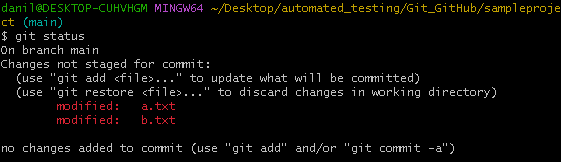
**Git commands**

**git init**  – for initialization (creating) local repository – it’ll represent .git folder in particular directory  
  
Изображение выглядит как текст, чек, снимок экрана, Шрифт

Автоматически созданное описание  
----------------------------------  
  
**git add -A** (adding all the file from the working space (this exact directory with .git folder) to the staging area) or **git add** **<file.extension>** (adding this exact file to the staging area) or **git add <file1.extension file2.extension…>** (adding several files at once to the staging area) or **git add \*.<extension>** (adding all the file with certain extension)  
  
Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

-------------------------------------  
  
**git status** – for checking out the status if there are some existing untracked files which are laying out in the working directory or there are some files in the staging area anticipating to be committed  
  


---------------------------------------  
  
**git restore --staged <file.extension>** (this command’ll restore file/files from the staging area back to the working space)  
  
Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

-----------------------------------------  
  
**git config --global user.email “<email>”  
git config --global user.name “<name>”** - this two commands are mainly required to use for providing your email and name just one time for this exact local repository (provide these configurations)

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

----------------------------------------

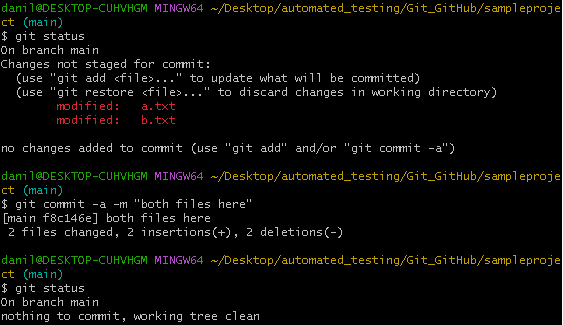
**git commit -m “<comment>”** (this command make our files from staging area move to the local repository(.git))

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

-------------------------------------------

**git commit -a -m** **“<comment>”** (we can employ such command when our files already were added to the local repository but later were changed, so they are not new files!)



----------------------------------------

**git log** (generates the info about all the commits we have done earlier)

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

------------------------------------------------

**Detailed analysis of next commands**

**Git diff**

1) **git diff** (for ex. I created a file index.txt and put some text in it, then I added it to the staging area by command “git add -A” and I changed a file in the working space again, and by checking the current status I have this situation:

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

, so now I execute following command to see the diverse between files in the staging area and working space:

Изображение выглядит как текст, снимок экрана, Шрифт, программное обеспечение

Автоматически созданное описание

In such case a/index.txt is a file storing in the staging area(source) and b/index.txt is a file laying in the working space. So here the comparison has been conducted (below we can see the difference)

-----------  
  
2) **git diff HEAD <name\_of\_file>** Now we’ll consider the case to compare files in working space and last commit(local repo):

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

3) **git diff--staged HEAD <name\_of\_file>** or **git diff--cached HEAD <name\_of\_file>** comparing staging area and last commit(local repo):

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

4) Now we’ll observe the comparison between specific commit and working space file, let’s consider: (here we gotta know the commit id, we can retrieve it through git log information because it’ll output the previous commits)

First I’ll execute a command called **git log --oneline** to get all the brief info about previous commits

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Now I may capture a particular id which I need:

**git diff <commit\_id> <name\_of\_file>**

**Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание**

5) comparison between specific commit and staging file:   
  
**git diff --staged <commit\_id>**

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

6) comparison between to specified commits: **git diff <commit\_id1> <commit\_id2> <name\_of\_file>**

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

We also can use git diff command for comparing branches or files between local and remote repos:  
  
**git diff <name\_of\_branch1> <name\_of\_branch2>** (branches)  
**git diff <name\_of\_branch1> origin/<name\_of\_branch2>** (branches in local and remote repos)

---------------------------------------------------------------  
  
  
**Git rm**

By first step I created new empty directory:  
  
Изображение выглядит как текст, снимок экрана, программное обеспечение, дисплей

Автоматически созданное описание

And here I created 3 files and local repository(by git init) and added these 3 files into staging area and committed inside local repo:  
  
Изображение выглядит как текст, снимок экрана, программное обеспечение, Шрифт

Автоматически созданное описание

Now I want to check which files are presented in the working directory and in the staging area consistently, for that I executed following commands(**ls** and **git ls-files**):  
Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Now I’ll accomplish some requirements:  
1. Remove file1 from both work directory and staging area  
**git rm <name\_of\_file>** (this command removes particular file from both sectors)

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

2. Remove all the files from both work directory and staging area

**git rm -r .** (this command removes all the files from both sectors)

Изображение выглядит как текст, снимок экрана, программное обеспечение, дисплей

Автоматически созданное описание

3. Remove files from only staging area

**git rm--cached <name\_of\_file>** (this command ‘ll remove file only from staging area)

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

4. Remove files only from working directory:  
rm <name\_of\_file> (this command’’ll remove file only from work directory)  
Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

-------------------------------------------------------------------

**Git checkout**

a) only works for working directory

b) to discard unstaged changes (the changes which are not added to staging area, but they have been applied to files in the working directory, files were modified)

c) in the tracked files (the files which are already added to staging area/committed earlier)  
  
At the beginning I created new file in my working directory, then sent it to the staging area and finally committed to local repo(had created it before by git init command):

Изображение выглядит как текст, снимок экрана, дисплей, Шрифт

Автоматически созданное описание

Now I changed file1.txt in my working directory by adding two more lines in the text:  
  
Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Here I’ll use the command to discard the unstaged changes in the tracked file1.txt of working directory: **git checkout -- <name\_of\_file>**

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

**Git reset**

1) to remove changes from staging area

2) to undo commits at repository level

At first I added my file(index.txt) to staging area and then with git reset <name\_of\_file> I remove changes from staging area, and now I don’t have file in the staging area:

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Now I want to provide reset commands to the local repository, for that I created 3 files (file1,file2,file3) and had removed index.txt; I added those 3 files to the staging area and committed them to local repo by different commits(in different time):

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

And here how we can accomplish this task:

We gotta execute such command: git reset <mode> <commit\_id>  
where mode might be --mixed, --soft or --hard  
  
1)--mixed Mode: discarding commits in the local repo and staging area (without touching working directory), I’d like to discard latest commit so I have to provide previous commit\_id completed right before the previous commit, in this case all the commits completed after second last commit will be discarded.

**git reset --mixed <commit\_id>**

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

2)--soft Mode, here after applying this parameter with git reset command commit will be discarded, but it won’t touch staging area and working directory

I again added and committed file3.txt so now I have this picture:

**git reset --soft <commit\_id>**

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

And now let’s apply--soft attribute and see what’ll happen:

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

3) Eventually let us discuss reset command with --hard option: here by employing this get reset with –hard option we’ll remove our files from everywhere: work directory, staging area, local repo. So we gotta be careful:   
command: **git reset --hard <commit\_id>**

**Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание**

-----------------------------

**Git Branching commands**

So to work separately on the same project (let’s say 3 developers working together, but each of them is preoccupied with certain feature, but there’s one set of files (one project) for all of them, so they have to apply branching approach and create branches, branch – is independent, isolated workspace for developer, so every developer may work without disturbance by other developers. And whatever files and commits are in the main branch, they all will be inherited by child branches (created during the process of development).  
  
1) to view available branches within this particular project:

**git branch**  
Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание  
Here I only have one branch, it was created when I committed my first file in local repo.  
\* - indicates current branch

2) creation of new branch

**git branch <desired\_name\_of\_branch>**

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

3) To switch to another branch  
**git checkout <branch\_name>**

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

4) I want to create and switch to another branch at once.

**git checkout -b <desired\_name\_of\_branch>**

**Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание**

after creating some branches we can use diverse set of commands to work with these branches, to check which files belong to which branches, so I made it with such commands as **touch <name\_of\_file>** - for creating an idle, empty file, git branch, git branch <name\_of\_branch>, git checkout <name\_of\_branch>, git checkout -b <name\_of\_branch>, ls, git ls-files, git log --oneline.  
------------------------------------------------

**Merging**

There’s a situation when I have my main branch and I created a new branch(let’s name it as a feature branch). So this feature branch has inherited all the files and commits from the main branch. Furthermore, I can merge my feature branch with my main branch to consolidate two branches into one holistic one.  
  
But there are two methods of merging we need to examine.  
  
First one is fast-forward merge. Case considers the situation when I created a new branch accomplished all the actions not correlated with my main branch in the new branch whilst my main branch stays immutable(unchangeable) and then I conduct the merging process. So that ways thete’s no chance to catch a conflict between two branches.  
  
Second one is called three-way merge. Analogously the previous instance I provide the same actions except one where I modify my main branch by adding new commits, files and so forth during my feature branch is also being changed by some developer. So after merging the feature branch and main branch together there’s a great chance to get a conflict and co-called merge commit will be created.

Let’s take a close look at such particular situations:  
  
**Fast-Forward merge**

First, I create a new project, create two files(a.txt, b.txt) and add them in created as well local repo.  
So currently I have 2 commits in my main branch

Main ------------> a.txt, b.txt

Изображение выглядит как текст, электроника, снимок экрана, программное обеспечение

Автоматически созданное описание

Now I’d like to create in a new branch(feature branch) two more files

Feature --------------------> a.txt, b.txt(inherited), x.txt, y.txt

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Изображение выглядит как текст, электроника, снимок экрана, программное обеспечение

Автоматически созданное описание

Now let’s move on to merging. Assuming that I want to merge my feature branch into my main branch I must execute that by being within my main branch so I have to pass this merging process from the main branch (need to switch with checkout command)

**git merge <name\_of\_desirable\_merging\_branch>** (This command will merge the feature branch(in my case) into the main branch)

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание  
  
Here I don’t have any conflicts because this is a fast-forward merge

Now I’ll try to reproduce such situation and see what happens:

**Three-way Merge**

Изображение выглядит как круг, снимок экрана, диаграмма, линия

Автоматически созданное описание

I’m gonna have two branches: first I’ll create two commits on my main branch, then I’ll make one more branch(feature) where I’ll add two another commits, then I’ll come back to my main branch and push one more commit and finally merge these branches.

Currently I have this particular situation.

Изображение выглядит как текст, снимок экрана, программное обеспечение, Шрифт

Автоматически созданное описание

Now adding file to the main branch and committing it

Изображение выглядит как текст, снимок экрана, программное обеспечение

Автоматически созданное описание

After trying to merge feature branch into main branch I got this display in the console. So I needed to click <‘i’, esc, :wq!> to save this merging.

Изображение выглядит как текст, электроника, снимок экрана, дисплей

Автоматически созданное описание

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

‘Ort’ represents three-way merge. But here I didn’t receive any conflicts, because all my files were added independently, not concerning previous commits.  
  
And on the main branch one more commit (merge commit ) was created after merging.

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

---------------------------------------

**Merge conflicts. How to resolve them**

Suppose I have to work with such situation:

Изображение выглядит как текст, диаграмма, снимок экрана, линия

Автоматически созданное описание

Now I’ll recreate this into my new project directory. Let’s see:

First I added two commits on one file in my main branch:

Изображение выглядит как текст, снимок экрана, программное обеспечение, Шрифт

Автоматически созданное описание

\*By the way, to look at the content of your file execute following command:

**cat <name\_of\_file>**now I’m gonna create another branch (child branch -> feature). And I want to append one extra line to my file text.txt being in the feature branch:  
  
Изображение выглядит как текст, снимок экрана, Шрифт, программное обеспечение

Автоматически созданное описание

Now I’ll return to my main(parent) branch and add one extra line to the same file but already being in my main branch.

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

And finally we’re about to merge these branches and see what’ll happen.

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

I got this message about unsuccessful try to merge two branches. There’s a conflict which we have to resolve, then we can again add our file to the staging area, make a commit and provide the merging again.

If I execute cat text.txt command I’ll get some lines with emphasizing problems.

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

As a developer I want to resolve this problem, I should resolve this manually. Through vim command I decide to keep both lines, so I save this file:

Изображение выглядит как текст, снимок экрана, дисплей, программное обеспечение

Автоматически созданное описание

And now by adding and committing this file being in the uneven state (main|merging – branch) we resolve this problem and merging happens automatically.

Изображение выглядит как текст, снимок экрана, программное обеспечение, Шрифт

Автоматически созданное описание

And by completing the command: **git log --oneline --graph** we can observe the situation more clear how It’s done.

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

We also can delete unused branch, so I’ll delete feature branch by implementing command  
**git branch -d <name\_of\_branch\_to\_delete>**

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

This is how we can resolve conflicts in local repository.

-----------------------------------------------

**Git Rebase**

It’s kind of a merging process but not exactly; by executing rebase command (considering previous examples from recent chapters – I only have two branches: ‘main’ and ‘feature’) concerning my commits in the feature branch, I create those commits’ objects of all commits incorporated in my feature branch. By that action I intentionally discard my commits from feature branch (but now there are the same number of commit’s objects) and sort of put these created objects on top of my main branch, then I provide merge command and merging goes subtly without any conflicts. Let’s see in the real instance how It’s gonna work.

I’m gonna do step by step process like it’s shown on the picture below.

Изображение выглядит как текст, диаграмма, снимок экрана, круг

Автоматически созданное описание

So first, I create two idle .txt files and committed them into local repository.

Изображение выглядит как текст, снимок экрана, программное обеспечение, Шрифт

Автоматически созданное описание

Now I’ll create a new branch from here and add two extra .txt files and commit them into local repository as well:

Изображение выглядит как текст, снимок экрана, дисплей, программное обеспечение

Автоматически созданное описание

Moreover, I came back to the main branch and appended one more .txt file and pushed it to the local repository again. Now I have this state:

Изображение выглядит как текст, снимок экрана, программное обеспечение, дисплей

Автоматически созданное описание

Now I’m moving forward to rebase part of this project, First I switch to my feature branch and start executing right from there.

**get rebase <name\_of\_branch>**

Изображение выглядит как текст, снимок экрана

Автоматически созданное описание

Thus, I conducted rebasing and now It’s like c3m commit was the base commit right before creating feature branch. So now I kind of put two feature branch commits on top of all commits from main branch. And also c1f and c2f commits became commit objects and they changed the ids.

And finally I execute already well-known git merge command to merge my feature branch into the main branch.

Изображение выглядит как текст, снимок экрана, программное обеспечение

Автоматически созданное описание

And we’re observing rebase concept here.

---------------------------------------

**GitHub**

This is hosting service for git repositories, it is a tool used by developers, testers and other people in need. So, after completing the task in the working directory we can commit our elaborations into local repository and then push it to the remote repository (GitHub) and before operations with code, before starting developing some new feature we tend to pull remote repository from github into our local repository and start working.

Изображение выглядит как текст, снимок экрана, диаграмма, дизайн

Автоматически созданное описание

This is our flow of work interfacing between remote and local repositories.  
So the main activities within local/remote repos system are git pull, git push, git clone commands.

Let’s see how it works practically.

Изображение выглядит как диаграмма, снимок экрана, мультфильм, линия

Автоматически созданное описание

Normally there’s a project created in remote repositories. And members of the team are asked to clone that project to their local machines, and after that they’re free to develop their code. Cloning means that they would create their local repositories based on remote one. Pushing means transmitting code written by developers or testers to remote repository. And pulling implies that there’s already existed project on your local machine, but some changes occurred on remote repository and you just want to get them by executing pull command.

Now I’ve created new remote repository (name: testrepo, public):

Изображение выглядит как текст, снимок экрана, Шрифт, число

Автоматически созданное описание

System has given a link to this repository (red mark), which is truly important:

Изображение выглядит как текст, снимок экрана, Шрифт, программное обеспечение

Автоматически созданное описание

I have added one file (myfile) inside this remote repository using GitHub.com itself.

Изображение выглядит как текст, снимок экрана, Шрифт, линия

Автоматически созданное описание

Now I want to clone this remote repository (by that creating local repository) so all I have to do is direct to some idle folder on my local machine and launch gitBash there and execute simple command: **git clone ‘<link\_of\_remote\_repo>’**

Изображение выглядит как текст, снимок экрана, программное обеспечение, Значок на компьютере

Автоматически созданное описание

By ‘cd testrepo’ command I moved to the local repository exactly. Now I want to modify the existing file. And I also created myfile2

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание  
  
These changes are only in local repository now, but I want them to be in the remote repository as well. So I have to complete push command:

**git push origin <name\_of\_desirable\_branch>**    
 **Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание**

so in the remote repository it reflects fine, we can observe this:

Изображение выглядит как текст, снимок экрана, Шрифт, число

Автоматически созданное описание

So, to test pull command I firstly edit myfile2 within remote repository:

Изображение выглядит как текст, снимок экрана, Шрифт, число

Автоматически созданное описание

These changes of course are not displaying into my local repository file (myfile2), because of that I should perform **git pull origin <name\_of\_desirable\_branch>** command:

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

--------------------------------------

**How to work with Remote repositories through Eclipse $ IntelliJ IDE’s**

First of all I’ve created idle remote repository on GitHub service and It has name ‘testrepo2’. Then I’ve appended a file ‘test1’ and committed it.

Изображение выглядит как текст, снимок экрана, Шрифт, число

Автоматически созданное описание

Now I’ll open Eclipse IDE and import(clone my existing remote repository into new project) stored file within remote repository.

Изображение выглядит как текст, снимок экрана, дисплей, число

Автоматически созданное описание

I have modified this file via Eclipse but as we remember it means that I changed it within only working directory. Now I want to add it to the staging area, local repo and so on.  
Through UI I can do it.

1. Adding to the staging area

Изображение выглядит как текст, снимок экрана, число, программное обеспечение

Автоматически созданное описание

2. Adding to the local repository:

Изображение выглядит как текст, снимок экрана, число, программное обеспечение

Автоматически созданное описание

Which will lead to arising of pop-up window in the right top corner Here we may select one of two options: either commit to local repository or commit to local and push our changes to remote repositories having provided credentials to GitHub first.

3. And also we can pull some changes from remote to local repository.