf2c First system message has been modified!
0ef0242 Our first system message!
```

This state is like this because head is detached to 2<sup>nd</sup> commit, but HEAD, the last 4<sup>th</sup> message still exist!

=====

Whole message in Message1 file now looks like this:

```
This is the first system message!  
Second system message:Welcome!
```

Now, we add new line into file.

```
a413@mN:~/gitLib/first_repo$ echo "Fifth system message:Done!" >> message1.txt
a413@mN:~/gitLib/first_repo$ git commit -am "Fifth system message has been added!"
```

Now check our log to see the difference:

```
a413@mN:~/gitLib/first_repo$ git log --online
759aec8 (HEAD) Fifth system message has been added!
aba3bd9 Second system message has been added!
650bf2c First system message has been modified!
0ef0242 Our first system message!
```

What if we go back now to master branch? Which commits will we have?

```
a413@mN:~/gitLib/first_repo$ git checkout master
```

Warning: you are leaving 1 commit behind, not connected to any of your branches:

```
759aec8 Fifth system message has been added!
```

If you want to keep it by creating a new branch, this may be a good time to do so with:

```
git branch <new-branch-name> 759aec8
```

Switched to branch 'master'

=====

Git inform us that there is one commit that is leaving behind and he is not connected to any of our branches. Now when we execute git log we will get this:

```
a413@mN:~/gitLib/first_repo$ git log --online
```

```
e0c20e6 (HEAD -> master) Fourth system message has been added!
```

```
24af2fa Third system message has been added!
```

```
aba3bd9 Second system message has been added!
```

```
650bf2c First system message has been modified!
```

```
0ef0242 Our first system message!
```

As we can see, here there is no 5<sup>th</sup> commit. Now we will again switch to second commit.

```
a413@mN:~/gitLib/first_repo$ git checkout aba3bd9
```

```
a413@mN:~/gitLib/first_repo$ git status
```

```
HEAD detached at aba3bd9
```

```
nothing to commit, working tree clean
```

```
a413@mN:~/gitLib/first_repo$ git log --online
```

```
aba3bd9 (HEAD) Second system message has been added!
```

```
650bf2c First system message has been modified!
```

```
0ef0242 Our first system message!
```

=====

Now, as we can see 5<sup>th</sup> message is gone from our git repository. Now we will create again 5<sup>th</sup> message, only this time we will create also new branch for our modification.

```
a413@mN:~/gitLib/first_repo$ git checkout -b feature/task5
```

```
Switched to a new branch 'feature/task5'
```

Now we defined new branch for our new change, so in case that we switch to master branch, we won't lose 5<sup>th</sup> message, when we go back to feature/task5 branch.

```
a413@mN:~/gitLib/first_repo$ echo "Fifth system message:Done!" >> message1.txt
```

```
a413@mN:~/gitLib/first_repo$ git commit -am "Fifth system message has been added!"
```

```
a413@mN:~/gitLib/first_repo$ git log --online
```

```
e20341d (HEAD -> feature/task5) Fifth system message has been added!
```

```
aba3bd9 Second system message has been added!
```

```
650bf2c First system message has been modified!
```

```
0ef0242 Our first system message!
```

```
a413@mN:~/gitLib/first_repo$ git checkout master
```

```
Switched to branch 'master'
```

If we now check log from master branch, we will see that there is no 5<sup>th</sup> log.

```
a413@mN:~/gitLib/first_repo$ git log --oneline
e0c20e6 (HEAD -> master) Fourth system message has been added!
24af2fa Third system message has been added!
aba3bd9 Second system message has been added!
650bf2c First system message has been modified!
0ef0242 Our first system message!
```

Now, when we go back to branch that we created, we will see that last 5<sup>th</sup> log is still there.

```
a413@mN:~/gitLib/first_repo$ git checkout feature/task5
Switched to branch 'feature/task5'

a413@mN:~/gitLib/first_repo$ git log --oneline
e20341d (HEAD -> feature/task5) Fifth system message has been added!
aba3bd9 Second system message has been added!
650bf2c First system message has been modified!
0ef0242 Our first system message!
```

Whole message in Message1 file at **feature/task5** branch is:

```
This is the first system message!
Second system message>Welcome!
Fifth system message:Done!
```

## REVERT CHANGES

Now we can delete branch if we don't need it:

```
a413@mN:~/gitLib/first_repo$ git branch
feature/task5
* master
```

```
a413@mN:~/gitLib/first_repo$ git branch -d feature/task5
error: The branch 'feature/task5' is not fully merged.
If you are sure you want to delete it, run 'git branch -D feature/task5'.
```

```
a413@mN:~/gitLib/first_repo$ git branch -D feature/task5
Deleted branch feature/task5 (was 749f509).
```

```
a413@mN:~/gitLib/first_repo$ echo "Fifth system message:Done!" >> message1.txt
```

```
a413@mN:~/gitLib/first_repo$ git commit -am "Fifth system message has been added!"
[master 140c7a6] Fifth system message has been added!
1 file changed, 1 insertion(+)
```

```
a413@mN:~/gitLib/first_repo$ git log --oneline
140c7a6 (HEAD -> master) Fifth system message has been added!
e0c20e6 Fourth system message has been added!
24af2fa Third system message has been added!
aba3bd9 Second system message has been added!
650bf2c First system message has been modified!
0ef0242 Our first system message!
```

Now, we don't want to delete whole branch, we need to revert only last commit, by git revert HEAD. What we do is that we undo last changes, and modify message for that commit that has been reverse. That means that our fifth message won't be anymore in our file.

```
a413@mN:~/gitLib/first_repo$ git revert HEAD
[master e1b4b1e] Revert "Fifth system message has been added!"
1 file changed, 1 deletion(-)
```

```
a413@mN:~/gitLib/first_repo$ git log --oneline
e1b4b1e (HEAD -> master) Revert "Fifth system message has been added!"
140c7a6 Fifth system message has been added!
e0c20e6 Fourth system message has been added!
24af2fa Third system message has been added!
aba3bd9 Second system message has been added!
650bf2c First system message has been modified!
0ef0242 Our first system message!
```

Now in our file we have:

```
This is the first system message!
Second system message:Welcome!
Third system message:Analyze!
Fourth system message:Proceed!
```

## RESETING GIT REPOSITORY

Revert is safe way to undo changes. With reset, there is no way to bring back original copy, it is permanent undo. This is for local changes, you should never reset snapshots, that has been shared with other developers!

When we have two modification for two different file, on the same stage, and we use "git add .", and then:

**git reset** > Will remove from stage any number of files that has been add on stage!

**git reset --hard**

On branch master

Your branch is up-to-date with 'origin/master'.

Nothing to commit, working directory is clean!

This function will right away tell us which commit now is HEAD, and inform us that working directory is clean!

**git reset d21a539**

This function will reset our repository to specific ID commit, in this case, d21a539, and now this is HEAD! We didn't delete next commit, modification from that commit are in unstaged state!

**git reset --hard a53g46h**

This function delete everything after this commit ID!

Example: HEAD is now at a53g46h 21<sup>st</sup> commit – Controller has been changed

Let's check all of this in our git repository:

```
a413@mN:~/gitLib/first_repo$ git log --oneline
e1b4b1e (HEAD -> master) Revert "Fifth system message has been added!"
140c7a6 Fifth system message has been added!
e0c20e6 Fourth system message has been added!
24af2fa Third system message has been added!
aba3bd9 Second system message has been added!
650bf2c First system message has been modified!
0ef0242 Our first system message!
```

```
a413@mN:~/gitLib/first_repo$ git reset --hard 24af2fa
HEAD is now at 24af2fa Third system message has been added!
```

```
a413@mN:~/gitLib/first_repo$ git log --oneline (All after 3rd is deleted!)
```

## CLEANING REPOSITORY

Reset only affect on tracing specific file!

We have new directory, that we init with git, and two new files. One we will put on stage, while second we won't.

```
git clean -n          > -n stands for notify, just to inform us what will clean do!  
Would remove clean-demo-file > This file is put on stage!
```

```
git clean -f          > This will remove tracked file for working directory  
git clean -f nameOfDirectory/ > This will remove untracked files from directory  
git clean -df         > This will remove untracked files and also untracked directory
```

```
git clean -xf  
> This will remove untracked files from current directory and any files that git will ignore, specified in .gitignore!
```

## GET REPOSITORY FROM GITHUB

Go to <https://github.com/join> and create your account.

Log in into your new GITHUB account. Then go to in another tab to:

<https://github.com/Architect413/demo-checkout-commit>

On GITHUB, you will see Fork function! Click on that, so the project can be transferred to your GITHUB!

It will be transferred to your account, and now you will have this repository on your GITHUB account. Now we can clone from our GITHUB account remote repository to our local repository. Or we can simply clone from originally location, if that repository is open for public clone. To clone project from GITHUB go to “Clone or Download” section and copy link for that project, then go to terminal:

```
a413@mN:~/gitLib/first_repo$ cd ..  
a413@mN:~/gitLib$ git clone https://github.com/Architect413/demo-checkout-commit.git  
a413@mN:~/gitLib$ ls  
demo-checkout-commit first_repo
```

## PUSHING INTO AND PULLING FROM GITHUB

```
-command "git pull" > Allows developer to sync with remote repository  
-command "git push" > Developer pushes his local changes or commits into remote repository
```

We have remote repository, that we have cloned into our local repository.

We have changed something on remote repository, and now we want to pull that info via git.

When we clone repository we created connection between local and remote repository.

```
git pull origin master > origin is remote repository, and master is master branch
```

After we stage, and commit something, we can push that changes to remote repository.

```
git push origin master
```

## TALKING WITH GITHUB VIA SSH (SECURE SOCKET SHELL)

- SSH KEYS > Way to identify trusted computers
- PUBLIC KEY > Public key is meant to be shared publicly
- PRIVATE KEY > Private key is meant to be kept securely to oneself
- KEY GENERATOR > Generate public-private key-pair

### SSH PROTOCOL

We have one public and one private key. Private key will stay in our personal computer, and public key we will upload to our server. Each time we want to connect, server will try to match those two keys.

```
a413@mN:~$ mkdir .ssh && cd .ssh
a413@mN:~/.ssh$ ssh-keygen -t rsa -b 8192 -N 12345 -f id_rsa
```

- RSA** Cryptosystem > RSA is one of the first practical public-key cryptosystems
- RSA Key Length** > Minimum 768 bits; Default 2048bits;
- N flag** > Write password for your key
- f flag** > Define file in which key will be stored

Setup SSH Environment:

```
sudo apt-get install sshpass
sudo apt-get install openssh-server
```

## ADD SSH TO AGENT AND COPY KEY TO GITHUB

If we want to connect with public key, we have sender and receiver of connection.

Receiver must tell sender PASSPHRASE of his private key. Sender must provide for receiver his public key.

Check if there is any keys in ~/.ssh directory:

```
a413@masterNode ~ $ ls -l ~/.ssh
total 8
-rw----- 1 a413 a413 1766 Aug 24 07:04 id_rsa
-rw-r--r-- 1 a413 a413 397 Aug 24 07:04 id_rsa.pub
```

If we want to delete keys, and generate new ones: **sudo rm -rf ~/.ssh**

Start SSH agent in background!

```
a413@mN ~ $ eval "$(ssh-agent -s)" > For example: Agent pid 22920

a413@mN ~ $ ssh-add ~/.ssh/id_rsa > Add key to SSH Agent
Enter passphrase for /home/a413/.ssh/id_rsa: > Enter passphrase if your key has the one
Identity added: /home/a413/.ssh/id_rsa (/home/a413/.ssh/id_rsa)
```

```
a413@mN ~ $ sudo apt-get y install xclip && sudo apt-get -y install geomview
```

```
a413@mN ~ $ cat ~/.ssh/id_rsa.pub | xclip -selection clipboard > Tool for copy content from specific file!
```

This last command copy output of cat command, so you can paste now that output wherever you want in system. Go to GITHUB, go to Settings section, and enter on SSH KEYS. Now paste your PUBLIC KEY, that you copied from CLIPBOARD, and give him the name. GitHub will redirect you to enter your password from GitHub as security check.

## CHECKING YOUR CONNECTION WITH GITHUB AND PUSH TO GITHUB REPO

Default port for SSH protocol is 443. Now, we want to authenticate ourself for GITHUB account.

```
a413@masterNode ~ $ ssh -T -p 443 git@ssh.github.com
The authenticity of host '[ssh.github.com]:443 ([192.30.253.123]:443)' can't be established.
RSA key fingerprint is SHA256:nThbg6kXUpJWG17E1IGOCspRomTxdCARLviKw6E5SY8.
```

Are you sure you want to continue connecting (yes/no)? Yes

(Enter your passphrase if your private key does have one!)

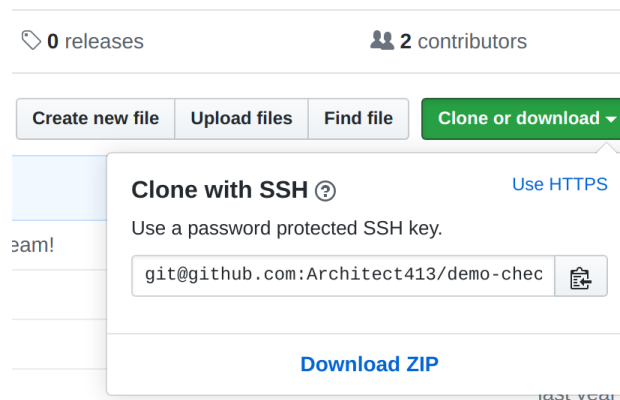
Warning: Permanently added '[ssh.github.com]:443,[192.30.253.123]:443' (RSA) to the list of known hosts.

Hi Architect413! You've successfully authenticated, but GitHub does not provide shell access.

Now we can clone our repository over SSH:

```
a413@mN:~/gitLib$ ls
demo-checkout-commit  first_repo
a413@mN:~/gitLib$ sudo rm -rf -R demo-checkout-commit/
a413@mN:~/gitLib$ ls
first_repo
```

Go to our repository on GITHUB, enter into CLONE OR DOWNLOAD section, and take link for SSH transfer:



```
a413@mN:~/gitLib$ git clone git@github.com:Architect413/demo-checkout-commit.git
```

Cloning into 'demo-checkout-commit'...

The authenticity of host 'github.com (140.82.118.4)' can't be established.

RSA key fingerprint is SHA256:nThbg6kXUpJWG17E1IGOCspRomTxdCARLviKw6E5SY8.

Are you sure you want to continue connecting (yes/no)? Yes

Warning: Permanently added 'github.com,140.82.118.4' (RSA) to the list of known hosts.

```
a413@mN:~/gitLib$ ls
first_repo demo-checkout-commit
```



## MERGE BRANCH > NO-FF-BRANCH

The `--no-ff` flag prevents git merge from executing a "fast-forward" if it detects that your current HEAD is an ancestor of the commit you're trying to merge. A fast-forward is when, instead of constructing a merge commit, git just moves your branch pointer to point at the incoming commit. This commonly occurs when doing a git pull without any local changes.

```
a413@mN:~/gitLib/first_repo$ git log --online
24af2fa (HEAD -> master) Third system message has been added!
aba3bd9 Second system message has been added!
650bf2c First system message has been modified!
0ef0242 Our first system message!
a413@mN:~/gitLib/first_repo$ git checkout -b feature/task4
Switched to a new branch 'feature/task4'
a413@mN:~/gitLib/first_repo$ echo "Fourth system message:Proceed!" >> message1.txt
a413@mN:~/gitLib/first_repo$ git commit -am "Fourth system message has been added!"
[feature/task4 f14dffe] Fourth system message has been added!
1 file changed, 1 insertion(+)
a413@mN:~/gitLib/first_repo$ echo "Fifth system message:Done!" >> message1.txt
a413@mN:~/gitLib/first_repo$ git commit -am "Fifth system message has been added!"
[feature/task4 4254085] Fifth system message has been added!
1 file changed, 1 insertion(+)

a413@mN:~/gitLib/first_repo$ git log --online --decorate --graph
* 4254085 (HEAD -> feature/task4) Fifth system message has been added!
* f14dffe Fourth system message has been added!
* 24af2fa (master) Third system message has been added!
* aba3bd9 Second system message has been added!
* 650bf2c First system message has been modified!
* 0ef0242 Our first system message!
```

```
a413@mN:~/gitLib/first_repo$ git checkout master
Switched to branch 'master'
a413@mN:~/gitLib/first_repo$ git merge feature/task4 --no-ff
Merge made by the 'recursive' strategy.
message1.txt | 2 ++
1 file changed, 2 insertions(+)
a413@mN:~/gitLib/first_repo$ git log --online --decorate --graph
* e1eaf23 (HEAD -> master) Merge branch 'feature/task4'
|\
| * 4254085 (feature/task4) Fifth system message has been added!
| * f14dffe Fourth system message has been added!
|/
* 24af2fa Third system message has been added!
* aba3bd9 Second system message has been added!
* 650bf2c First system message has been modified!
* 0ef0242 Our first system message!
```

**What happened here?** We didn't remove data from `feature/task4` branch. We just merge that data with master branch without removing data from that branch, so we can continue to do some stuff on that branch, even if we merge that state, at specific moment, to master branch.

## AMENDING GIT COMMITS – CHANGING COMMIT MESSAGE

If the **commit** only exists in your local repository and has not been pushed to GitHub, you can **amend** the **commit** message with the **git commit --amend** command.

```
a413@masterNode ~/gitMainRepository/demo-checkout-commit $ git log --online
```

```
b14044e Our first 3-way merge!
```

```
3bc9cee Info about team number 3!
```

```
c068184 Modified Solution Focus!
```

```
a413@masterNode ~/gitMainRepository/demo-checkout-commit $ nano Message2.txt
```

```
!!br0ken!!
```

```
[master 66ee579] Our first 3-way merge!
```

```
Date: Fri Aug 24 13:14:08 2018 +0200
```

```
a413@masterNode ~/gitMainRepository/demo-checkout-commit $ git log --online
```

```
66ee579 Our first 3-way merge!
```

```
3bc9cee Info about team number 3!
```

```
c068184 Modified Solution Focus!
```

```
a413@masterNode ~/gitMainRepository/demo-checkout-commit $ nano Message3.txt
```

```
a413@masterNode ~/gitMainRepository/demo-checkout-commit $ git add Message3.txt
```

```
a413@masterNode ~/gitMainRepository/demo-checkout-commit $ git commit --amend
```

```
[master 0229304] Our first 3-way merge! Team 3 info has been modified!
```

```
Date: Fri Aug 24 13:14:08 2018 +0200
```

```
a413@masterNode ~/gitMainRepository/demo-checkout-commit $ git log --online
```

```
0229304 Our first 3-way merge! Team 3 info has been modified!
```

```
3bc9cee Info about team number 3!
```

```
c068184 Modified Solution Focus!
```

After this you have modified file with last changes in unstaged status!

## REBASE

In REBASE Function you can define: ID, Branch Name, a tag, or relative reference to HEAD!  
To maintain log project history we use: `git rebase master`

**Explanation:** After creating new feature branch, master had progress let's say 2 commits. When we create 2 new commits on new feature branch, after rebase, those 2 commits from new feature branch, become two new steps, after last commit on master branch.

```
a413@mN:~/gitLib/first_repo$ git log --online
e1eaf23 (HEAD -> master) Merge branch 'feature/task4'
4254085 (feature/task4) Fifth system message has been added!
f14dffe Fourth system message has been added!
24af2fa Third system message has been added!
.....
```

```
a413@mN:~/gitLib/first_repo$ git reset --hard f14dffe
HEAD is now at f14dffe Fourth system message has been added!
a413@mN:~/gitLib/first_repo$ git log --online
f14dffe (HEAD -> master) Fourth system message has been added!
24af2fa Third system message has been added!
.....
```

```
a413@mN:~/gitLib/first_repo$ git checkout -b feature/task5
Switched to a new branch 'feature/task5'
a413@mN:~/gitLib/first_repo$ git log --online
f14dffe (HEAD -> feature/task5, master) Fourth system message has been added!
24af2fa Third system message has been added!
.....
```

```
a413@mN:~/gitLib/first_repo$ echo "Fifth system message:Done!" >> message1.txt
a413@mN:~/gitLib/first_repo$ git commit -am "Fifth system message has been added!"
[feature/task5 5ffabe2] Fifth system message has been added!
1 file changed, 1 insertion(+)
a413@mN:~/gitLib/first_repo$ echo "Sixth system message:Testing!" >> message1.txt
a413@mN:~/gitLib/first_repo$ git commit -am "Sixth system message has been added!"
[feature/task5 6224156] Sixth system message has been added!
1 file changed, 1 insertion(+)
a413@mN:~/gitLib/first_repo$ git checkout master
Switched to branch 'master'
a413@mN:~/gitLib/first_repo$ echo "Seventh system message:Passed!" >> message1.txt
a413@mN:~/gitLib/first_repo$ git commit -am "Seventh system message has been added!"
[master 94261bb] Seventh system message has been added!
1 file changed, 1 insertion(+)
a413@mN:~/gitLib/first_repo$ git log --online --decorate --graph --all
* 94261bb (HEAD -> master) Seventh system message has been added!
| * 6224156 (feature/task5) Sixth system message has been added!
| * 5ffabe2 Fifth system message has been added!
|/
* f14dffe Fourth system message has been added!
* 24af2fa Third system message has been added!
.....
a413@mN:~/gitLib/first_repo$ git checkout feature/task5
Switched to branch 'feature/task5'
```

```
a413@mN:~/gitLib/first_repo$ git rebase master
```

First, rewinding head to replay your work on top of it...

Applying: Fifth system message has been added!

Using index info to reconstruct a base tree...

M message1.txt

Falling back to patching base and 3-way merge...

Auto-merging message1.txt

CONFLICT (content): Merge conflict in message1.txt

error: Failed to merge in the changes.

**Patch failed at 0001 Fifth system message has been added!**

hint: Use 'git am --show-current-patch' to see the failed patch

Resolve all conflicts manually, mark them as resolved with "git add/rm <conflicted\_files>", then run "git rebase --continue".

You can instead skip this commit: run "git rebase --skip".

To abort and get back to the state before "git rebase", run "git rebase --abort".

```
=====
a413@mN:~/gitLib/first_repo$ git rebase --continue
```

message1.txt: needs merge

You must edit all merge conflicts and then

mark them as resolved using git add

**Note:**

When you have resolved this problem, run "git rebase --continue".

If you prefer to skip this patch, run "git rebase --skip" instead.

To check out the original branch and stop rebasing, run "git rebase --abort".

```
a413@mN:~/gitLib/first_repo$ git branch
```

\* (no branch, rebasing feature/task5)

feature/task5

master

```
=====
REMEMBER: Never changes the same files over two branches, because if you see, we have:
```

**2 modification on Message3.txt on master branch!**

**2 modification on Solution1.txt on top-feature branch!**

**1 modification on Message1.txt and 1 modification on Solution1.txt on master branch! >> CONFLICT!**

Because of this we can't rebase our branch! To resolve conflict, we need to go back on master branch, undo changes in last commit which is related to file Solution1.txt, then go again on top-feature branch, and do again rebase!

```
a413@mN:~/gitLib/first_repo$ git rebase --abort
```

```
a413@mN:~/gitLib/first_repo$ git branch
```

\* feature/task5

master

```
a413@mN:~/gitLib/first_repo$ git checkout master
```

Switched to branch 'master'

```
=====
Now delete last line that we added in our message1.txt file on master branch. We can do that with nano editor.
```

Last line that we added was **"Seventh system message:Passed!"**:

```
a413@mN:~/gitLib/first_repo$ sudo nano message1.txt
```

```
a413@mN:~/gitLib/first_repo$ git commit -am "Resolving conflict Rebase in message1.txt!"
```

[master 88d4fd7] Resolving conflict Rebase in message1.txt!

1 file changed, 1 deletion(-)

```
a413@mN:~/gitLib/first_repo$ git branch
```

```
feature/task5
```

```
* master
```

```
a413@mN:~/gitLib/first_repo$ git checkout feature/task5
```

```
Switched to branch 'feature/task5'
```

```
a413@mN:~/gitLib/first_repo$ git rebase master
```

```
First, rewinding head to replay your work on top of it...
```

```
Applying: Fifth system message has been added!
```

```
Applying: Sixth system message has been added!
```

```
a413@mN:~/gitLib/first_repo$ git log --oneline --decorate --graph --all
```

```
* 4aa3940 (HEAD -> feature/task5) Sixth system message has been added!
```

```
* 5e110cf Fifth system message has been added!
```

```
* 88d4fd7 (master) Resolving conflict Rebase in message1.txt! >>This resolved conflict!
```

```
* 94261bb Seventh system message has been added!
```

```
* f14dffe Fourth system message has been added!
```

```
* 24af2fa Third system message has been added!
```

```
.....
```

```
=====
```

Now everything from maser branch has been added on top-feature branch, like they always been there.

Now we can merge branch feature/task5 with master.

```
a413@mN:~/gitLib/first_repo$ git branch
```

```
* feature/task5
```

```
master
```

```
a413@mN:~/gitLib/first_repo$ git checkout master
```

```
Switched to branch 'master'
```

```
a413@mN:~/gitLib/first_repo$ git merge feature/task5
```

```
Updating 88d4fd7..4aa3940
```

```
Fast-forward
```

```
message1.txt | 2 ++
```

```
1 file changed, 2 insertions(+)
```

```
=====
```

```
a413@mN:~/gitLib/first_repo$ git log --oneline --decorate --graph --all
```

```
* 4aa3940 (HEAD -> master, feature/task5) Sixth system message has been added!
```

```
* 5e110cf Fifth system message has been added!
```

```
* 88d4fd7 Resolving conflict Rebase in message1.txt!
```

```
* 94261bb Seventh system message has been added!
```

```
.....
```

```
a413@mN:~/gitLib/first_repo$ git branch -d feature/task5
```

```
Deleted branch feature/task5 (was 4aa3940).
```

```
a413@mN:~/gitLib/first_repo$ git log --oneline --decorate --graph --all
```

```
* 4aa3940 (HEAD -> master) Sixth system message has been added!
```

```
* 5e110cf Fifth system message has been added!
```

```
* 88d4fd7 Resolving conflict Rebase in message1.txt!
```

```
* 94261bb Seventh system message has been added!
```

```
.....
```

## GIT FETCH

When you **fetch**, Git gathers any commits from the target branch that do not exist in your current branch and **stores them in your local repository**. However, **it does not merge them with your current branch**. This is particularly useful if you need to keep your repository up to date, but are working on something that might break if you update your files. To integrate the commits into your master branch, you use merge.

When you use **pull**, Git tries to automatically do your work for you. It is **context sensitive**, so Git will merge any pulled commits into the branch you are currently working in. pull **automatically merges the commits without letting you review them first**. If you don't closely manage your branches, you may run into frequent conflicts. In the simplest terms, git pull does a git fetch followed by a git merge. You can do a git fetch at any time to update your remote-tracking branches.

- **git fetch** is the command that says "bring my local copy of the remote repository up to date."
- **git pull** says "bring the changes in the remote repository to where I keep my own code."

Normally **git pull** does this by doing a **git fetch** to bring the local copy of the remote repository up to date, and then merging the changes into your own code repository and possibly your working copy.

Go to [git@github.com:Architect413/demo-checkout-commit](https://github.com/Architect413/demo-checkout-commit) and fork that repository to your GITHUB account. After that clone that repository from your remote to local repository. In this example, I will clone mine:

```
a413@mN:~/gitLib$ sudo rm -rf -R demo-checkout-commit/
a413@mN:~/gitLib$ git clone git@github.com:Architect413/demo-checkout-commit.git
a413@mN:~/gitLib$ cd demo-checkout-commit/
a413@mN:~/gitLib/demo-checkout-commit$ git push -u origin master
Branch 'master' set up to track remote branch 'master' from 'origin'. Everything up-to-date
```

Now, we can check how branches are connected between remote and local repository:

```
a413@mN:~/gitLib/demo-checkout-commit$ git remote -v
origin https://github.com/Architect413/demo-checkout-commit.git (fetch)
origin https://github.com/Architect413/demo-checkout-commit.git (push)

a413@mN:~/gitLib/demo-checkout-commit$ git log --oneline --decorate --graph --all
* 92ff416 (HEAD -> master, origin/master, origin/HEAD) file_for_pull_request
* 5a0ac55 Team_2 go go go!
.....
```

Now we see all the commits that has been made in past inside of this repository. After that each time, if there is something new, as update, on remote repository, and we don't have that new changes on our local, we can use fetch, to get those updates also on our local repository.

```
a413@mN:~/gitLib/demo-checkout-commit$ git fetch
```

If output is empty, we don't have any new updates. Now, we want to make change, on remote branch, on GITHUB repository. Enter into repository and go to CREATE NEW FILE.

testSSH

[Manage topics](#)

Edit

26 commits 1 branch 0 releases 2 contributors

Branch: master New pull request

Create new file Upload files Find file Clone or download

Architect413 file\_for\_pull\_request ... Latest commit 92ff416 on Jun 13

Message1.txt	master:Team_1 - Info about connections in the team!	last year
Message2.txt	Team_2 go go go!	last year
Message3.txt	master:added info about name of team 3!	last year
Solution1.txt	Solution_1 Advice!	last year
file_for_pull_request	file_for_pull_request	5 months ago

Then add name of the file and content of the file. After that scroll down to commit changes.

Architect413 / demo-checkout-commit Watch 0 Star 0 Fork 0

[Code](#) [Pull requests 0](#) [Projects 0](#) [Wiki](#) [Security](#) [Insights](#) [Settings](#)

demo-checkout-commit / Message4.txt Cancel

Edit new file Preview Spaces 2 No wrap

```
1 Code is on testing! Wait until testing is completed.
2 Until then watch monitoring, is there any red points on monitor.
```

Create commit, add description and execute commit.

Commit new file

Create new system message - Message 4

Description about testing

☒ Commit directly to the master branch.

☐ Create a new branch for this commit and start a pull request. [Learn more about pull requests.](#)

Commit new file Cancel



Now we can see in our remote repository, new changes.

Branch: master ▾	New pull request	Create new file	Upload files	Find file	Clone or download ▾
Architect413 Create new system message - Message 4 ...					Latest commit d4d0d2a now
Message1.txt	master:Team_1 - Info about connections in the team!				last year
Message2.txt	Team_2 go go go!				last year
Message3.txt	master:added info about name of team 3!				last year
Message4.txt	Create new system message - Message 4				now
Solution1.txt	Solution_1 Advice!				last year
file_for_pull_request	file_for_pull_request				5 months ago

=====

Now we can execute git fetch to see how our local repository will grab new updates from GITHUB.

```
a413@mN:~/gitLib/demo-checkout-commit$ git fetch
```

```
Warning: Permanently added the RSA host key for IP address '140.82.118.3' to the list of known hosts.
```

```
remote: Enumerating objects: 3, done.
```

```
remote: Counting objects: 100% (3/3), done.
```

```
remote: Compressing objects: 100% (3/3), done.
```

```
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
```

```
Unpacking objects: 100% (3/3), done.
```

```
From github.com:Architect413/demo-checkout-commit
```

```
92ff416..d4d0d2a master -> origin/master
```

```
a413@mN:~/gitLib/demo-checkout-commit$ git log --oneline --decorate --graph --all
```

```
* d4d0d2a (origin/master, origin/HEAD) Create new system message - Message 4
```

```
* 92ff416 (HEAD -> master) file_for_pull_request
```

```
* 5a0ac55 Team_2 go go go!
```

```
.....
```

```
a413@mN:~/gitLib/demo-checkout-commit$ git branch -r (list of remote branches on tracking)
```

```
origin/HEAD -> origin/master
```

```
origin/master
```

```
a413@mN:~/gitLib/demo-checkout-commit$ git branch -a
```

```
* master
```

```
remotes/origin/HEAD -> origin/master
```

```
remotes/origin/master
```

```
a413@mN:~/gitLib/demo-checkout-commit$ git merge origin/master
```

```
Updating 92ff416..d4d0d2a
```

```
Fast-forward
```

```
Message4.txt | 2 ++
```

```
1 file changed, 2 insertions(+)
```

```
create mode 100644 Message4.txt
```

```
a413@mN:~/gitLib/demo-checkout-commit$ git log --oneline --decorate --graph --all
```

```
* d4d0d2a (HEAD -> master, origin/master, origin/HEAD) Create new system message - Message 4
```

```
* 92ff416 file_for_pull_request
```

```
* 5a0ac55 Team_2 go go go!
```

```
.....
```

## Advice:

Fetch before you start your day's work, before you push to remote repository and stay in sync as much as possible!



## GIT PULL WITH REBASE

`GIT PULL --REBASE <REMOTE-NAME> <BRANCH-NAME>`

Create repository on GitHub account under name: demo-pull-rebase



On home page of your GITHUB account enter into REPOSITORIES tab. Click on NEW button.

Overview **Repositories 3** Projects 1 Packages 0 Stars 0 Followers 1 Following 0

---

Find a repository... Type: All Language: All 

Owner **Repository name \***

 Architect413 / demo-pull-rebase 

Great repository names are short and memorable. Need inspiration? How about **fictional-guacamole**?

Description (optional)


pull-testing


☒ **Public**  
Anyone can see this repository. You choose who can commit.

☐ **Private**  
You choose who can see and commit to this repository.

Skip this step if you're importing an existing repository.

☐ **Initialize this repository with a README**  
This will let you immediately clone the repository to your computer.

Add .gitignore: None Add a license: None 



Enter name of new repository.  
Add description.

Setup repository as public.

Click on CREATE REPOSITORY.

Now, on our local machine, create demo-pull-rebase directory for our GIT repository.

```
a413@mN:~/gitLib$ mkdir demo-pull-rebase && cd demo-pull-rebase/
a413@mN:~/gitLib/demo-pull-rebase$ git init
Initialized empty Git repository in /home/a413/gitLib/demo-pull-rebase/.git/
a413@mN:~/gitLib/demo-pull-rebase$ git remote add origin git@github.com:Architect413/demo-pull-rebase.git

a413@mN:~/gitLib/demo-pull-rebase$ git remote -v
origin    git@github.com:Architect413/demo-pull-rebase.git (fetch)
origin    git@github.com:Architect413/demo-pull-rebase.git (push)

a413@mN:~/gitLib/demo-pull-rebase$ echo "Now we start pull rebase!" >> Monitoring.txt

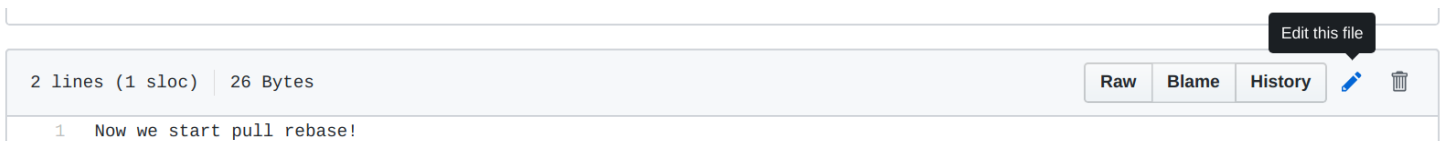
a413@mN:~/gitLib/demo-pull-rebase$ git add .
a413@mN:~/gitLib/demo-pull-rebase$ git commit -am "Test Pull Rebase!"
[master (root-commit) ba2e3c0] Test Pull Rebase!
 1 file changed, 1 insertion(+)
 create mode 100644 Monitoring.txt
a413@mN:~/gitLib/demo-pull-rebase$ git push -u origin master
Enumerating objects: 3, done.
To github.com:Architect413/demo-pull-rebase.git
 * [new branch]      master -> master
Branch 'master' set up to track remote branch 'master' from 'origin'.
```

Now check on your GITHUB account demo-pull-rebase to see updates which we pushed from local repo to remote.

```
=====
a413@mN:~/gitLib/demo-pull-rebase$ echo "ID of our engine is: 1918176543!" >> Monitoring.txt
a413@mN:~/gitLib/demo-pull-rebase$ git commit -am "local:added info about engine!"
[master 9830796] local:added info about engine!
 1 file changed, 1 insertion(+)
a413@mN:~/gitLib/demo-pull-rebase$ echo "Engine has 7 parts and 16 valves!" >> Monitoring.txt
a413@mN:~/gitLib/demo-pull-rebase$ git commit -am "local:added info about engine structure!"
[master 1c8a813] local:added info about engine structure!
 1 file changed, 1 insertion(+)
a413@mN:~/gitLib/demo-pull-rebase$ echo "Engine price is lower then expected!" >> Monitoring.txt
a413@mN:~/gitLib/demo-pull-rebase$ git commit -am "local:added info about engine price!"
[master 7ca7af2] local:added info about engine price!
 1 file changed, 1 insertion(+)
=====
```

```
=====
a413@mN:~/gitLib/demo-pull-rebase$ git log --oneline --decorate --graph --all
* 7ca7af2 (HEAD -> master) local:added info about engine price!
* 1c8a813 local:added info about engine structure!
* 9830796 local:added info about engine!
* ba2e3c0 (origin/master) Test Pull Rebase!
=====
```

Now, go to remote repo, do two changes with two different commits. Let commit names be “remote:first impression” and “remote:performances”. When you open Monitoring.txt file, or right side you will see button for Edit.



Execute two different commits, on this way.

A screenshot of a "Commit changes" dialog box. At the top left is a small icon of a person. The title is "Commit changes". Below it is a text input field containing "remote:first impression". Underneath is a larger text area with the placeholder text "Desc about impression". At the bottom, there are two radio button options: the first is selected and labeled "Commit directly to the master branch.", and the second is labeled "Create a new branch for this commit and start a pull request. Learn more about pull requests.". At the very bottom are two buttons: "Commit changes" (green) and "Cancel" (grey).A screenshot of a "Commit changes" dialog box, similar to the one above. It has the same layout with a person icon, title "Commit changes", a text input field containing "remote:performances", a text area with placeholder "Desc about performances", two radio button options (the first is selected for "Commit directly to the master branch."), and "Commit changes" and "Cancel" buttons at the bottom.

After these two commits we have in our file on remote repo, this:

Now we start pull rebase!

First impression is very good!

(From `remote:first impression` commit)

Performances are much higher then expected!

(From `remote:performances` commit)

```
a413@mN:~/gitLib/demo-pull-rebase$ git pull --rebase origin master
```

```
remote: Total 6 (delta 1), reused 0 (delta 0), pack-reused 0
```

```
Unpacking objects: 100% (6/6), done.
```

```
From github.com:Architect413/demo-pull-rebase
```

```
* branch          master    -> FETCH_HEAD
   ba2e3c0..5ab8d23 master    -> origin/master
```

```
First, rewinding head to replay your work on top of it...
```

```
Applying: local:added info about engine!
```

```
Using index info to reconstruct a base tree...
```

```
M   Monitoring.txt
```

```
Falling back to patching base and 3-way merge...
```

```
Auto-merging Monitoring.txt
```

```
CONFLICT (content): Merge conflict in Monitoring.txt
```

```
error: Failed to merge in the changes.
```

```
Patch failed at 0001 local:added info about engine!
```

```
hint: Use 'git am --show-current-patch' to see the failed patch. Resolve all conflicts manually, mark them as resolved with "git add/rm <conflicted_files>", then run "git rebase --continue". You can skip this commit: run "git rebase --skip". To abort and get back to the state before "git rebase", run "git rebase --abort".
```

```
a413@mN:~/gitLib/demo-pull-rebase$ git rebase --abort
```

```
a413@mN:~/gitLib/demo-pull-rebase$ git log --oneline --decorate --graph --all
```

```
* 5ab8d23 (origin/master) remote:performances
```

```
* 409a7b1 remote:first impression
```

```
| * 7ca7af2 (HEAD -> master) local:added info about engine price!
```

```
| * 1c8a813 local:added info about engine structure!
```

```
| * 9830796 local:added info about engine!
```

```
|/
```

```
* ba2e3c0 Test Pull Rebase!
```

=====

We had conflict because we edit on remote and in the same time on local the same file. That is not allowed. That is why our rebase failed and we aborted that process. So how we can resolve that?

We will go on GITHUB account create new commit where we will delete changes that we made from “remote:first impression” and “remote:performances” commits. In that way, we will back state of repository at the moment where we didn’t have that commits.

=====

After that commit, when we enter into demo-pull-rebase repo and go into COMMITS tab, we will see:

Branch: master ▼

Commits on Oct 29, 2019

remote:delete-impression-and-performances ...	Verified		857a46c	
Architect413 committed 17 minutes ago				
remote:performances ...	Verified		5ab8d23	
Architect413 committed 23 minutes ago				
remote:first impression ...	Verified		409a7b1	
Architect413 committed 25 minutes ago				

After this we will again execute rebase, and we will get this:

```
a413@mN:~/gitLib/demo-pull-rebase$ git pull --rebase origin master
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (1/1), done.
remote: Total 3 (delta 0), reused 2 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From github.com:Architect413/demo-pull-rebase
* branch          master      -> FETCH_HEAD
  5ab8d23..857a46c master      -> origin/master
First, rewinding head to replay your work on top of it...
Applying: local:added info about engine!
Applying: local:added info about engine structure!
Applying: local:added info about engine price!
a413@mN:~/gitLib/demo-pull-rebase$ git log --online --decorate --graph --all
* d9acbf8 (HEAD -> master) local:added info about engine price!
* cbca979 local:added info about engine structure!
* e1600d0 local:added info about engine!
* 857a46c (origin/master) remote:delete-impression-and-performaces
* 5ab8d23 remote:performances
* 409a7b1 remote:first impression
* ba2e3c0 Test Pull Rebase!
```

As we can see rebase now has been successfully. We also see last commit **"remote:delete-impression-and-performaces"** from remote repository. Now we don't have conflict between remote and local repo.

Now, we will go to remote repo, create new file under name **programming-language.txt** and edit that file with text:

Our favorite programming language is Golang!

We will commit our changes under commit name **"remote:favorite programming language"**.

Now we will again execute rebase from remote repo, and we will get this:

```
a413@mN:~/gitLib/demo-pull-rebase$ git pull --rebase origin master
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From github.com:Architect413/demo-pull-rebase
* branch          master      -> FETCH_HEAD
  857a46c..1e73022 master      -> origin/master
First, rewinding head to replay your work on top of it...
Applying: local:added info about engine!
Applying: local:added info about engine structure!
Applying: local:added info about engine price!

a413@mN:~/gitLib/demo-pull-rebase$ git log --online --decorate --graph --all
* 7db123a (HEAD -> master) local:added info about engine price!
* b03d1eb local:added info about engine structure!
* 73a0bc8 local:added info about engine!
* 1e73022 (origin/master) remote:favorite programming language
* 857a46c remote:delete-impression-and-performaces
```

## GIT REFLOG

REFLOG - reflog is an acronym for reference logs. The current HEAD can get updated for multiple events such as:

- > by switching branches
- > pulling in new changes
- > rewriting history
- > simply by adding new commits

Every time when your branch is updated for any reason, git stores that information in reflog.

```
a413@mN:~/gitLib/demo-pull-rebase$ git reflog
```

```
7db123a (HEAD -> master) HEAD@{0}: rebase finished: returning to refs/heads/master
7db123a (HEAD -> master) HEAD@{1}: pull --rebase origin master: local:added info about engine price!
b03d1eb HEAD@{2}: pull --rebase origin master: local:added info about engine structure!
73a0bc8 HEAD@{3}: pull --rebase origin master: local:added info about engine!
1e73022 (origin/master) HEAD@{4}: pull --rebase origin master: checkout 1e730229a13dfceb1eb688e9954e424f304e9ace
d9acbf8 HEAD@{5}: rebase finished: returning to refs/heads/master
d9acbf8 HEAD@{6}: pull --rebase origin master: local:added info about engine price!
cbca979 HEAD@{7}: pull --rebase origin master: local:added info about engine structure!
e1600d0 HEAD@{8}: pull --rebase origin master: local:added info about engine!
857a46c HEAD@{9}: pull --rebase origin master: checkout 857a46c9267aa0f334116736261a4122c7477bb4
7ca7af2 HEAD@{10}: reset: moving to 7ca7af2
7ca7af2 HEAD@{11}: rebase: updating HEAD
5ab8d23 HEAD@{12}: pull --rebase origin master: checkout 5ab8d237c0feba1e980bf09260bdd3c65acae5f1
7ca7af2 HEAD@{13}: commit: local:added info about engine price!
1c8a813 HEAD@{14}: commit: local:added info about engine structure!
9830796 HEAD@{15}: commit: local:added info about engine!
ba2e3c0 HEAD@{16}: commit (initial): Test Pull Rebase!
```

You can sort your logs by time! Various supported forms include:

```
1.minute.ago    > 1.hour.ago    > 1.day.ago    > yesterday
1.week.ago      > 1.month.ago    > 1.year.ago    > 2015-05-28.14:13:14
```

The plural forms are also accepted e.g. 2.weeks.ago and also combinations e.g. 1.day.2.hours.ago!

Try these following examples:

```
a413@mN:~/gitLib/demo-pull-rebase$ git show HEAD@{5}
a413@mN:~/gitLib/demo-pull-rebase$ git show master@{1.hour.ago}
a413@mN:~/gitLib/demo-pull-rebase$ git show master@{1.day.ago}
a413@mN:~/gitLib/demo-pull-rebase$ git log -g master
```

If you cloned two months ago project function git show master@{2.month.ago} will work!

If we cloned hour ago, we will get no results. The reflog data is kept in the .git/logs directory.

If we use git reset --hard commit\_id, we will delete data, but log remains!

Via that log, we can back in time to specific commit!

```
git checkout HEAD@{1}
```

## TAG

We can tag specific commit in log history! Release Candidate (RC) is the build released internally to check if any critical problems have gone undetected into the code during the previous development period. Release candidates are NOT for production deployment, but they are for testing purposes only.

```
a413@mN:~/gitLib/demo-pull-rebase$ git log --online --decorate --graph --all
* 7db123a (HEAD -> master) local:added info about engine price!
* b03d1eb local:added info about engine structure!
* 73a0bc8 local:added info about engine!
* 1e73022 (origin/master) remote:favorite programming language
* 857a46c remote:delete-impression-and-performaces
* 5ab8d23 remote:performances
* 409a7b1 remote:first impression
* ba2e3c0 Test Pull Rebase!
```

Before we start to create tags, we will merge branches, by pushing local changes to remote repository.

```
a413@mN:~/gitLib/demo-pull-rebase$ git push -u origin master
Enumerating objects: 11, done.
Counting objects: 100% (11/11), done.
Delta compression using up to 32 threads
Compressing objects: 100% (9/9), done.
Writing objects: 100% (9/9), 1012 bytes | 1012.00 KiB/s, done.
Total 9 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), done.
To github.com:Architect413/demo-pull-rebase.git
 1e73022..7db123a master -> master
Branch 'master' set up to track remote branch 'master' from 'origin'.
```

```
a413@mN:~/gitLib/demo-pull-rebase$ git log --online --decorate --graph --all
* 7db123a (HEAD -> master, origin/master) local:added info about engine price!
* b03d1eb local:added info about engine structure!
* 73a0bc8 local:added info about engine!
* 1e73022 remote:favorite programming language
* 857a46c remote:delete-impression-and-performaces
* 5ab8d23 remote:performances
* 409a7b1 remote:first impression
* ba2e3c0 Test Pull Rebase!
```

Now, when everything is on the same state, let's create tags.

```
a413@mN:~/gitLib/demo-pull-rebase$ git tag v-1.7-rc1
a413@mN:~/gitLib/demo-pull-rebase$ git tag
v-1.7-rc1
a413@mN:~/gitLib/demo-pull-rebase$ git tag --list
v-1.7-rc1
a413@mN:~/gitLib/demo-pull-rebase$ git log --online
7db123a (HEAD -> master, tag: v-1.7-rc1, origin/master) local:added info about engine price!
b03d1eb local:added info about engine structure!
73a0bc8 local:added info about engine!
.....
```



```
a413@mN:~/gitLib/demo-pull-rebase$ git show v-1.7-rc1
commit 7db123a1a2f204f95d7ff4ca3307b94e7719809c (HEAD -> master, tag: v-1.7-rc1, origin/master)
Author: a413 <nikola_djurdjevic@hotmail.co.uk>
Date: Tue Oct 29 19:34:22 2019 +0100
```

local:added info about engine price!

```
diff --git a/Monitoring.txt b/Monitoring.txt
index 140bafd..fed7245 100644
--- a/Monitoring.txt
+++ b/Monitoring.txt
@@ -1,3 +1,4 @@
Now we start pull rebase!
ID of our engine is: 1918176543!
Engine has 7 parts and 16 valves!
+Engine price is lower then expected!
```

```
=====
a413@mN:~/gitLib/demo-pull-rebase$ echo "We also love to cooperate with C++!" >> programming-language.txt
a413@mN:~/gitLib/demo-pull-rebase$ git commit -am "master:modified desc about language"
[master 4c5ca3d] master:modified desc about language
1 file changed, 1 insertion(+)
```

```
a413@mN:~/gitLib/demo-pull-rebase$ git tag v-1.8-rc2
a413@mN:~/gitLib/demo-pull-rebase$ git log --oneline
4c5ca3d (HEAD -> master, tag: v-1.8-rc2) master:modified desc about language
7db123a (tag: v-1.7-rc1, origin/master) local:added info about engine price!
b03d1eb local:added info about engine structure!
73a0bc8 local:added info about engine!
1e73022 remote:favorite programming language
.....
```

```
=====
a413@mN:~/gitLib/demo-pull-rebase$ echo "We talk to each other via gRPC!" >> programming-language.txt
a413@mN:~/gitLib/demo-pull-rebase$ git commit -am "master:modified desc about protocol"
[master 322b3e6] master:modified desc about protocol
1 file changed, 1 insertion(+)
```

```
a413@mN:~/gitLib/demo-pull-rebase$ git tag -a v-1.8.1
a413@mN:~/gitLib/demo-pull-rebase$ git log --oneline
322b3e6 (HEAD -> master, tag: v-1.8.1) master:modified desc about protocol
4c5ca3d (tag: v-1.8-rc2) master:modified desc about language
7db123a (tag: v-1.7-rc1, origin/master) local:added info about engine price!
b03d1eb local:added info about engine structure!
73a0bc8 local:added info about engine!
1e73022 remote:favorite programming language
.....
```

```
a413@mN:~/gitLib/demo-pull-rebase$ git tag --list
v-1.7-rc1
v-1.8-rc2
v-1.8.1
```

## TAG TYPE

**GIT CAT-FILE** PROVIDES CONTENT OR TYPE AND SIZE INFORMATION FOR REPOSITORY OBJECTS.

```
git tag -l "v1.8.5*" == git tag --list "v1.8.5*"
```

```
a413@mN:~/gitLib/demo-pull-rebase$ git cat-file -t v-1.8-rc2    > commit
a413@mN:~/gitLib/demo-pull-rebase$ git cat-file -t v-1.8.1      > tag
```

### Annotated Tags

Creating an annotated tag in Git is simple. The easiest way is to specify `-a` when you run the `tag` command:

```
$ git tag -a v1.4 -m "my version 1.4"
```

### Lightweight Tags

Another way to tag commits is with a lightweight tag. This is basically the commit checksum stored in a file no other information is kept. To create a lightweight tag, don't supply any of the `-a`, `-s`, or `-m` options, just provide a tag name:

```
$ git tag v1.4-lw
git log --pretty=oneline
```

### Sharing Tags

By default, the `git push` command doesn't transfer tags to remote servers. You will have to explicitly push tags to a shared server after you have created them. This process is just like sharing remote branches — you can run `git push origin <tagname>`.

```
$ git push origin v1.5
```

If you have a lot of tags that you want to push up at once, you can also use the `--tags` option to the `git push` command. This will transfer all of your tags to the remote server that are not already there.

```
$ git push origin --tags
```

```
a413@mN:~/gitLib/demo-pull-rebase$ git describe --all
tags/v-1.8.1
a413@mN:~/gitLib/demo-pull-rebase$ git describe --all --tags
tags/v-1.8.1
a413@mN:~/gitLib/demo-pull-rebase$ git describe --all --tags --contains
tags/v-1.8.1 ^0
a413@mN:~/gitLib/demo-pull-rebase$ git tag | tail -n2
v-1.8-rc2
v-1.8.1
```

```
a413@mN:~/gitLib/demo-pull-rebase$ git diff v-1.8-rc2 v-1.8.1
diff --git a/programming-language.txt b/programming-language.txt
index c27b4ae..2b3bb3d 100644
--- a/programming-language.txt
+++ b/programming-language.txt
@@ -1,2 +1,3 @@
 Our favorite programming language is Golang!
 We also love to cooperate with C++!
+We talk to each other via gRPC!
```

We can transfer tag name to specific ID commit:

```
git tag -a v-3.1.0-beta --force 391b524      git tag v-3.1.0-beta --delete
```



Updated tag v-3.1.0-beta (was 763y342) Deleted tag v-3.1.0-beta (was 391b524)

git push origin :v-3.1.0-beta :v-3.1.0-alpha > Delete those two tags from remote repo!  
git push origin --tags > Send all tags to remote repo!

git push origin --follow-tags > Push only annotated tags!  
git config --global push.followTags true > Push by default all annotated tags!  
git config --global --list > Check all configuration setups!

If we push lightWeight tag, and we setup default for annotated tags, by pushing the LW, we will push them all.

git push -u origin master > “-u” creates new branching relationship when we push for the first time!

We are changing to new branch, but we are also switching our HEAD to that specific TAG:

git checkout -b branch-v-1.8 v-1.8-rc1

## GIT STASH

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.

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.

Firstly clone demo-pull-rebase repository from <https://github.com/Architect413/demo-pull-rebase> via HTTPS protocol, if you didn't do that already. Then see logs, to see is everything in sync.

```
a413@mN:~/gitLib/demo-pull-rebase$ git log --oneline
b854bf3 (HEAD -> master) master:modified desc about connection
322b3e6 (tag: v-1.8.1, origin/master) master:modified desc about protocol
4c5ca3d (tag: v-1.8-rc2) master:modified desc about language
7db123a (tag: v-1.7-rc1) local:added info about engine price!
b03d1eb local:added info about engine structure!
73a0bc8 local:added info about engine!
```

.....

Remote and local are not in sync. To sync them, we will execute push command and check again logs.

```
a413@mN:~/gitLib/demo-pull-rebase$ git push -u origin master
Enumerating objects: 8, done.
Counting objects: 100% (8/8), done.
Delta compression using up to 32 threads
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 608 bytes | 608.00 KiB/s, done.
Total 6 (delta 3), reused 0 (delta 0)
remote: Resolving deltas: 100% (3/3), completed with 1 local object.
To github.com:Architect413/demo-pull-rebase.git
 7db123a..322b3e6  master -> master
Branch 'master' set up to track remote branch 'master' from 'origin'.
```

```
a413@mN:~/gitLib/demo-pull-rebase$ git log --oneline
322b3e6 (HEAD -> master, tag: v-1.8.1, origin/master) master:modified desc about protocol
4c5ca3d (tag: v-1.8-rc2) master:modified desc about language
7db123a (tag: v-1.7-rc1) local:added info about engine price!
b03d1eb local:added info about engine structure!
73a0bc8 local:added info about engine!
```

.....

Make some changes to one file, and execute **git stash**.

```
a413@mN:~/gitLib/demo-pull-rebase$ echo "We use only HTTP2 connection!" >> programming-language.txt
a413@mN:~/gitLib/demo-pull-rebase$ git commit -am "master:modified desc about connection"
[master b854bf3] master:modified desc about connection
 1 file changed, 1 insertion(+)
a413@mN:~/gitLib/demo-pull-rebase$ git stash
No local changes to save
```

**What is the problem here?** When we have new change, we won't create new commit, but immediately create git stash:

```
a413@mN:~/gitLib/demo-pull-rebase$ echo "For every connection we use SSL!" >> programming-language.txt
```

```
a413@mN:~/gitLib/demo-pull-rebase$ git stash
```

```
Saved working directory and index state WIP on master: b854bf3 master:modified desc about connection
```

```
a413@mN:~/gitLib/demo-pull-rebase$ git stash list
```

```
stash@{0}: WIP on master: b854bf3 master:modified desc about connection
```

```
a413@mN:~/gitLib/demo-pull-rebase$ echo "For database we use Postgres!" >> programming-language.txt
```

```
a413@mN:~/gitLib/demo-pull-rebase$ git stash save "master:modified desc about databases"
```

```
Saved working directory and index state On master: master:modified desc about databases
```

```
a413@mN:~/gitLib/demo-pull-rebase$ echo "For hashing we use Redis!" >> programming-language.txt
```

```
a413@mN:~/gitLib/demo-pull-rebase$ git stash save "master:modified desc about realtime"
```

```
Saved working directory and index state On master: master:modified desc about realtime
```

```
a413@mN:~/gitLib/demo-pull-rebase$ git stash show stash@{3}
```

```
fatal: Log for 'stash' only has 3 entries.
```

```
a413@mN:~/gitLib/demo-pull-rebase$ git stash show stash@{1}
```

```
programming-language.txt | 1 +
```

```
1 file changed, 1 insertion(+)
```

**What seems to be the problem here?** Index of stash is starting for 0. So, if we have 3 stashes, indexes are 0,1 and 2. If we check in logs, to see the difference, we will see that, this changes are not recognized as commits, but as stashes.

```
a413@mN:~/gitLib/demo-pull-rebase$ git log --online
```

```
b854bf3 (HEAD -> master, origin/master) master:modified desc about connection
```

```
322b3e6 (tag: v-1.8.1) master:modified desc about protocol
```

```
4c5ca3d (tag: v-1.8-rc2) master:modified desc about language
```

```
7db123a (tag: v-1.7-rc1) local:added info about engine price!
```

```
b03d1eb local:added info about engine structure!
```

```
73a0bc8 local:added info about engine!
```

```
.....
```

```
a413@mN:~/gitLib/demo-pull-rebase$ git stash list
```

```
stash@{0}: On master: master:modified desc about realtime
```

```
stash@{1}: WIP on master: b854bf3 master:modified desc about connection
```

```
stash@{2}: On master: master:modified desc about databases
```

If we check updated in our text file, we won't see them, because they are not commits, they are stashes.

Our favorite programming language is Golang!

We also love to cooperate with C++!

We talk to each other via gRPC!

We use only HTTP2 connection!

Updates from our 3 stashes are not here. Let's see what we can do with stash.

This is the list of stash commands:

```
=====
git stash                > No local changes to save!
git stash -u             > Stash untracked files!
git stash apply stash@{2} > Execute specific stash!
git stash apply           > Execute the latest stash!
```

```
git stash drop
```

```
git stash drop stash@{2}
git stash pop stash@{2}    > Apply and remove permanently stash!
git stash clear            > Delete all the stashes!
```

=====

Let's apply stash to our repository with index 2:

```
a413@mN:~/gitLib/demo-pull-rebase$ git stash apply stash@{2}
On branch master - Your branch is up to date with 'origin/master'.
```

Changes not staged for commit:

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

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

**modified: programming-language.txt**

no changes added to commit (use "git add" and/or "git commit -a")

=====

What stash apply did? With **apply command** he copied all changes from stash to our repository, but they are not yet on stage. If we want to commit them, before we do that, we must add them on stage.

Let's check now file which we were modified:

```
a413@mN:~/gitLib/demo-pull-rebase$ nano programming-language.txt
```

Our favorite programming language is Golang!

We also love to cooperate with C++!

We talk to each other via gRPC!

We use only HTTP2 connection!

**For databases we use Postgres,Mongo and Redis!**

=====

Now we see that modifications from our stash are applied to our repository, and they are not on stage.

If we check the list of stashes, we will still see them all.

```
a413@mN:~/gitLib/demo-pull-rebase$ git stash list
stash@{0}: On master: master:modified desc about realtime
stash@{1}: WIP on master: b854bf3 master:modified desc about connection
stash@{2}: On master: master:modified desc about databases
```

If we want to remove stash from stash list after applying, we will use git stash with pop extension, like this:

```
git stash pop stash@{2}
```

=====

What if we want to apply another stash? Let's execute another apply command:

```
a413@mN:~/gitLib/demo-pull-rebase$ git stash apply stash@{1}
error: Your local changes to the following files would be overwritten by merge:
  programming-language.txt
Please commit your changes or stash them before you merge – Aborting.
```

Well, now we have conflict, but what is the problem? Git inform us, that one stash is already applied, and if we want to apply another, we can do 3 things:

- > commit changes
- > merge changes
- > remove last updates from last stash

```
a413@mN:~/gitLib/demo-pull-rebase$ git reset
```

Unstaged changes after reset:

M programming-language.txt

```
a413@mN:~/gitLib/demo-pull-rebase$ git status
```

On branch master

Your branch is up to date with 'origin/master'.

Changes not staged for commit:

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

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

**modified: programming-language.txt**

no changes added to commit (use "git add" and/or "git commit -a")

Did we resolve conflict? No, we didn't. Why? Actually, we didn't done anything with git reset, because we didn't change state of repository. Modifications are still there, and they were already be removed from stage. Then what we need to do?

```
a413@mN:~/gitLib/demo-pull-rebase$ nano programming-language.txt
```

Our favorite programming language is Golang!

We also love to cooperate with C++!

We talk to each other via gRPC!

We use only HTTP2 connection!

For databases we use Postgres,Mongo and Redis!

We need to execute git reset --hard, because in that case, we will also remove modifications from our repository:

```
a413@mN:~/gitLib/demo-pull-rebase$ git reset --hard
```

HEAD is now at b854bf3 master:modified desc about connection

```
a413@mN:~/gitLib/demo-pull-rebase$ git status
```

On branch master

Your branch is up to date with 'origin/master'. Nothing to commit, working tree clean!

```
a413@mN:~/gitLib/demo-pull-rebase$ git log --oneline
```

b854bf3 (HEAD -> master, origin/master) master:modified desc about connection

322b3e6 (tag: v-1.8.1) master:modified desc about protocol

4c5ca3d (tag: v-1.8-rc2) master:modified desc about language

7db123a (tag: v-1.7-rc1) local:added info about engine price!

b03d1eb local:added info about engine structure!

73a0bc8 local:added info about engine!

.....

Let's now execute another stash:

```
a413@mN:~/gitLib/demo-pull-rebase$ git stash apply stash@{1}
```

On branch master. Your branch is up to date with 'origin/master'.

Changes not staged for commit:

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

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

**modified: programming-language.txt**

no changes added to commit (use "git add" and/or "git commit -a")

```
a413@mN:~/gitLib/demo-pull-rebase$ nano programming-language.txt
```

Our favorite programming language is Golang!

We also love to cooperate with C++!

We talk to each other via gRPC!

We use only HTTP2 connection!

For every connection we use SSL!

> Update from stash 1!

Now, we see update which is applied from stash with index 1.

Let's now, check again list of stashes. After that remove last updated with reset --hard. Then drop stash with index 2. After that, apply stash with index 0, but with pop command, so that stash would be drop after applying.

```
a413@mN:~/gitLib/demo-pull-rebase$ git stash list
stash@{0}: On master: master:modified desc about realtime
stash@{1}: WIP on master: b854bf3 master:modified desc about connection
stash@{2}: On master: master:modified desc about databases
```

```
a413@mN:~/gitLib/demo-pull-re-base$ git reset --hard
HEAD is now at b854bf3 master:modified desc about connection
```

```
a413@mN:~/gitLib/demo-pull-rebase$ git stash drop stash@{2}
Dropped stash@{2} (cd017d6634407a522ec73013bf8afb6d53a23a0a)
```

```
a413@mN:~/gitLib/demo-pull-rebase$ git stash pop stash@{0}
On branch master. Your branch is up to date with 'origin/master'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   programming-language.txt
```

```
no changes added to commit (use "git add" and/or "git commit -a")
Dropped stash@{0} (64ae4ae2bfee386f02beed38674e8392ab1190ab)
```

If we look on stash list again, we will see again index 0. How? Because indexing is started from last stash that exist on the list. It didn't stay index 1. Index 1 became 0 because, it's the only stash that is left on out list.

```
a413@mN:~/gitLib/demo-pull-rebase$ git stash list
stash@{0}: WIP on master: b854bf3 master:modified desc about connection
```

If we want to create new branch from stash, we can do this:

```
a413@mN:~/gitLib/demo-pull-rebase$ git reset --hard
HEAD is now at b854bf3 master:modified desc about connection

a413@mN:~/gitLib/demo-pull-rebase$ git stash branch feature/task94 stash@{0}
Switched to a new branch 'feature/task94'. On branch feature/task94.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   programming-language.txt
```

```
no changes added to commit (use "git add" and/or "git commit -a")
Dropped stash@{0} (c93389b3d0911b4100af82cd631bd3a02f06b001)
```

What we done here? **Created branch, switch to new branch, transfer stash to new branch, and delete stash after transfer!** After stash, if some file was staged, his status after stash apply, will not be on stage!

```
a413@mN:~/gitLib/demo-pull-rebase$ git branch
* feature/task94
  master
```

```
a413@mN:~/gitLib/demo-pull-rebase$ git add .
a413@mN:~/gitLib/demo-pull-rebase$ git commit -am "feature/task94-connection desc"
[feature/task94 cff6091] feature/task94-connection desc
1 file changed, 1 insertion(+)
```

```
a413@mN:~/gitLib/demo-pull-rebase$ git checkout master
Switched to branch 'master'. Your branch is up to date with 'origin/master'.
```

=====

Let's now merge new branch with master branch, and after that see the status and log of that difference.

```
a413@mN:~/gitLib/demo-pull-rebase$ git merge feature/task94
Updating b854bf3..cff6091
Fast-forward
 programming-language.txt | 1 +
1 file changed, 1 insertion(+)
a413@mN:~/gitLib/demo-pull-rebase$ git status
On branch master. Your branch is ahead of 'origin/master' by 1 commit.
(use "git push" to publish your local commits)
nothing to commit, working tree clean

a413@mN:~/gitLib/demo-pull-rebase$ git log --online
cff6091 (HEAD -> master, feature/task94) feature/task94-connection desc
b854bf3 (origin/master) master:modified desc about connection
322b3e6 (tag: v-1.8.1) master:modified desc about protocol
4c5ca3d (tag: v-1.8-rc2) master:modified desc about language
7db123a (tag: v-1.7-rc1) local:added info about engine price!
b03d1eb local:added info about engine structure!
73a0bc8 local:added info about engine!
.....
```

### Stashing your work

**git stash** temporarily shelves (or stashes) changes you've made to your working copy so you can work on something else, and then come back and re-apply them later on. Stashing is handy if you need to quickly switch context and work on something else, but you're mid-way through a code change and aren't quite ready to commit.

The **git stash** command takes your uncommitted changes (both staged and unstaged), saves them away for later use, and then reverts them from your working copy. At this point you're free to make changes, create new commits, switch branches, and perform any other Git operations; then come back and re-apply your stash when you're ready. Note that the stash is local to your Git repository; stashes are not transferred to the server when you push.

### Workflow of data: UNTRACKED > TRACKED > STAGED!

New file is untracked <??>, tracked is file which is modified but not put on stage <M>, and staged file is modified file which is add to stage, and he is ready for commit <M>!

### Stashing untracked or ignored files

By default, running **git stash** will stash:

- changes that have been added to your index (staged changes)
- changes made to files that are currently tracked by Git (unstaged changes)

But it will **not** stash:

- new files in your working copy that have not yet been staged
- files that have been ignored



## GIT DIFF

Diffing is a function that takes two input data sets and outputs the changes between them. git diff is a multi-use Git command that when executed runs a diff function on Git data sources.

This document will discuss common invocations of git diff and diffing work flow patterns. The git diff command is often used along with git status and git log to analyze the current state of a Git repo.

Reading diffs: outputs  
Raw output format

```
a413@mN:~/gitLib/demo-pull-rebase$ git log --oneline
cff6091 (HEAD -> master, feature/task94) feature/task94-connection desc
b854bf3 (origin/master) master:modified desc about connection
322b3e6 (tag: v-1.8.1) master:modified desc about protocol
4c5ca3d (tag: v-1.8-rc2) master:modified desc about language
.....

a413@mN:~/gitLib/demo-pull-rebase$ git status
On branch master. Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)
nothing to commit, working tree clean

a413@mN:~/gitLib/demo-pull-rebase$ git push origin master

a413@mN:~/gitLib/demo-pull-rebase$ git log --oneline
cff6091 (HEAD -> master, origin/master, feature/task94) feature/task94-connection desc
b854bf3 master:modified desc about connection
322b3e6 (tag: v-1.8.1) master:modified desc about protocol
.....
```

Let's add new line into `programming-language.txt` file and see updated output.

```
a413@mN:~/gitLib/demo-pull-rebase$ echo "New service for Payment!" >> programming-language.txt
a413@mN:~/gitLib/demo-pull-rebase$ cat programming-language.txt
Our favorite programming language is Golang!
We also love to cooperate with C++!
We talk to each other via gRPC!
We use only HTTP2 connection!
For every connection we use SSL!
New service for Payment!
```

```
a413@mN:~/gitLib/demo-pull-rebase$ git status
On branch master. Your branch is up to date with 'origin/master'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   programming-language.txt
no changes added to commit (use "git add" and/or "git commit -a")

a413@mN:~/gitLib/demo-pull-rebase$ git diff --name-only
programming-language.txt
```



```
a413@mN:~/gitLib/demo-pull-rebase$ git diff --shortstat
1 file changed, 1 insertion(+)
a413@mN:~/gitLib/demo-pull-rebase$ git diff --name-status
M      programming-language.txt (M is status Letter!)
```

Possible status letters are:

- A: addition of a file
- C: copy of a file into a new one
- D: deletion of a file
- M: modification of the contents or mode of a file
- R: renaming of a file
- T: change in the type of the file
- U: file is unmerged (you must complete the merge before it can be committed)
- X: "unknown" change type (most probably a bug, please report it)

```
=====
a413@mN:~/gitLib/demo-pull-rebase$ git diff -a > Mark all differences!
```

```
diff --git a/programming-language.txt b/programming-language.txt
index 03a5832..5f00320 100644
```

```
--- a/programming-language.txt
```

```
+++ b/programming-language.txt
```

```
@@ -3,3 +3,4 @@ We also love to cooperate with C++!
```

```
We talk to each other via gRPC!
```

```
We use only HTTP2 connection!
```

```
For every connection we use SSL!
```

```
+New service for Payment!
```

```
a413@mN:~/gitLib/demo-pull-rebase$ git diff --color-words=service > Mark specific pattern!
```

```
diff --git a/programming-language.txt b/programming-language.txt
index 03a5832..5f00320 100644
```

```
--- a/programming-language.txt
```

```
+++ b/programming-language.txt
```

```
@@ -3,3 +3,4 @@ We also love to cooperate with C++!
```

```
We talk to each other via gRPC!
```

```
We use only HTTP2 connection!
```

```
For every connection we use SSL!
```

```
New service for Payment!
```

Let's get list of logs in pretty mod, so we get see ID of specific commit:

```
a413@mN:~/gitLib/demo-pull-rebase$ git log --pretty=oneline
```

```
cff609191e2bf4d3597cc85d2c5f2bcf770dc2ec (HEAD -> master, origin/master, feature/task94) feature/task94-connection desc
```

```
b854bf37c926ec2bdb419c63cfe1371ee448c131 master:modified desc about connection
```

```
322b3e647af5bec32db95226d7db4b270072cb35 (tag: v-1.8.1) master:modified desc about protocol
```

```
4c5ca3d16d487fc2a808588dea15c55a6ef7dc33 (tag: v-1.8-rc2) master:modified desc about language
```

```
7db123a1a2f204f95d7ff4ca3307b94e7719809c (tag: v-1.7-rc1) local:added info about engine price!
```

```
.....
```

### Comparing files between two different commits

**git diff** can be passed Git refs to commits to diff. Some example refs are, HEAD, tags, and branch names. Every commit in Git has a commit ID which you can get when you execute GIT LOG. You can also pass this commit ID to git diff. In comparing we are searching what is the difference in moving from second location to first location.

```
a413@mN:~/gitLib/demo-pull-rebase$ git diff cff609191e2bf4d3597cc85d2c5f2bcf770dc2ec b854bf37c926ec2bdb419c63cfe1371ee448c131
```

```
diff --git a/programming-language.txt b/programming-language.txt
```

```
index 03a5832..cb92e5b 100644
```

```
--- a/programming-language.txt
```

```
+++ b/programming-language.txt
```

@@ -2,4 +2,3 @@ Our favorite programming language is Golang!

We also love to cooperate with C++!

We talk to each other via gRPC!

We use only HTTP2 connection!

**-For every connection we use SSL!**

Here we see that on second location we don't have line "For every connection we use SSL!", while on first location we have. Also we can switch places in git diff command, to see the difference in another way. Also we don't have to put all characters from ID. Six or eight should be fine for git to know, what are you searching for.

```
a413@mN:~/gitLib/demo-pull-rebase$ git diff b854bf cff609
diff --git a/programming-language.txt b/programming-language.txt
index cb92e5b..03a5832 100644
--- a/programming-language.txt
+++ b/programming-language.txt
@@ -2,3 +2,4 @@ Our favorite programming language is Golang!
 We also love to cooperate with C++!
 We talk to each other via gRPC!
 We use only HTTP2 connection!
+For every connection we use SSL!
```

We can also present last commit, without ID, just with typing HEAD, which is the point to last commit:

```
a413@mN:~/gitLib/demo-pull-rebase$ git diff b854bf HEAD
diff --git a/programming-language.txt b/programming-language.txt
index cb92e5b..03a5832 100644
--- a/programming-language.txt
+++ b/programming-language.txt
@@ -2,3 +2,4 @@ Our favorite programming language is Golang!
 We also love to cooperate with C++!
 We talk to each other via gRPC!
 We use only HTTP2 connection!
+For every connection we use SSL!
```

The two dots in this example indicate the diff input is the tips of both branches. The same effect happens if the dots are omitted and a space is used between the branches. Additionally, there is a three dot operator:

**git diff branch1...other-feature-branch**

The three dot operator initiates the diff by changing the first input parameter branch1. It changes branch1 into a ref of the shared common ancestor commit between the two diff inputs, the shared ancestor of branch1 and other-feature-branch. The last parameter input parameter remains unchanged as the tip of other-feature-branch.

### Comparing files from two branches

To compare a specific file across branches, pass in the path of the file as the third argument to git diff:

**git diff master new-branch ./diff\_test.txt**

We can also redirect our differences, as stream into some file, so we can have as log:

```
a413@mN:~/gitLib/demo-pull-rebase$ git diff b854b HEAD >> ~/diff-HEAD-Commit-1.txt
a413@mN:~/gitLib/demo-pull-rebase$ cat ~/diff-HEAD-Commit-1.txt
```

If we need to compare two tags, with `git log --pretty=oneline` command see which tags are connected to which commits, and then compare, those two commits, like you are comparing two tags.

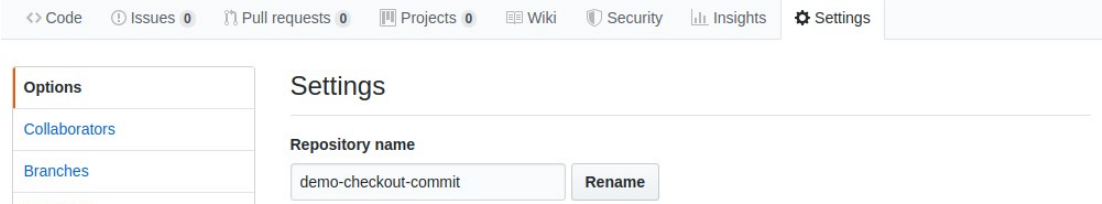
```
a413@mN:~/gitLib/demo-pull-rebase$ git log --pretty=oneline
cfff609191e2bf4d3597cc85d2c5f2bcf770dc2ec (HEAD -> master, origin/master, feature/task94) feature/task94-connection desc
b854bf37c926ec2bdb419c63cfe1371ee448c131 master:modified desc about connection
322b3e647af5bec32db95226d7db4b270072cb35 (tag: v-1.8.1) master:modified desc about protocol
4c5ca3d16d487fc2a808588dea15c55a6ef7dc33 (tag: v-1.8-rc2) master:modified desc about language
```

```
a413@mN:~/gitLib/demo-pull-rebase$ git diff 322b3e 4c5ca3
```

=====

MAKE GITHUB PRIVATE REPOSITORY

=====



=====

On settings tab, scroll down to the danger zone, where you will find “Make this repository private”. When you click on “make private” you will get this message.

Danger Zone

**Make this repository private**  
Hide this repository from the public.

Make private

**Transfer ownership**  
Transfer this repository to another user or to an organization where you have the ability to create repositories.

Transfer

**Archive this repository**  
Mark this repository as archived and read-only.

Archive this repository

**Delete this repository**  
Once you delete a repository, there is no going back. Please be certain.

Delete this repository

Make this repository private

Warning: this is a potentially destructive action.

- You will permanently lose:
  - All stars and watchers of this repository.
  - All pages published from this repository.
- You can upgrade your plan to also avoid losing access to:
  - Codeowners functionality.
  - Any existing wikis.
  - Pulse, Contributors, Community, Traffic, Commits, Code Frequency and Network on the Insights page.
  - Unlimited collaborators. This repository will now be limited to 3 collaborators.
  - Branch protection rules.

Please type in the name of the repository to confirm.

I understand, make this repository private.

**Created By Djurdjevic Nikola**

**Contact:**

**Discord:** Architect413#9400

**Email:** architect@413.world