Check your system

First, let's see if we have Git installed. We can do it on different ways

dpkg -l | grep git

The ii in the list means (if there are packages installed, you should see this mark) that the package is correctly installed and available.

apt list git -a

apt-cache pkgnames git

or... Just try run git command :D

As you can see, Git is installed, but nevertheless, we'll try to install it again

# install Git

In order to install Git, you can use multiple ways, we will use apt command.

apt install -y git

In case if package was installed in the same version, you wil lse notification about it. Sometimes system will communicate the posibility of purging the obsolete files. But now we can skip it.

# Did we install Git successfully?

Let's check, by running

git --version

This command will show you version of installed package.

ubuntu $ dpkg -l | grep git

ii git 1:2.25.1-1ubuntu3.2 amd64 fast, scalable, distributed revision control system

ii git-man 1:2.25.1-1ubuntu3.2 all fast, scalable, distributed revision control system (manual pages)

ii libplymouth5:amd64 0.9.4git20200323-0ubuntu6.2 amd64 graphical boot animation and logger - shared libraries

ii librtmp1:amd64 2.4+20151223.gitfa8646d.1-2build1 amd64 toolkit for RTMP streams (shared library)

ii plymouth 0.9.4git20200323-0ubuntu6.2 amd64 boot animation, logger and I/O multiplexer

ii plymouth-theme-ubuntu-text 0.9.4git20200323-0ubuntu6.2 amd64 boot animation, logger and I/O multiplexer - ubuntu text theme

ubuntu $ apt list git -a

Listing... Done

git/focal-updates,focal-security 1:2.25.1-1ubuntu3.5 amd64 [upgradable from: 1:2.25.1-1ubuntu3.2]

git/now 1:2.25.1-1ubuntu3.2 amd64 [installed,upgradable to: 1:2.25.1-1ubuntu3.5]

git/focal 1:2.25.1-1ubuntu3 amd64

ubuntu $ apt-cache pkgnames git

git-phab

gitsome

git-ftp

git-gui

gitlab-shell

git-reintegrate

git-merge-changelog

git2cl

git-review

gitano

git-revise

git-sizer

git-daemon-run

git-lfs

gitbatch

git-man

gitmagic

git-extras

git-remote-gcrypt

git-restore-mtime

github-backup

gitit

gitbrute

gitso

git-svn

git

git-annex-remote-rclone

gitlab-cli

git-daemon-sysvinit

git-publish

git-buildpackage

gitpkg

gitg

gitk

git-build-recipe

gitlab-runner

gitlab-workhorse

gitweb

git-mediawiki

gitinspector

git-quick-stats

git-repair

git-crecord

git-email

git-secret

git-buildpackage-rpm

git-annex

git-crypt

git-debrebase

gitless

gitolite3

gitlabracadabra

git-debpush

git-all

git-cola

gitlint

git-secrets

git-cvs

git-doc

git-dpm

git-flow

git-el

ubuntu $ apt install -y git

Reading package lists... Done

Building dependency tree

Reading state information... Done

Suggested packages:

git-daemon-run | git-daemon-sysvinit git-doc git-el git-email git-gui gitk gitweb git-cvs git-mediawiki git-svn

The following packages will be upgraded:

git

1 upgraded, 0 newly installed, 0 to remove and 165 not upgraded.

Need to get 4557 kB of archives.

After this operation, 41.0 kB of additional disk space will be used.

Get:1 http://archive.ubuntu.com/ubuntu focal-updates/main amd64 git amd64 1:2.25.1-1ubuntu3.5 [4557 kB]

Fetched 4557 kB in 1s (4254 kB/s)

(Reading database ... 72425 files and directories currently installed.)

Preparing to unpack .../git\_1%3a2.25.1-1ubuntu3.5\_amd64.deb ...

Unpacking git (1:2.25.1-1ubuntu3.5) over (1:2.25.1-1ubuntu3.2) ...

Setting up git (1:2.25.1-1ubuntu3.5) ...

ubuntu $ git --version

git version 2.25.1

# Configure git

# Config's must have

let's run

git config --global --list

hm... Don't be surprised if system returned error.

Ok, let's configure the must have things.

At this moment we are working on global table, which means we are setting global configuration, relevant everywhere **for this user**, until overwritten.

So:

--system - table relevant for the whole machine

--global - for the current user

--local (default) for the current repository

Ok, let's make some entries.

git config --global user.name "John Doe"

git config --global user.email johndoe@myemail.com

We created our "personality". Of course it can be overwritten with the --local in any of repos.

And now the file .gitconfig in the root directory of our user exist: ls -al

Let's add one more important thing.

git config --global core.editor vim

Let's see what we have in our config.

git config --list -a ll settings are printed.

git config --list --global - only --global table is listed

git config user.name - selected record is printed.

# Config of the repository

In the background system created a directory and initialized git repo. Let's list the content.

cd test && ls -al

There is a .git directory, which is a hidden one (. on the beginning). This directory is created during git init operation.

What is inside?

ls -al .git

So, what is what? Let's explain the important files /directories.

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.

Ok, so, what we have in config file?

cat .git/config

For now it doesn't contains much. But when you start to work with branches, remote, etc, more lines will appear there.

# Nice to have things

Let's come back to our configuration.

Maybe we will add some colors?

git config --global color.status auto

git config --global color.branch auto

git config --global color.interactive auto

git config --global color.diff auto

git config --global alias.adog "log --all --decorate --oneline --graph"

Explanation. A dog is very popular way to remember the most useful set of parameters for git log.

git config --list

We have a file to commit in the repo. We can do it now, as we configured our user.

git add .

git commit testfile-01 -m "create testfile-01"

And now it is time to execute our alias.

git adog

These are only simple example. You can go crazy here.

ubuntu $ git config --global --list

fatal: unable to read config file '/root/.gitconfig': No such file or directory

ubuntu $ git config --global user.name "John Doe"

ubuntu $ git config --global user.email johndoe@myemail.com

ubuntu $ git config --global core.editor vim

ubuntu $ git config --list

user.name=John Doe

user.email=johndoe@myemail.com

core.editor=vim

ubuntu $ git config --list --global

user.name=John Doe

user.email=johndoe@myemail.com

core.editor=vim

ubuntu $ git config user.name

John Doe

ubuntu $ cd test && ls -al

total 16

drwxr-xr-x 3 root root 4096 Sep 5 10:19 .

drwx------ 5 root root 4096 Sep 5 10:20 ..

drwxr-xr-x 7 root root 4096 Sep 5 10:19 .git

-rw-r--r-- 1 root root 5 Sep 5 10:19 testfile-01

ubuntu $ ls -al .git

total 40

drwxr-xr-x 7 root root 4096 Sep 5 10:19 .

drwxr-xr-x 3 root root 4096 Sep 5 10:19 ..

-rw-r--r-- 1 root root 23 Sep 5 10:19 HEAD

drwxr-xr-x 2 root root 4096 Sep 5 10:19 branches

-rw-r--r-- 1 root root 92 Sep 5 10:19 config

-rw-r--r-- 1 root root 73 Sep 5 10:19 description

drwxr-xr-x 2 root root 4096 Sep 5 10:19 hooks

drwxr-xr-x 2 root root 4096 Sep 5 10:19 info

drwxr-xr-x 4 root root 4096 Sep 5 10:19 objects

drwxr-xr-x 4 root root 4096 Sep 5 10:19 refs

ubuntu $ cat .git/config

[core]

repositoryformatversion = 0

filemode = true

bare = false

logallrefupdates = true

ubuntu $ git config --global color.status auto

ubuntu $ git config --global color.branch auto

ubuntu $ git config --global color.interactive auto

ubuntu $ git config --global color.diff auto

ubuntu $ git config --global alias.adog "log --all --decorate --oneline --graph"

ubuntu $ git config --list

user.name=John Doe

user.email=johndoe@myemail.com

core.editor=vim

color.status=auto

color.branch=auto

color.interactive=auto

color.diff=auto

alias.adog=log --all --decorate --oneline --graph

core.repositoryformatversion=0

core.filemode=true

core.bare=false

core.logallrefupdates=true

ubuntu $ git add .

ubuntu $ git commit testfile-01 -m "create testfile-01"

[master (root-commit) e2a1b49] create testfile-01

1 file changed, 1 insertion(+)

create mode 100644 testfile-01

ubuntu $ git adog

\* e2a1b49 (HEAD -> master) create testfile-01

ubuntu $

# Lesson 3: Commiting the first files

# nitialize the repository

If you start work on the new repository locally (what means repo wasn't cloned earlier), you need to initialize the repository and build all internal structure needed to work with version control. In the nutshell, it will create the .git directory and all content inside (this will be part of another scenario)

## **Step 1**

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

# Commit first file

We are already in proper directory, and git was initialized. Now it is the time to create first file.

touch newfile && echo hello > newfile

Let's check, if git sees the file

git status

In the output you should see that our new file is untracked.

# add file to staging area

It is the time to inform git, that we wish to stage our changes. to do so, execute command

git add newfile

this command added the file to the stage area, what can be seen after

git status

You see that the file is tracked and ready to be commited.

It is important to know, if you stage the file once, you don't need to repeat this action again during the work.

# Commit the file

Now it is the time to commit the file.

Commit is a proces to create a revision of the file. A version which will be stored in Version Control.

Let's be sure that our file is staged

git status

Yes, it is, so let's commit it!

git commit newfile -m "my first commit"

By adding -m <comment> we describe what this commit is about.

We commited a specific file here, a newfile. By using -a instead, we are able to commit all files from repository. By using . we will commit changes from current path recursively.

It is not the place here to explain the best practices how to create commit messages, though.

git status

Now our file is commited and its version is created. Status command shows empty list.

Congratulations, you commited your first file!

Add and commit multiple files

Scenario created some files for you. What was created?

* two directories
* one file in root directory of our repo
* one file in directory
* please notice, one directory is empty

tree

So, how our git status looks?

git status

What we have here?

First, we see one file is ready to be staged. One directory also.

But where are the missing pieces?

file inside the directory is not visibe, as the directory itself is not tracked yet. But Git is aware, that this directory is not empty, therefore we see it as ready to be added.

Second directory is empty, Git knows that, and it decides that adding the directory at this moment is not needed.

Lets add all files in one simple command. Last time we added specific file with git add command. This time we will use

git add .

Let's check

git status

Please, notice the difference. Now, your file is listed as listed to be commited. Empty directory is ommited.

Please be aware of one thing. git add . will add all files **from your current directory and subdirectories, not from repository's root**. If you are in subdirectory, only files from this path will be added.

Now we are ready to commit.

git commit -m "my second commit"

Let's check

git status

You succesfully commited multiple files!

ubuntu $ mkdir test-repo

ubuntu $ cd test-repo

ubuntu $ ls -al

total 8

drwxr-xr-x 2 root root 4096 Sep 5 10:21 .

drwx------ 5 root root 4096 Sep 5 10:21 ..

ubuntu $ git init

Initialized empty Git repository in /root/test-repo/.git/

ubuntu $ git status

On branch master

No commits yet

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

ubuntu $ ls -al

total 12

drwxr-xr-x 3 root root 4096 Sep 5 10:21 .

drwx------ 5 root root 4096 Sep 5 10:21 ..

drwxr-xr-x 7 root root 4096 Sep 5 10:21 .git

ubuntu $ touch newfile && echo hello > newfile

ubuntu $ git status

On branch master

No commits yet

Untracked files:

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

newfile

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

ubuntu $ git add newfile

ubuntu $ git status

On branch master

No commits yet

Changes to be committed:

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

new file: newfile

ubuntu $ git status

On branch master

No commits yet

Changes to be committed:

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

new file: newfile

ubuntu $ git commit newfile -m "my first commit"

[master (root-commit) cd636e2] my first commit

1 file changed, 1 insertion(+)

create mode 100644 newfile

ubuntu $ git status

On branch master

nothing to commit, working tree clean

ubuntu $ tree

.

|-- anotherfile

|-- directory

| `-- thirdfile

|-- emptydirectory

`-- newfile

2 directories, 3 files

ubuntu $ git status

On branch master

Untracked files:

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

anotherfile

directory/

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

ubuntu $ git add .

ubuntu $ git status

On branch master

Changes to be committed:

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

new file: anotherfile

new file: directory/thirdfile

ubuntu $ git commit -m "my second commit"

[master e0a4b92] my second commit

2 files changed, 2 insertions(+)

create mode 100644 anotherfile

create mode 100644 directory/thirdfile

ubuntu $ git status

On branch master

nothing to commit, working tree clean

# Lesson 4: Remove file from commit

# Remove files from stage

Let's check, what we have in stage.

But before we do it, we need to navigate to repo's directory.

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

Great! We removed one file from stage. What about three others?

Yes, we can do the same command like before and change the name of the file in the command. But what if we have 10 files? 100?

git rm --cached -r .

please notice, we used . to say everything from here and -r which means recursive.

Our stage should be empty and all files ready for being staged.

git status

Yes, our assumption was correct!

NEXT

# Restore previous state of the file

Our files were staged again and even commited in the background.

Let's see our commit

git log

About git log we will talk in another scenario.

Also, you can see some modifications were done in files after previous commit.

git status

We have all 4 files ready to be commited, but unfortunately, we don't want those changes.

First, let's check what is in the file testfile-01

cat testfile-01

git checkout testfile-01

cat testfile-01

git status

We succesfully reset the file to the state from previous commit, using git checkout. In this way we do the checkout of the last indexed state of this file on current branch.

In similar way like in some scenarios before, we can remove all changes in one short command. Let's do it.

git checkout .

git status

All changes are reverted and our directory in consistent with previous commit.

# Reset the current HEAD to the selected state

git reset moves the current pointer in HEAD and branch refs to specific state. git checkout does similar thing, but in fact, both of these commands operates on different levels. This theory will be explained later.

Reset has three main ways of operating, but we will touch only two of them. This time --mixed is not covered.

## **--soft**

git log shows that we have many changes done. We have one commit for each file.

git log

Let't reset the HEAD to the state before commiting the last file.

But before, let's see what is inside the files

cat testfile-04

cat testfile-02

git reset --soft HEAD~1

Now let's see what we have.

git status

git log

cat testfile-04

With --soft parameter we came back to the previous HEAD of the repository, but all changes which we commited are unchanged.

## **--hard**

Now let's try something more powerful.

git reset --hard HEAD~2

And let's look what happened

git status

git log

cat testfile-04

cat testfile-02

What we did?

We came back two more commits (~HEAD~2) and we said, this time we want to not only move back, but also we want to remove all changes which were done.

BACKNEXT

ubuntu $ cd test

ubuntu $ git status

On branch master

No commits yet

Changes to be committed:

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

new file: testfile-01

new file: testfile-02

new file: testfile-03

new file: testfile-04

ubuntu $ git rm --cached testfile-01

rm 'testfile-01'

ubuntu $ git status

On branch master

No commits yet

Changes to be committed:

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

new file: testfile-02

new file: testfile-03

new file: testfile-04

Untracked files:

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

testfile-01

ubuntu $ git rm --cached -r .

rm 'testfile-02'

rm 'testfile-03'

rm 'testfile-04'

ubuntu $ git status

On branch master

No commits yet

Untracked files:

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

testfile-01

testfile-02

testfile-03

testfile-04

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

ubuntu $ git log

commit 19e75c58caad76336e4894e6721a858f92ed2581 (HEAD -> master)

Author: John Doe <student@example.com>

Date: Mon Sep 5 10:24:09 2022 +0000

first commit

ubuntu $ 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: testfile-01

modified: testfile-02

modified: testfile-03

modified: testfile-04

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

ubuntu $ cat testfile-01

first file

change

ubuntu $ git checkout testfile-01

Updated 1 path from the index

ubuntu $ cat testfile-01

first file

ubuntu $ 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: testfile-02

modified: testfile-03

modified: testfile-04

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

ubuntu $ git checkout .

Updated 3 paths from the index

ubuntu $ git status

On branch master

nothing to commit, working tree clean

ubuntu $ git log

commit 81c5703f6d505c5fbc767a2b6653259efcf1cedc (HEAD -> master)

Author: John Doe <student@example.com>

Date: Mon Sep 5 10:24:31 2022 +0000

commit of testfile-04

commit 5c363081beec7c15325cc03be3601a85914dc662

Author: John Doe <student@example.com>

Date: Mon Sep 5 10:24:31 2022 +0000

commit of testfile-03

commit c27a4f03c47a017ffff9d5a6ae3535019c07f938

Author: John Doe <student@example.com>

Date: Mon Sep 5 10:24:31 2022 +0000

commit of testfile-02

commit 4ebbf664ba79773bb781afe39c019966762466b9

Author: John Doe <student@example.com>

Date: Mon Sep 5 10:24:31 2022 +0000

commit of testfile-01

commit 19e75c58caad76336e4894e6721a858f92ed2581

Author: John Doe <student@example.com>

Date: Mon Sep 5 10:24:09 2022 +0000

first commit

ubuntu $ cat testfile-04

fourth file

change

ubuntu $ cat testfile-02

second file

change

ubuntu $ git reset --soft HEAD~1

ubuntu $ git status

On branch master

Changes to be committed:

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

modified: testfile-04

ubuntu $ git log

commit 5c363081beec7c15325cc03be3601a85914dc662 (HEAD -> master)

Author: John Doe <student@example.com>

Date: Mon Sep 5 10:24:31 2022 +0000

commit of testfile-03

commit c27a4f03c47a017ffff9d5a6ae3535019c07f938

Author: John Doe <student@example.com>

Date: Mon Sep 5 10:24:31 2022 +0000

commit of testfile-02

commit 4ebbf664ba79773bb781afe39c019966762466b9

Author: John Doe <student@example.com>

Date: Mon Sep 5 10:24:31 2022 +0000

commit of testfile-01

commit 19e75c58caad76336e4894e6721a858f92ed2581

Author: John Doe <student@example.com>

Date: Mon Sep 5 10:24:09 2022 +0000

first commit

ubuntu $ cat testfile-04

fourth file

change

ubuntu $ git status

On branch master

Changes to be committed:

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

modified: testfile-04

ubuntu $ git reset --hard HEAD~2

HEAD is now at 4ebbf66 commit of testfile-01

ubuntu $ git status

On branch master

nothing to commit, working tree clean

ubuntu $ git log

commit 4ebbf664ba79773bb781afe39c019966762466b9 (HEAD -> master)

Author: John Doe <student@example.com>

Date: Mon Sep 5 10:24:31 2022 +0000

commit of testfile-01

commit 19e75c58caad76336e4894e6721a858f92ed2581

Author: John Doe <student@example.com>

Date: Mon Sep 5 10:24:09 2022 +0000

first commit

ubuntu $ cat testfile-02

second file

ubuntu $

# Lesson 5: Visual commit

# Text editor for create message for commit

Until now we add messages to commits using -m parameter. Let's do something different. We have repo prepared and some files are in stage, so let's commit one of them

cd test

git commit testfile-01

As you see, the default editor was executed and we can add commit using it. Feel free to add message and commit, or exit.

## **Default editor**

Let's check if we have anything configured

git config --global --list | grep editor

No, we do not have anything configured.

What we've done? We asked git to show us the whole configuration in global space. We can use system and local too. Local obviously means the configuration of this specific repo. Also, we grep the output to find information about editor only.

Let's change it to the best editor ever - vi!

git config --global core.editor vi

Check the config again

git config --global --list | grep editor

All looks good, so now it is the time to commit our next file

git commit testfile-02

This time we used vi!

Ah, vi ;P Do you wish to quit the editor? Click ESC key and then write :q!

# Lesson 6: Revert changes

# Git revert

We have our repo prepared, two files created and two commits done. Let's see how it looks.

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 :)

git log

Of course, we can revert more changes than the last commit.

We can revert multiple commits by

git revert --edit HEAD~3

This time we reverted to the beginning of our history. And we forced git to give us possibility to add message (by the way, --edit is default behavior).

So, to make sure we all are on the ame page. HEAD is the current 'place' where we are. For now, let's think about it as combination of current branch and current place in git history.

So, with HEAD~3 we said: I want to move back from HEAD by 3 commits . But that's not all. This place becomes our new HEAD .

We can revert to specific commit hash too. more about hashes in git log scenario.

**Important** Very often you can experience problems when revert more than one commit and the same file is involed. We will talk about troubleshooting in future lesson.

ubuntu $ cd test

ubuntu $ git status

On branch master

nothing to commit, working tree clean

ubuntu $ git log

commit d0fbe1e8d1cbd9852dfba29e60c3f6702208078e (HEAD -> master)

Author: John Doe <student@example.com>

Date: Mon Sep 5 10:27:31 2022 +0000

commit for testfile-02

commit 1d39d421b85bedff3fcf14af4e1edf38acd6e3bc

Author: John Doe <student@example.com>

Date: Mon Sep 5 10:27:31 2022 +0000

commit for testfile-01

ubuntu $ git revert --no-edit HEAD

Removing testfile-02

[master e69a5d4] Revert "commit for testfile-02"

Date: Mon Sep 5 10:28:33 2022 +0000

1 file changed, 1 deletion(-)

delete mode 100644 testfile-02

ubuntu $ git log

commit e69a5d48114b25d9529d5a285dbbd98c2f0e9323 (HEAD -> master)

Author: John Doe <student@example.com>

Date: Mon Sep 5 10:28:33 2022 +0000

Revert "commit for testfile-02"

This reverts commit d0fbe1e8d1cbd9852dfba29e60c3f6702208078e.

commit d0fbe1e8d1cbd9852dfba29e60c3f6702208078e

Author: John Doe <student@example.com>

Date: Mon Sep 5 10:27:31 2022 +0000

commit for testfile-02

commit 1d39d421b85bedff3fcf14af4e1edf38acd6e3bc

Author: John Doe <student@example.com>

Date: Mon Sep 5 10:27:31 2022 +0000

commit for testfile-01

ubuntu $ git revert --no-edit HEAD

[master 950e27b] Revert "Revert "commit for testfile-02""

Date: Mon Sep 5 10:28:38 2022 +0000

1 file changed, 1 insertion(+)

create mode 100644 testfile-02

ubuntu $ git log

commit 950e27b2d9a4731af8e755c8711242017e8b71cd (HEAD -> master)

Author: John Doe <student@example.com>

Date: Mon Sep 5 10:28:38 2022 +0000

Revert "Revert "commit for testfile-02""

This reverts commit e69a5d48114b25d9529d5a285dbbd98c2f0e9323.

commit e69a5d48114b25d9529d5a285dbbd98c2f0e9323

Author: John Doe <student@example.com>

Date: Mon Sep 5 10:28:33 2022 +0000

Revert "commit for testfile-02"

This reverts commit d0fbe1e8d1cbd9852dfba29e60c3f6702208078e.

commit d0fbe1e8d1cbd9852dfba29e60c3f6702208078e

Author: John Doe <student@example.com>

Date: Mon Sep 5 10:27:31 2022 +0000

commit for testfile-02

commit 1d39d421b85bedff3fcf14af4e1edf38acd6e3bc

Author: John Doe <student@example.com>

Date: Mon Sep 5 10:27:31 2022 +0000

commit for testfile-01

ubuntu $ git revert --edit HEAD~3

Removing testfile-01

[master e10f534] Revert "commit for testfile-01"

1 file changed, 1 deletion(-)

delete mode 100644 testfile-01

ubuntu $

# Lesson 7: Check the differences

# Check the changes

We have repo prepared. changes are done and commited. And one file is updated again.

Before we start, we have to change the work directory.

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.

# Staged differences

We already learned that git diff shows differences between commit and working directory. So, what about staged changes?

Let's test

touch testfile-03 && echo "testfile-03" > testfile-03 && git add testfile-03

We created a new file and added it to staging. now let's see the difference between the current working directory and HEAD.

git diff

Well, there is nothing about our new file.

In order to see the difference between staged work and HEAD, you need to say this explicitly.

git diff --staged

# Git show

Another useful commad is git show. This will give you information similar to git log.

git show work by default against HEAD. Of course you can use it to any commit and any file.

git show

# Lesson 8: Detailed information about previous commits

# Formatting information from git log

Right now we know how git log works. Well, it is ok, but is there any way to extract some data or format messages? Or even query them?

Yes!

**Remember to quit git log with q. If logs are longer than your screen, it always goes to interactive mode.**

We have repo prepared. Some work was done. Even one revert was executed.

git log shows it pretty well. But imagine, how it will look with thousands of commits?

## **Best for programmatic access**

Sometimes you want to check commits through scripts. In this case git log itself is unhandy. What to do to have it better formatted for scripts?

cd test && clear && git log --oneline

--oneline shows only most important info about commits. You have only hash and commit message.

Is this enough? We can argue. But definitely is more clear for high level understanding of previous work. How to make it better we will see soon.

Let's clear our screen

clear

Do you remember the

git show command?

It shows more details about last (or selected) commit.

But hey! We can do this for all commits!

clear && git log -p

Please remember to use q key to escape and come back to shell.

Some statistics' folks wish to know what amount of work was done in the commit. Git gives this possibility

clear && git log --stat

Now we can clearly see how many lines were added or removed in each commit.

Maybe you wish to see the information sorted by author of the commits?

clear && git shortlog is the answer!

# Prettier output?

Let's clear the screen first.

Ok, we know how to do some basics tricks. But maybe we want to visualize some work with branches, etc?

git log --graph

You can clearly see that in some moment we created another branch, we did some work there and then we merged it (branches will be described soon)

But... It doesn't look good, doesn't it?

Git gives us the possibility to connect multiple parameters in one command.

clear && git log --oneline --graph

# Formatting

Git allows us to use special formatting to print exactly what we want.

Sometimes your log is very plain. If this is your case and you would like to have more colorful output, use --decorate parameter.

A few examples which we combine into a working command:

%Cblue - switch output text to blue

%Creset - resets the color to default

%H - commit hash. Preferable is to use short hash with %h.

%an - who commited the changes

%ad - date of commit. For scripting purposes you can use UNIX timestamp - %at.

%s - commit message

How to format our commad to use these?

git log --pretty - this is how we start.

Please remember about q ;)

So let's try

clear && git log --graph --pretty="%C(yellow) Hash: %h %C(blue)Date: %ad %C(red) Message: %s " --date=human

Please, go through the command and understand all arguments.

Let's test some different ideas. Try to guess what the command output will be, before you execute them!

clear && git log --graph --pretty="%ad" --date=short

For better clarity you can add format to parameter

git log --graph --pretty=format:"%ad" --date=short

It will do the same. Here are another examples.

clear && git log --graph --pretty="%ad"

clear && git log --graph --pretty="%at"

clear && git log --graph --pretty="%as"

clear && git log --graph --pretty="%C(bold blue)%h"

clear && git log --graph --pretty="%C(bold blue)%h %Cred%s %C(Yellow)by %an"

Another example found on the Internet

clear && git log --graph --pretty=format:"%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr) %C(bold blue)<%an>%Creset" --abbrev-commit

But... Will you enter this command every single time? No? What you need is on the next page :)

# Git config and aliases

You can create the aliases for your most commonly used commands.

Previously, we used very long command. Let's create an alias for it. We need to add it to the git config file. Of course logical will be to add it to global table.

git config --global alias.lg 'log --color --graph --pretty="%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr) %C(bold blue)<%an>%Creset" --abbrev-commit'

We instruct git that we want to change configuration (config), on global table (--global) and we configure alias called lg (alias.lg). The alias itself is inside the '' section.

Now you can execute your alias

clear && git lg

# Querying

Or more useful functionality - searching through git log.

You already know how to make your git log beautiful and how to format the output. But all the time we used all information from log. How to narrow the search?

We have a few possibilities.

clear && git log --author="John Doe" will show all commits done by specific author.

clear && git log --author="John Doe\|Joe Smith" - with this regex we asked for all commits authored by these two people.

## **And now a little bit of theory**

It is a good practice to have standards of creating messages for commits. There are a lot of examples in the Internet. Most of them propose to use kind of identifier. When we work with tools like Jira, we have amazing identifier given by the tool - Jira ticket number. So our commit message should look similar to:

JIRA-1234 my commit message

And voila! we have identifier, we can even attach it to Jira itself (by plugins). And use it... for searching through git log!

clear && git log --grep="JIRA-1234"

## **Theory ends :)**

Another way is to look at logs for specific file.

clear && git log -- testfile-01

clear && git log -- testfile-02 branchfile-01

Please notice the --. This way we inform git that we have files, not branches in mind.

We can compare two branches

clear && git log master..second-branch

clear && git log second-branch..master

Please, notice the differences between those two!!

Previously we tried to look for "merge" messages. There is better way.

clear && git log --merges

So, how to exclude merge commits:

clear && git log --no-merges

In our example it will not work very well, but we can search using dates

clear && git log --after=2021-4-21

clear && git log --before=2021-4-21

clear && git log --before 2021-4-30 --after=2021-4-1

clear && git log --after=yesterday

on the end, the easiest thing. We can limit number of returned commits

clear && git log -1

clear && git log -3

And also we can mix search functionalitites

clear && git log -1 --grep="JIRA"

clear && git log -5 --grep="commit" --oneline

Please notice revert message, without commit. hm... Please, check why we see it?

# Lesson 9: Houston, we have conflicts!

Conflict!

We will create a conflict in unusual but not uncommon way. In the background the system did some work, commits, etc. Now our goal is to go back two commits with revert. Unfortunately, our last two commits are about the same file.

Let's go to our repo cd test

How the history looks now?

git log --oneline

And for the curiosity, what is in testfile-02 ?

cat testfile-02

Let's try to do revert.

clear && git revert --no-edit HEAD~1

Ouch...

CONFLICT (content): Merge conflict in testfile-02

error: could not revert XXXXXXX... my new feature

Let's see what we have in file

cat testfile-02

Thank you git. You were supposed to destroy mess in the universe, not create one!

But wait... Let's look closely what happened.

How the revert works? It removes changes from selected commit, not from all changes!. Now things should be more clear, right?

* change1 commit
* change2 commit
* revert to change1

That means, we have change2 unattended!

<<<<<<< HEAD

This is my important change

=======

>>>>>>> parent of XXXX... my new feature

Here is our conflict.

<<<<<< - this is what we have in current branch

====== = center of our conflict

>>>>>> - this wants to be merged / added. In our case this part is empty (as it should be).

How to solve it? The simplest will be... talk with developers who did the changes which we are reverting, first. Find common solution. And then use vim and make appropriate changes.

Normally you should use vim , and remove lines from 2 to 5 . But here we do:

sed -i '2,5d' testfile-02

We removed mentioned lines. Now the file looks like we want.

cat testfile-02

Please notice, this example is extremely simple :)

The last thing to do is to add and commit the file again. Please look carefully on comments after

clear && git status

Ok, let's do the work.

git add testfile-02

git status

clear && git commit testfile-02 -m "fixed conflict"

git status

# Lesson 10: Powerful stash!

# Lesson 11: How to ignore some content

.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"

And now we can run git status to see what is changed.

Yes, we can confirm, what shouldn't be in remote isn't on the list.

# More configs

Ok, so now we are able to ignore some specified files.

Now we want the file with the same name neveringit from firstdirectory to be sent to remote. Maybe it is only an example how it should work. Some instruction for other people for example.

Let's create additional file then.

touch firstdirectory/neveringit

And its content

echo "this file has to go to git!" >> firstdirectory/neveringit

Ok. do

git status

git add . and then again

git status

Well. It is not what we want.

Ok, first, we know, the filename we used itself works for all directories in repo. Git has strict rules how these 'masks' work, please check in documentation if you want to have it more complicated.

The good practice is to be as most specific as possible. So, what we need, really is:

\*\*/neveringit

!firstdirectory/neveringit

First line says explicitly - all files anywhere in the structure (and now it is clear, readable and visible

In second line the exlamation mark negates the pattern. Another words, we negate the deny and allow this file to be sent to remote.

Ok, let's check.

sed -i 's/neveringit/\*\*\/neveringit/g' .gitignore

echo '!firstdirectory/neveringit' >> .gitignore

And check

cat .gitignore

Ok, to be sure, let's do an experiment.

mkdir thirddirectory && echo "gotogit" > thirddirectory/togit && echo "not for git!" > thirddirectory/neveringit

After a

git add . let's run

git status

Yep, we can confirm, neveringit file is never added to commit, except only one situation: when we forced it.

# Lesson 12: Git tags

# Create tags

The system prepared the repo with a few commits. At this moment we can say "ok, this is what we can treat as full functionality", a snapshot, if you wish.

First, let's go into repo's directory.

cd work

Now, let's check the status and log

git status

git adog

Yes, we see some work done.

We can do lightweight tags or annotated. Honestly, I do not like the first one. I prefer annotated tags as we can see some info there.

The difference here is how these types of tags are build. For lightweight tags git creates something similar to branch that doesn't change. In case of annotaded tag, git stores it as whole odject in database.

Let's check if we have any tags

git tag

No, we don't.

So we make one!

git tag -a v1.0 -m "version 1.0. initial state"

You can use what you wish for a tag. But good practice is to use v and number which describes the changes from previous tag. How to create proper tagging strategy - there is many documents about it.

# Check the tags

Ok, now we can check our tags again.

git tag

Yes, one is created. What is inside?

git show v1.0

# More tags and navigation

Right now system creates a few new files and commits them.

So, we will create more tags.

Let's try with checking what we have and the we tag current HEAD.

git adog

As we can see, tag v1.0 is marker somewhere in the middle.

Ok, we do the new tag now.

git tag -a v2.0 -m "version 2.0. A lot of new features"

git tag

git adog

As we can see, now git has two tags.

# Tag by commit

Oh... We just realized... We had to tag our state with v1.5 after the commit for testfile-06...

Now worries, we still can do it.

git adog | grep 'testfile-06' | awk '{print $2}' | head -n1

What happened here?

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.

Ok, as this is Killercoda, let's have some automation. We chaange output of the command above into variable.

commit2tag=$(git adog | grep 'testfile-06' | awk '{print $2}' | head -n1)

To be sure, we confirm if variable contains proper value.

cat $commit2tag

Now we will use this variable for commit.

git tag -a v1.5 -m "version 1.5. Some updates" $commit2tag

git tag

git adog

We succesfully tagged commit from history.

# Navigate through tags

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

Or

git describe --tags

to be more exact.

git describe will return info about current tag. If there is no tag created yet, git will throw an error. If we are somewhere in the middle:

commitbetweentags=$(git adog | grep 'testfile-05' | awk '{print $2}' | head -n1)

git checkout $commitbetweentags

git adog

git describe

We see now the tag on which we already started to build our changes.

# Delete tag

Ok, let's go back to the Master branch (what means to the top of our work before the recent change - in our case).

git checkout master

And now we delete v1.0 tag.

git tag -d v1.0

git tag

git adog

Yep, tag is deleted.

**v1.0**

**v1.5**

**v2.0**

**ubuntu $ git checkout v1.0**

**Note: switching to 'v1.0'.**

**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 9221558 my new feature**

**ubuntu $ git adog**

**\* 306943c (tag: v2.0, master) new features**

**\* 604e85d (tag: v1.5) commit for testfile-06**

**\* a702fd4 commit for testfile-05**

**\* 9221558 (HEAD, tag: v1.0) my new feature**

**\* ca98126 commit for all new files**

**\* bb92606 commit for testfile-02**

**\* 260234f commit for testfile-01**

**ubuntu $ git branch**

**\* (HEAD detached at v1.0)**

**master**

**ubuntu $ git tag**

**v1.0**

**v1.5**

**v2.0**

**ubuntu $ git describe**

**v1.0**

**ubuntu $ git describe --tags**

**v1.0**

**ubuntu $ commitbetweentags=$(git adog | grep 'testfile-05' | awk '{print $2}' | head -n1)**

**ubuntu $ git checkout $commitbetweentags**

**Previous HEAD position was 9221558 my new feature**

**HEAD is now at a702fd4 commit for testfile-05**

**ubuntu $ git adog**

**\* 306943c (tag: v2.0, master) new features**

**\* 604e85d (tag: v1.5) commit for testfile-06**

**\* a702fd4 (HEAD) commit for testfile-05**

**\* 9221558 (tag: v1.0) my new feature**

**\* ca98126 commit for all new files**

**\* bb92606 commit for testfile-02**

**\* 260234f commit for testfile-01**

**ubuntu $ git describe**

**v1.0-1-ga702fd4**

**ubuntu $ git checkout master**

**Previous HEAD position was a702fd4 commit for testfile-05**

**Switched to branch 'master'**

**ubuntu $ git tag -d v1.0**

**Deleted tag 'v1.0' (was 6a0085a)**

**ubuntu $ git tag**

**v1.5**

**v2.0**

**ubuntu $ git adog**

**\* 306943c (HEAD -> master, tag: v2.0) new features**

**\* 604e85d (tag: v1.5) commit for testfile-06**

**\* a702fd4 commit for testfile-05**

**\* 9221558 my new feature**

**\* ca98126 commit for all new files**

**\* bb92606 commit for testfile-02**

**\* 260234f commit for testfile-01**

**ubuntu $**

# Lesson 13: Merge branches

# Branches

As Atlassians explains, branch represents an independent line of development. We create branches to independently and uninterruptly work on some functionality, which later is merged to the main line.

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.

Are we on proper branch?

git branch

Yes, we are.

By the way, when git branch -r is executed, we list branches from remote. In our case we cannot do it, as we didn't set the remote for our repo.

Let's do some commits.

touch branchfile

echo "this was done on branch!" >> branchfile

git add .

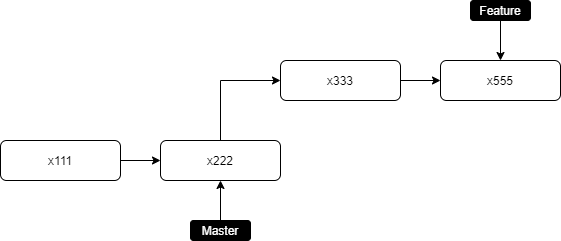
git commit branchfile -m "first commit on branch"

echo "added on branch" >> testfile-02

git commit testfile-02 -m "second commit on branch"

git adog

We see the log is quite flat now. That's ok. In fact, we have this:



Right, let's make some changes on master and then come back to this branch.

git checkout master

Confirm the proper branch by

git branch and check what we have in log

clear && git adog . Please notice the difference between HEAD pointer when checked on branch or master.

touch testfile-05

echo "this was done on master!" >> testfile-05

git add .

git commit testfile-05 -m "next commit on master"

echo "added on master" >> testfile-01

git commit testfile-01 -m "another commit on master"

git checkout newbranch

echo "another change on branch, again" >> testfile-02

git add .

git commit testfile-02 -m "commit from branch"

git adog

Ok, let's go back to master.

git checkout master

clear && git adog

This is what we have now:

Ok, now git log shows some trees, at least :) Why?

Simple, there are more changes, done in different time on different branches. The files and content of them started do be different.

The picture shows curent state.

Now we can switch to master and merge changes.

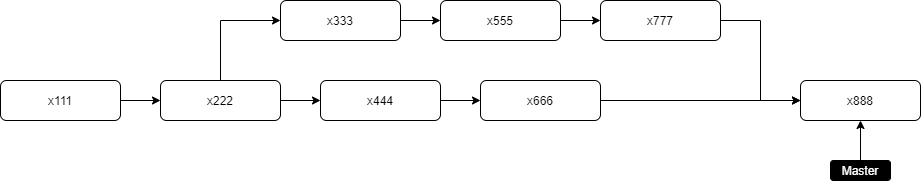
git checkout master

git merge newbranch --no-edit

clear && git adog

And this is the final state. Merge created a new commit to reflect changes caused by the process.

Final status is like on the picture below.



As we finished work on branch, we can delete it

git branch -d newbranch && git branch

# Conflicts

During the work on two branches it is possible that the changes will be implemented to the same file.

The system prepared the repo, let's see what we have in branch. One file was modified in master and in branch too.

So, let's switch to new repo.

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 adog - yes, this is what we expected

and what we have in our problematic file?

cat testfile-02

git branch -d newbranch && git branch

# Better approach

If you work on huge amount of changes, the good practice is to merge the master to your branch before merge to master.

It doesn't help to solve conflicts, but conflict will be on your branch, not master. What is definitelly more confortable for everyone.

Ok, we need to go to the new repo

cd ~/merge3

Ok, we are on the branch, let's do the merge.

git branch

git merge master --no-edit

Yes, we have conflict, but master is still clean.

sed -i '8d' testfile-02 && sed -i '6d' testfile-02 && sed -i '3d' testfile-02 && cat testfile-02

Worth noticing, the center of the conflict (=======) lies in different place than in the example in conflicts lesson.

Now, like last time, we need to stage and commit file.

git status

git add testfile-02

git commit -a -m "merge master to newbranch"

git adog

Ok, we fixed problems, and we are ready to merge our changes on master.

git checkout master

git merge newbranch --no-edit

git adog

So, we completed our merge.

git branch -d newbranch && git branch

**\* 1ebdee3 (HEAD -> master) Merge branch 'newbranch'**

**|\**

**| \* de2fc45 (newbranch) commit from branch**

**| \* 6aeec77 second commit on branch**

**| \* f9ce94b first commit on branch**

**\* | ecd17c4 another commit on master**

**\* | bd310db next commit on master**

**|/**

**\* ba6cd2f my new feature**

**\* 97ed987 commit for all new files**

**\* 83ab310 commit for testfile-02**

**\* 3f93659 commit for testfile-01**

**ubuntu $ git branch -d newbranch && git branch**

**Deleted branch newbranch (was de2fc45).**

**\* master**

**ubuntu $ cd ~/merge2**

**ubuntu $ git branch**

**master**

**\* newbranch**

**ubuntu $ git adog**

**\* 7a91bb7 (master) another commit on master - file testfile-02**

**\* d49afc4 next commit on master**

**| \* 17b631f (HEAD -> newbranch) commit from branch - testfile-02**

**| \* ff0d4da second commit on branch**

**| \* 456f8d1 first commit on branch**

**|/**

**\* 8fed563 my new feature**

**\* 37f0394 commit for all new files**

**\* 1ec0dfc commit for testfile-02**

**\* 18fef50 commit for testfile-01**

**ubuntu $ git checkout master && git adog**

**Switched to branch 'master'**

**\* 7a91bb7 (HEAD -> master) another commit on master - file testfile-02**

**\* d49afc4 next commit on master**

**| \* 17b631f (newbranch) commit from branch - testfile-02**

**| \* ff0d4da second commit on branch**

**| \* 456f8d1 first commit on branch**

**|/**

**\* 8fed563 my new feature**

**\* 37f0394 commit for all new files**

**\* 1ec0dfc commit for testfile-02**

**\* 18fef50 commit for testfile-01**

**ubuntu $ git merge newbranch**

**Auto-merging testfile-02**

**CONFLICT (content): Merge conflict in testfile-02**

**Automatic merge failed; fix conflicts and then commit the result.**

**ubuntu $ cat testfile-02**

**second file**

**This is my important change**

**<<<<<<< HEAD**

**added on master**

**=======**

**added on branch**

**another change on branch, again**

**>>>>>>> newbranch**

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

**second file**

**This is my important change**

**added on master**

**added on branch**

**another change on branch, again**

**ubuntu $ git status**

**On branch master**

**You have unmerged paths.**

**(fix conflicts and run "git commit")**

**(use "git merge --abort" to abort the merge)**

**Changes to be committed:**

**new file: branchfile**

**Unmerged paths:**

**(use "git add <file>..." to mark resolution)**

**both modified: testfile-02**

**ubuntu $ git add testfile-02**

**ubuntu $ git commit -a -m "confirm merge"**

**[master 6e07f8e] confirm merge**

**ubuntu $ cat testfile-02**

**second file**

**This is my important change**

**added on master**

**added on branch**

**another change on branch, again**

**ubuntu $ git branch -d newbranch && git branch**

**Deleted branch newbranch (was 17b631f).**

**\* master**

**ubuntu $ cd ~/merge3**

**ubuntu $ git branch**

**master**

**\* newbranch**

**ubuntu $ git merge master --no-edit**

**Auto-merging testfile-02**

**CONFLICT (content): Merge conflict in testfile-02**

**Automatic merge failed; fix conflicts and then commit the result.**

**ubuntu $ sed -i '8d' testfile-02 && sed -i '6d' testfile-02 && sed -i '3d' testfile-02 && cat testfile-02**

**second file**

**This is my important change**

**added on branch**

**another change on branch, again**

**added on master**

**ubuntu $ git status**

**On branch newbranch**

**You have unmerged paths.**

**(fix conflicts and run "git commit")**

**(use "git merge --abort" to abort the merge)**

**Changes to be committed:**

**new file: testfile-05**

**Unmerged paths:**

**(use "git add <file>..." to mark resolution)**

**both modified: testfile-02**

**ubuntu $ git add testfile-02**

**ubuntu $ git commit -a -m "merge master to newbranch"**

**[newbranch 4b5e6cd] merge master to newbranch**

**ubuntu $ git adog**

**\* 4b5e6cd (HEAD -> newbranch) merge master to newbranch**

**|\**

**| \* df23782 (master) another commit on master - file testfile-02**

**| \* e2d5da7 next commit on master**

**\* | 8cd44b8 commit from branch - testfile-02**

**\* | a1eff49 second commit on branch**

**\* | 5a1085f first commit on branch**

**|/**

**\* ef9ff63 my new feature**

**\* c83fc0c commit for all new files**

**\* 6b0b6b4 commit for testfile-02**

**\* ebecc7f commit for testfile-01**

**ubuntu $ git checkout master**

**Switched to branch 'master'**

**ubuntu $ git merge newbranch --no-edit**

**Updating df23782..4b5e6cd**

**Fast-forward**

**branchfile | 1 +**

**testfile-02 | 2 ++**

**2 files changed, 3 insertions(+)**

**create mode 100644 branchfile**

**ubuntu $ git adog**

**\* 4b5e6cd (HEAD -> master, newbranch) merge master to newbranch**

**|\**

**| \* df23782 another commit on master - file testfile-02**

**| \* e2d5da7 next commit on master**

**\* | 8cd44b8 commit from branch - testfile-02**

**\* | a1eff49 second commit on branch**

**\* | 5a1085f first commit on branch**

**|/**

**\* ef9ff63 my new feature**

**\* c83fc0c commit for all new files**

**\* 6b0b6b4 commit for testfile-02**

**\* ebecc7f commit for testfile-01**

**ubuntu $ git branch -d newbranch && git branch**

**Deleted branch newbranch (was 4b5e6cd).**

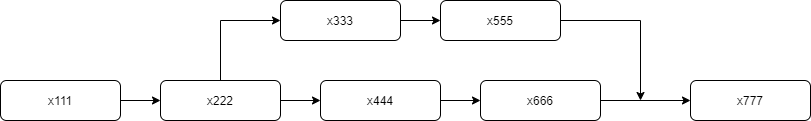
**\* master**

**ubuntu $**

# Lesson 14: Cheat the history!

# 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

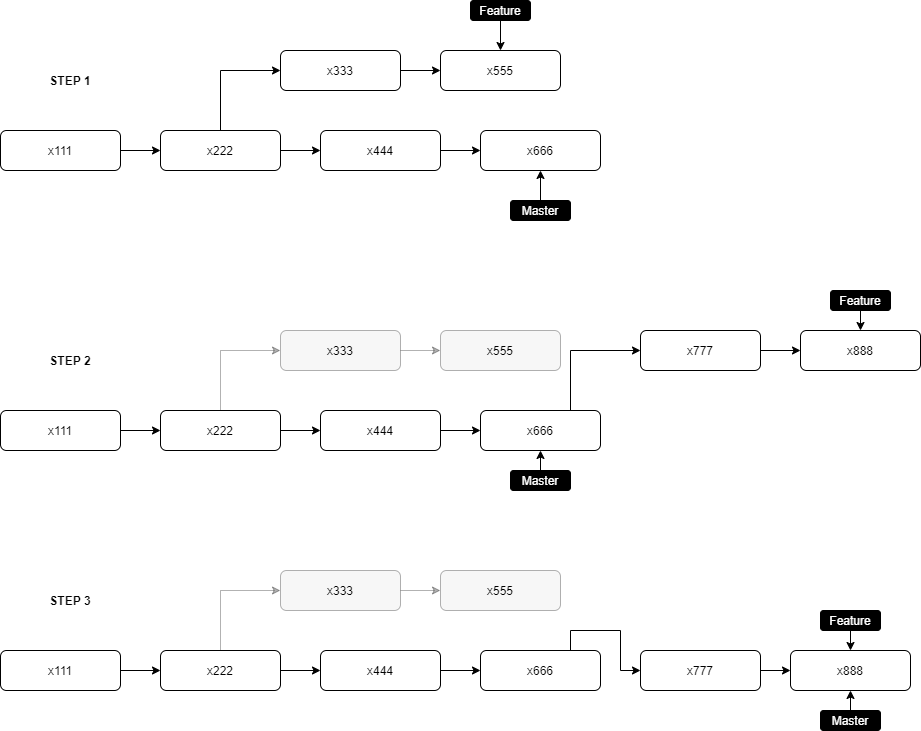
(do you remember our "a dog" config setting from git config module?)

Pretty simple, and nothing new. So, how rebase works?

Rebase

Let's see how the rebase looks like.

First take a look on the picture.



It is imperative to understand the rebase, especially the risks behind.

We are using rebase to make history more clean, clear and ordered. This clearly means, we play with history of commits. Not in the very aggresive way, but still, history is kind of corrupted. This is huge risk when a huge team is working on the same repository.

Also rebase may lead to many conflicts.

Ok, it is time to explain the steps.

* 1 step is similar to previous activity - merge. We have master and branch.
* Step2. When we perform the rebase operation, parent commit of the branch (the one from where we created branch) is moved to the HEAD of master, all differences between master and branch are merger and all commits on the branch are "copied/moved" (with all respective changes from master branch).
* Step 3. The last thing is to do merge on master branch in order to move HEAD of master to the proper place.

Let's see it in the example.

System prepared second repository in rebase directory, and the state is exactly the same like in the merge repo, but without final merge.

cd ../rebase

git adog

We see that branches are not merged.

Let's switch to the second-branch (in case if we are not there, but we should) and do the rebase with master.

git checkout second-branch

git rebase master

git adog

Right now we can see two things

* history looks totally different. Is 'flat' now and shuffled.
* HEAD of master is in the wrong position.

If we leave it that way, believe, we generate a chaos, mess, and comments of hatred ;P So, let's switch to master and then move the marker by merge the branch to master.

git checkout master

git merge second-branch

git adog

Now we have HEAD in proper place and the history which, in fact, does not represent the past work order, but is more readable.

Pleace notice, the merge was created in 'fast-forward' mode. It means that no additional commit was created, but the HEAD marker was moved.

**ubuntu $ cd merge**

**ubuntu $ git adog**

**\* 26ce1eb (HEAD -> master) Merge branch 'second-branch'**

**|\**

**| \* a908aed (second-branch) commit of branchfile-02 in branch**

**| \* 54b0f0f commit of branchfile-01 in branch**

**\* | a643f1b another work on master**

**|/**

**\* 522e98a JIRA-1234 my better feature!**

**\* bbd158c Revert "my new feature"**

**\* f45ad74 my new feature**

**\* 4d9e6d0 commit for all new files**

**\* b3f5b42 commit for testfile-02**

**\* 8c83747 commit for testfile-01**

**ubuntu $ cd ../rebase**

**ubuntu $ git adog**

**\* a643f1b (master) another work on master**

**| \* a908aed (HEAD -> second-branch) commit of branchfile-02 in branch**

**| \* 54b0f0f commit of branchfile-01 in branch**

**|/**

**\* 522e98a JIRA-1234 my better feature!**

**\* bbd158c Revert "my new feature"**

**\* f45ad74 my new feature**

**\* 4d9e6d0 commit for all new files**

**\* b3f5b42 commit for testfile-02**

**\* 8c83747 commit for testfile-01**

**ubuntu $ git checkout second-branch**

**Already on 'second-branch'**

**ubuntu $ git rebase master**

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

**Applying: commit of branchfile-01 in branch**

**Applying: commit of branchfile-02 in branch**

**ubuntu $ git adog**

**\* fb6dfb1 (HEAD -> second-branch) commit of branchfile-02 in branch**

**\* 32dcda2 commit of branchfile-01 in branch**

**\* a643f1b (master) another work on master**

**\* 522e98a JIRA-1234 my better feature!**

**\* bbd158c Revert "my new feature"**

**\* f45ad74 my new feature**

**\* 4d9e6d0 commit for all new files**

**\* b3f5b42 commit for testfile-02**

**\* 8c83747 commit for testfile-01**

**ubuntu $ git checkout master**

**Switched to branch 'master'**

**ubuntu $ git merge second-branch**

**Updating a643f1b..fb6dfb1**

**Fast-forward**

**branchfile-01 | 1 +**

**branchfile-02 | 1 +**

**2 files changed, 2 insertions(+)**

**create mode 100644 branchfile-01**

**create mode 100644 branchfile-02**

**ubuntu $ git adog**

**\* fb6dfb1 (HEAD -> master, second-branch) commit of branchfile-02 in branch**

**\* 32dcda2 commit of branchfile-01 in branch**

**\* a643f1b another work on master**

**\* 522e98a JIRA-1234 my better feature!**

**\* bbd158c Revert "my new feature"**

**\* f45ad74 my new feature**

**\* 4d9e6d0 commit for all new files**

**\* b3f5b42 commit for testfile-02**

**\* 8c83747 commit for testfile-01**