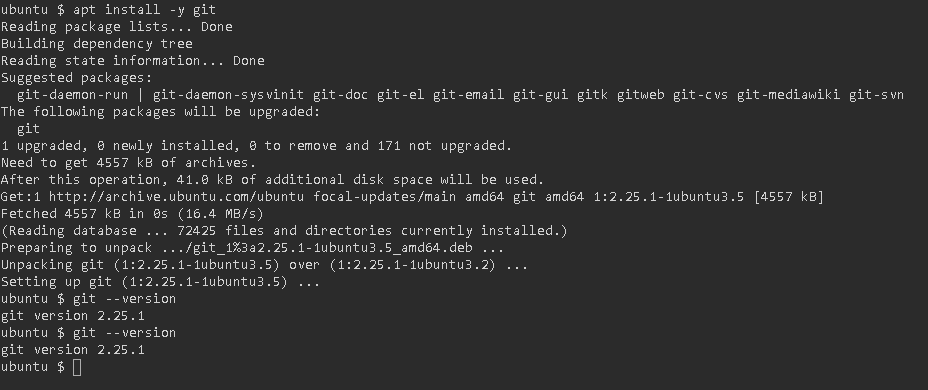
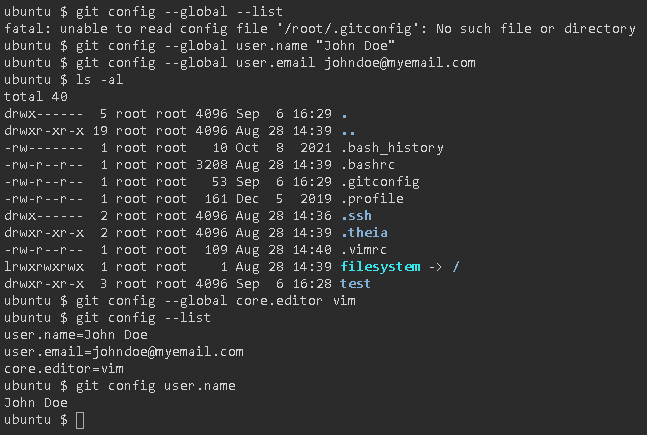
|  |  |
| --- | --- |
| **DOCUMENT RULES:** | |
| **Task Number / Name:** | **Git** |
| **Task name & column name should be written:** | **Bold (CTRL+B)** |
| **Commands should be written in the after # sign:** | *Italic (CTRL+I) #hostname* |
| **Output photo should be cropped or compressed:**  **Photo could be more than one:**  **If you need extra lines, add the line next after it:** | ***Description photo should be with title bar (CTRL + I + B)*** |
| **All other text should be written:** | Standard |
| **Font name and text size:** | Calibri and 9 |
| **Group name:** | Dev\_ops\_ |
| **Student name and surname:** | Murad Abbaszade |
| **E-mail:** | [muradabbaszade6@gmail.com](mailto:muradabbaszade6@gmail.com) |
| **WhatsApp number:** | **+994703664205** |

Install Git on Ubuntu



Configure Git



hooks directory contains all custom hooks. These are small (usually) scripts which have to be executed before commit, or after, before push, etc.

branches - this is deprecated. Don't think about it anymore.

HEAD - pointer to the current branch and its latest commit.

config - configuration file for the repository.

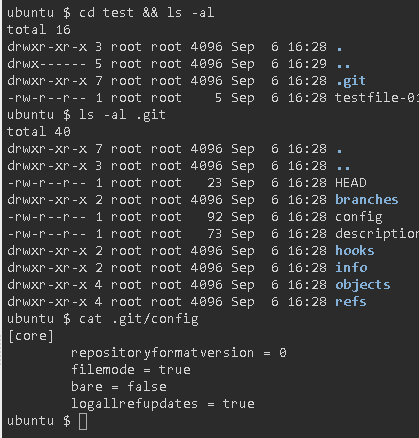
info - the place where you stage the files using git add.

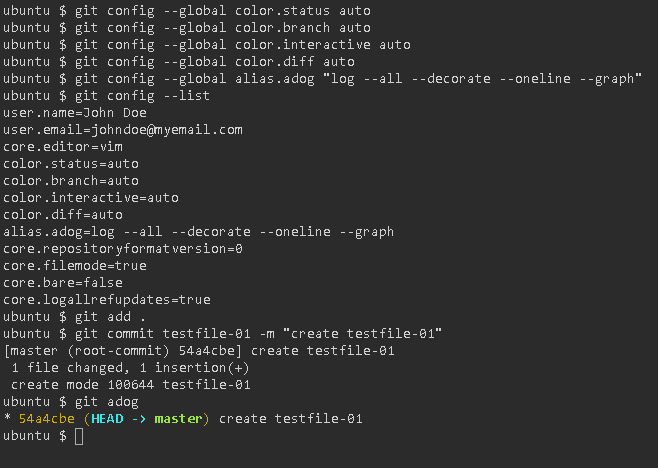
refs - the current state of the whole repo.

objects - commits, trees and blobs are stored here. May be very big.

logs - quite self explanatory.

description - description of the repository.





Commiting the first files

Create a directory

mkdir test-repo

and navigate there

cd test-repo

The directory is obviously empty

ls -al

Now it is time to initialize repository

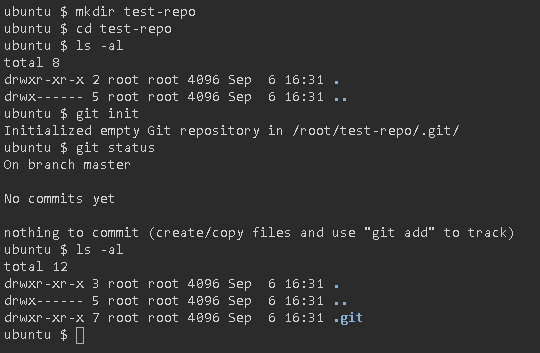
git init

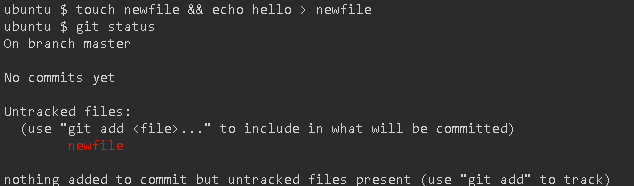
Now you should be able to execute status command

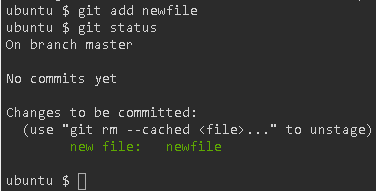
git status

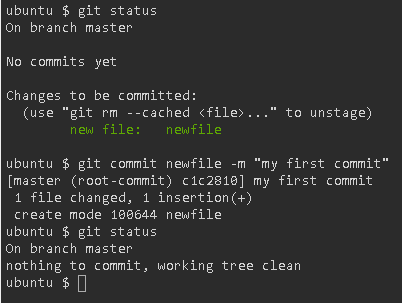
And also the .git directory

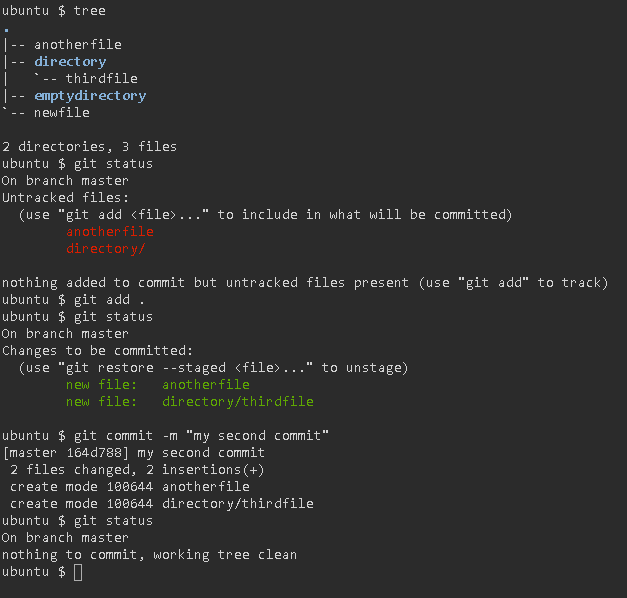
ls -al







****

****

**Remove file from commit**

cd test

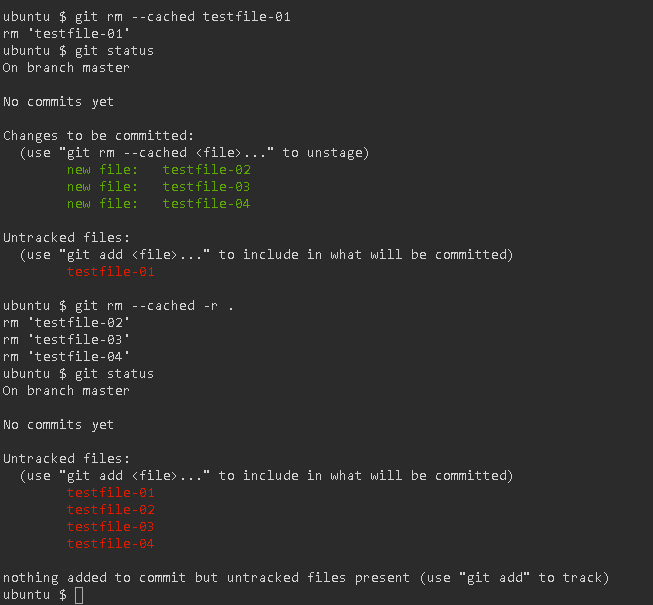
git status

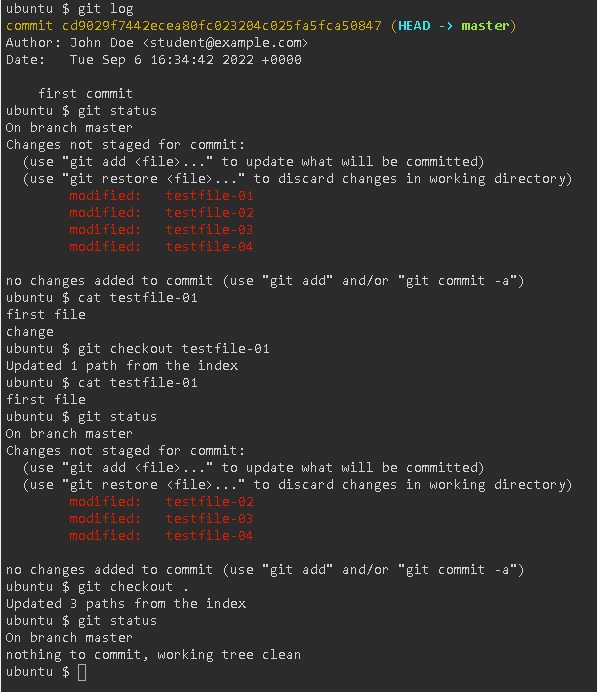
Let's imagine the situation, you want to remove all staged files from index for some reason. Honestly speaking, if you need to do so, please, revise your approach to work :)

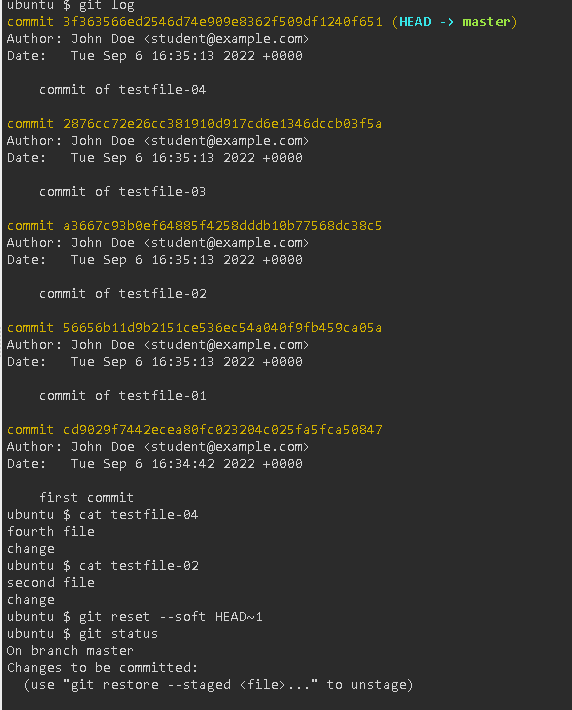
Anyway, we have 4 files in stage. Let's remove testfile-01

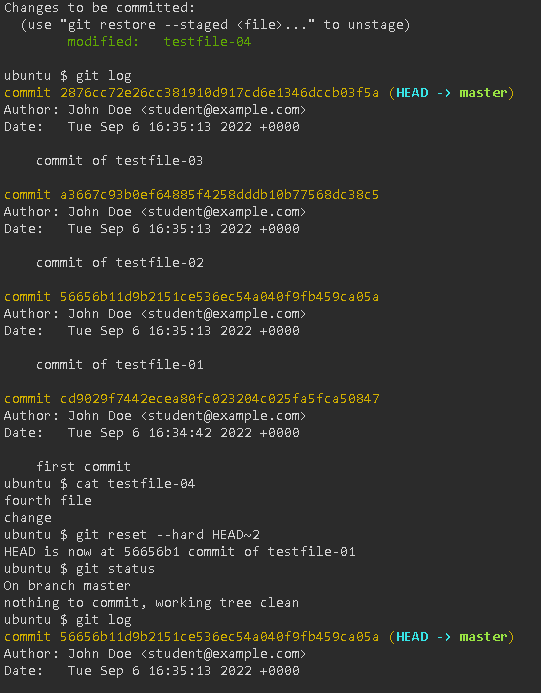
git rm --cached testfile-01

git status

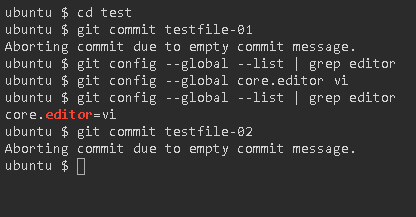
****

****

****

****

**Visual Commit**

****

**Git revert**

cd test

git status

git log

Let's move ourselves back by one commit.

git revert --no-edit HEAD

We succesfully moved back by creating new commit.

git log

By using --no-edit we informed git that we don't want to pass any message and we ask to use default. We will see what git did for us, using git log soon.

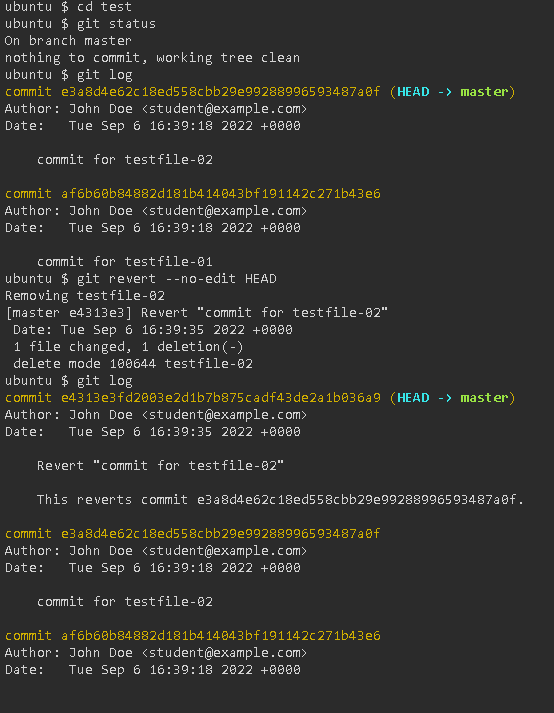
So, now we are back one commit from two already done and... we have three commits. Strange? Please, try to think about the logic behind. We do not want to create problems for other users which can work on the same repo with us. So, we 'go back by going forward'. We keep the history and logic process of work, however we bring back the part of the work from kind of 'backup'.

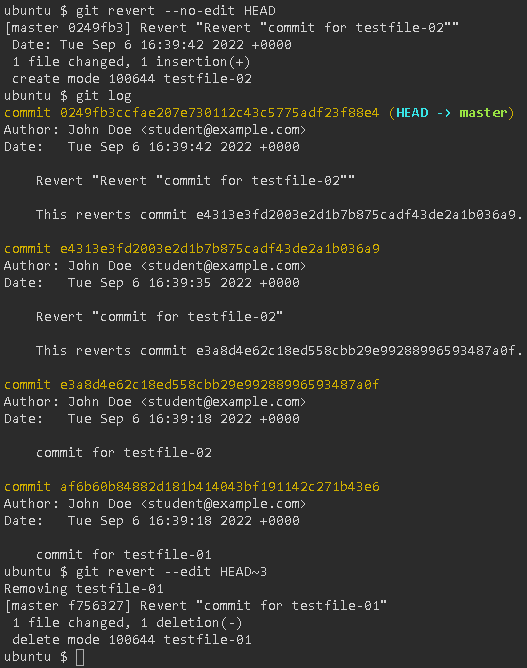
Our current status is like this: we are between two commits, for file-01 and file-02.

Let's have some fun, and revert our revert. Shall we?

git revert --no-edit HEAD

Before you run next commad, please try to imagine what happened in our git :)

****

****

**Check the differences**

cd test

git status

git log

Another command, git diff, allows to check the differences between HEAD and current working directory. Another words, what was changed during our recent work.

git diff

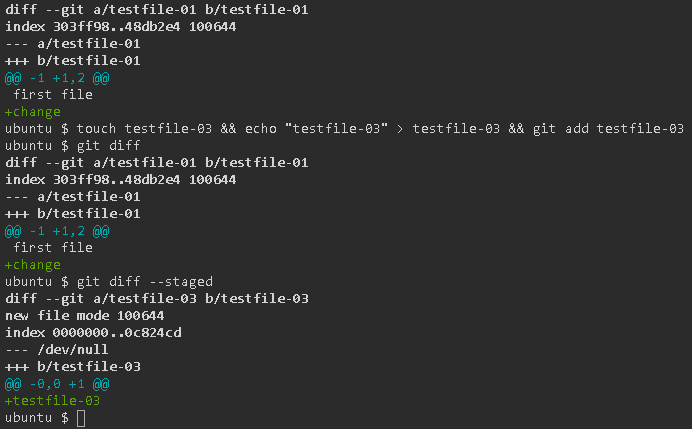
As usual, this command checks HEAD by default. However, we can modify it.

clear && git diff HEAD~1

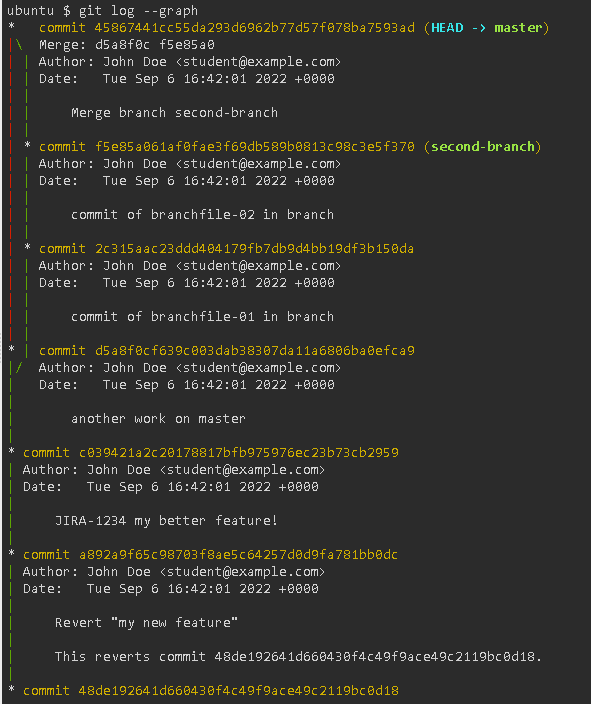
As you can expect, when we go deeper into past, more information is printed. If we want to avoid the mess, we can check diff for one file.

clear && git diff HEAD~1 testfile-01

Now we see information about testfile-02 only in comparision of current working directory and one commit before HEAD.

****

**Detailed information about previous commits**

****

**Gitignore**

Git gives us the possibility to control what will be synchronized with remote and what will be not.

This is realized by file named .gitignore

Let's see what we have. In the background system initialized the repo and do some work. Nothing is staged, nothing is commited.

cd test && ls -al

git status

We don't want to commit

* seconddirectory{{}} directory and its content
* neveringit file

But these are in git status, right?

Let's create gitignore file

touch .gitignore

echo neveringit >> .gitignore

echo seconddirectory >> .gitignore

cat .gitignore

Let'ss check the status again

git status

Ok, we are ready to run

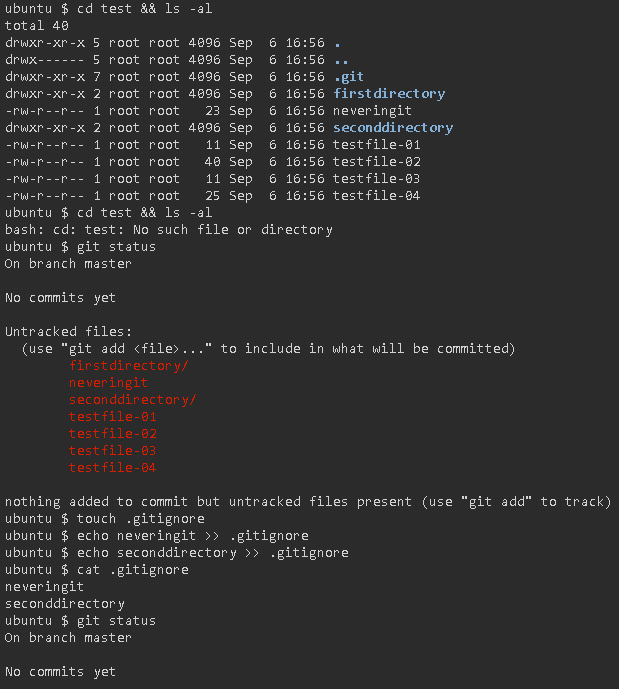
git add .

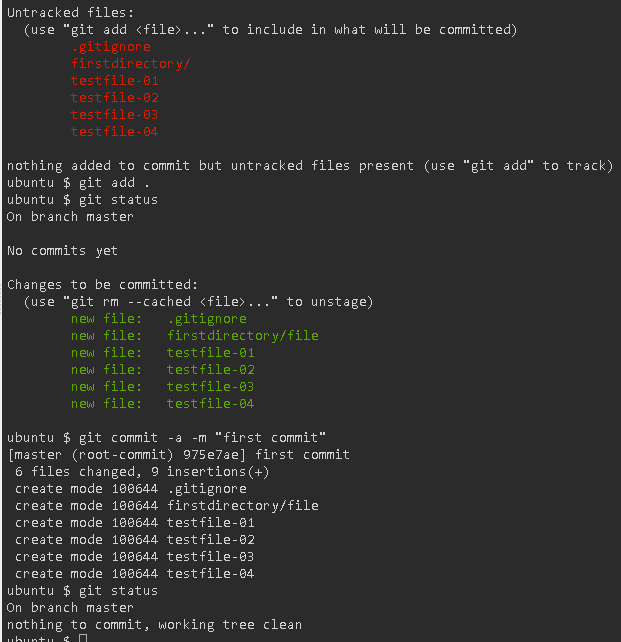
and

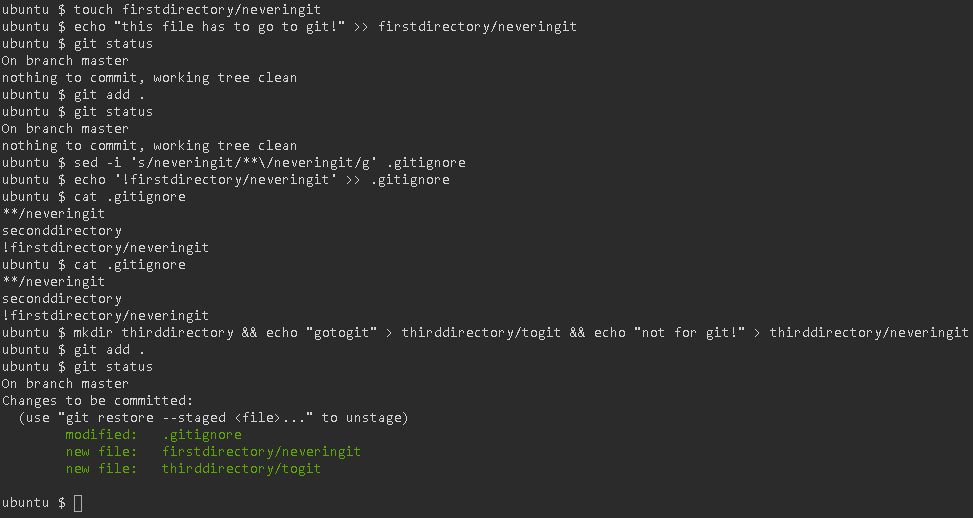
git status

And commit

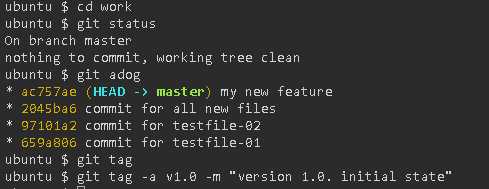
git commit -a -m "first commit"

****

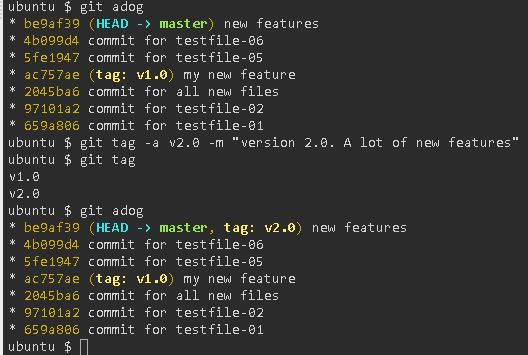
****

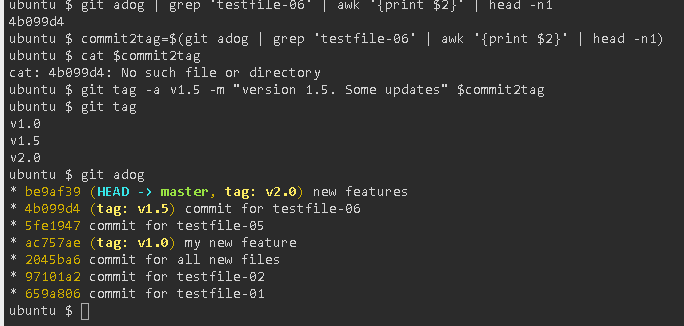
****

**Git tags**

****

****

****

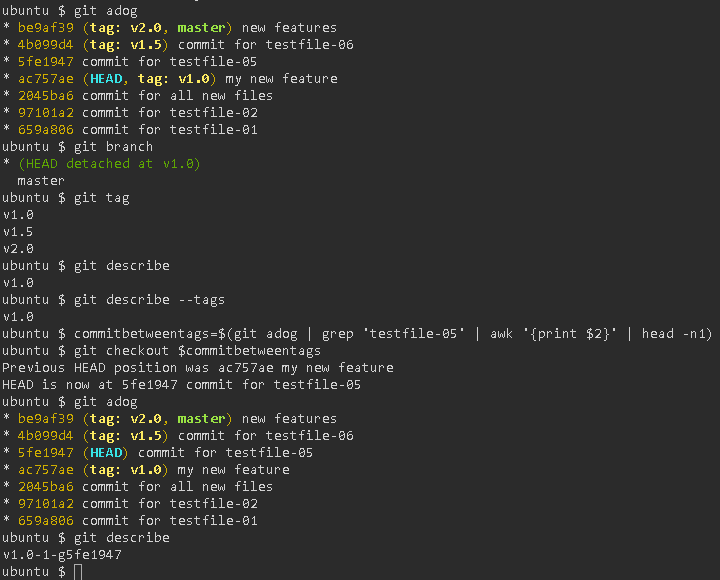
****

git adog - this is obvious by now, I hope.

grep 'testfile-06 - this will select entries with message where this filename occurs (not the best way, but in our case it is more than enough).

awk '{print $2}' - with awk we are 'cutting' the output and print only the third (counted from 0) element, where separator (default one) is a space.

head -n1 - on the end we are printing ony the first element (if there is more records with the same name). git log shows commits by descending through date, so this works for us.

****

Navigation between tags is very easy. We already know the command.

clear && git tag

let's jump to first tag

git checkout v1.0

Ok. We switched. Probably. Let's check.

git adog

Yes, HEAD is on the same commit as tag v1.0.

But...

Where we are on our branch?

git branch

Oh... We are detached from master. Makes sense, we jumped back into past.

Ok. log gives us information where we are. Let's be sure.

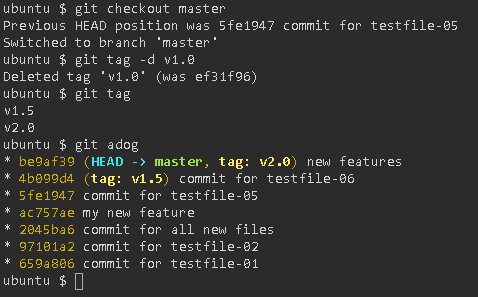
git tag

Hm... Not very helpfull, right?

To check what on what tag you are currently on, use

git describe

**Delete tag**

****

git checkout master

And now we delete v1.0 tag.

git tag -d v1.0

git tag

git adog

Yep, tag is deleted.

Git merge

cd merge

The system prepared a few commits in the background. Let's see them.

git adog

Now we can check what branches we have and also on which branch we currently are.

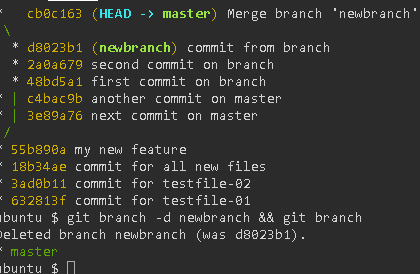
git branch

Our current branch is pointed by asterisk (and color if enabled).

Ok, we are ready to create and swith to just created branch. We are lazy, so let's do it with one line command.

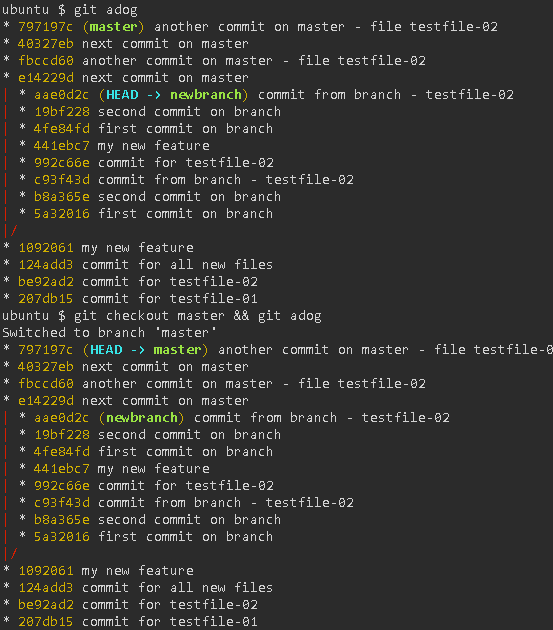
git checkout -b newbranch

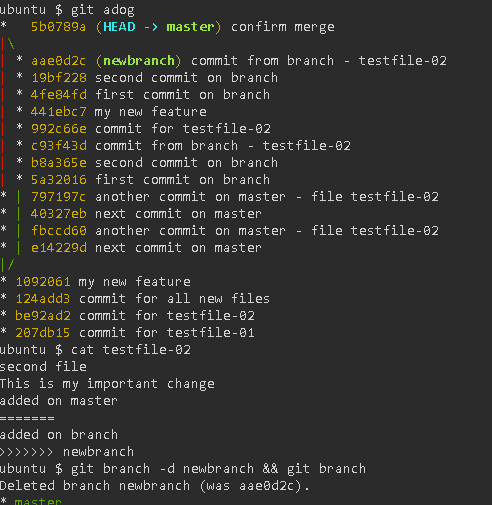
git checkout moves us between branches. When we add -b, a new branch is created.

****

As we finished work on branch, we can delete it

git branch -d newbranch && git branch





cd ~/merge2

git branch

git adog

git checkout master && git adog

We are ready to merge.

git merge newbranch

Ouch.

What happened?

Well, testfile-02 was updated that way, that different changes are done in the same place.

We need to fix the conflict.

Let's take it easy now. We remove only these things which are pointing the conflict.

cat testfile-02

sed -i '8d' testfile-02 && sed -i '5d' testfile-02 && sed -i '3d' testfile-02 && cat testfile-02

Ok.

git status says, we need to confirm fixes by adding the file to stage again.

git add testfile-02

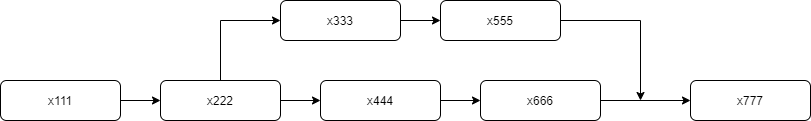
Now we are ready to finalize merge

git commit -a -m "confirm merge"

Git rebase

merge vs rebase

We already knows what merge do. Let's refresh our memory by this small picture. (numbers represent commit hashes)



It is quite simple. We create branch, do changes here and there and then merge. Work history should show exactly this. Let's check.

System creates two repos. One in merge directory, where all process was completed (final merge). Let's see how our history looks.

cd merge

git adog

(do you remember our "a dog" config setting from git config module?)

