# What Is Git ? – Explore A Distributed Version Control Tool

## ****What is Git – Why Git**** ****Came Into Existence?****

We all know “Necessity is the mother of all inventions”. And similarly Git was also invented to fulfill certain necessities that the developers faced before Git. So, let us take a step back to learn all about Version Control Systems (VCS) and how Git came into existence.

Version Control is the management of changes to documents, computer programs, large websites and other collection of information.

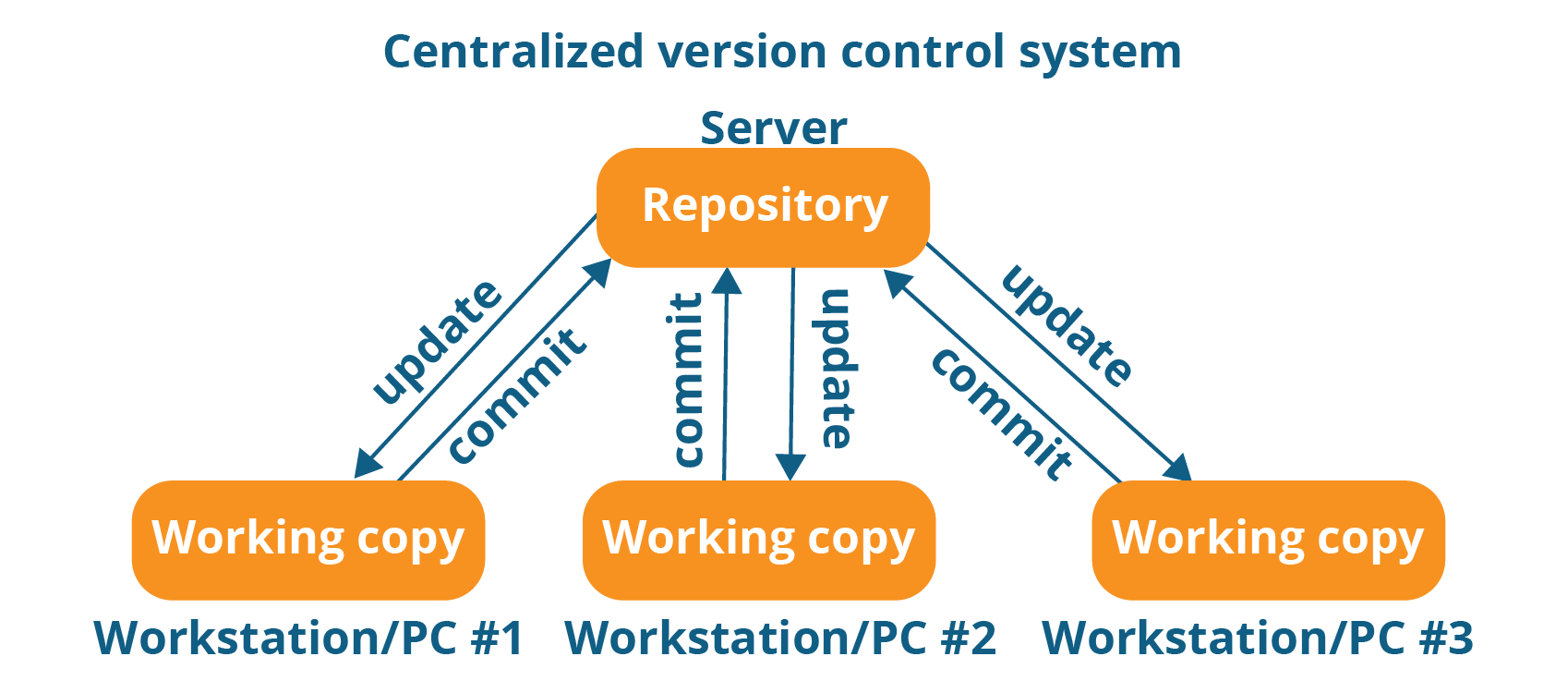
There are two types of VCS:

* Centralized Version Control System (CVCS)
* Distributed Version Control System (DVCS)

## ****Centralized VCS****

Centralized version control system (CVCS) uses a central server to store all files and enables team collaboration. It works on a single repository to which users can directly access a central server.

Please refer to the diagram below to get a better idea of CVCS:



The repository in the above diagram indicates a central server that could be local or remote which is directly connected to each of the programmer’s workstation.

Every programmer can extract or **update** their workstations with the data present in the repository or can make changes to the data or **commit** in the repository. Every operation is performed directly on the repository.

Even though it seems pretty convenient to maintain a single repository, it has some major drawbacks. Some of them are:

* It is not locally available; meaning you always need to be connected to a network to perform any action.
* Since everything is centralized, in any case of the central server getting crashed or corrupted will result in losing the entire data of the project.

This is when Distributed VCS comes to the rescue.

## ****Distributed VCS****

These systems do not necessarily rely on a central server to store all the versions of a project file.

In Distributed VCS, every contributor has a local copy or “clone” of the main repository i.e. everyone maintains a local repository of their own which contains all the files and metadata present in the main repository.

You will understand it better by referring to the diagram below:



As you can see in the above diagram, every programmer maintains a local repository on its own, which is actually the copy or clone of the central repository on their hard drive. They can commit and update their local repository without any interference.

They can update their local repositories with new data from the central server by an operation called “**pull**” and affect changes to the main repository by an operation called “**push**” from their local repository.

The act of cloning an entire repository into your workstation to get a local repository gives you the following advantages:

* All operations (except push & pull) are very fast because the tool only needs to access the hard drive, not a remote server. Hence, you do not always need an internet connection.
* Committing new change-sets can be done locally without manipulating the data on the main repository. Once you have a group of change-sets ready, you can push them all at once.
* Since every contributor has a full copy of the project repository, they can share changes with one another if they want to get some feedback before affecting changes in the main repository.
* If the central server gets crashed at any point of time, the lost data can be easily recovered from any one of the contributor’s local repositories.

## ****What Is Git?****

Git is a Distributed Version Control tool that supports distributed non-linear workflows by providing data assurance for developing quality software. Before you go ahead, check out this video on GIT which will give you better in-sight.

Git provides with all the Distributed VCS facilities to the user that was mentioned earlier. Git repositories are very easy to find and access. You will know how flexible and compatible Git is with your system when you go through the features mentioned below:

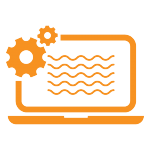
## ****What is Git – Features Of Git****

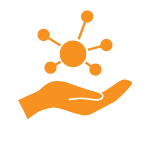
**Free and open source:**  
Git is released under GPL’s (General Public License) open source license. You don’t need to purchase Git. It is absolutely free. And since it is open source, you can modify the source code as per your requirement.

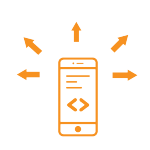
**Speed:**Since you do not have to connect to any network for performing all operations, it completes all the tasks really fast. Performance tests done by Mozilla showed it was an order of magnitude faster than other version control systems. Fetching version history from a locally stored repository can be one hundred times faster than fetching it from the remote server. The core part of Git is written in C, which avoids runtime overheads associated with other high level languages.  
**Scalable:**Git is very scalable. So, if in future , the number of collaborators increase Git can easily handle this change. Though Git represents an entire repository, the data stored on the client’s side is very small as Git compresses all the huge data through a lossless compression technique.

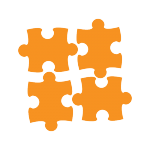
**Reliable:**Since every contributor has its own local repository, on the events of a system crash, the lost data can be recovered from any of the local repositories. You will always have a backup of all your files.

**Secure:**Git uses the ***SHA1*** (Secure Hash Function) to name and identify objects within its repository. Every file and commit is check-summed and retrieved by its checksum at the time of checkout. The Git history is stored in such a way that the ID of a particular version (a commit in Git terms) depends upon the complete development history leading up to that commit. Once it is published, it is not possible to change the old versions without it being noticed.

**Economical:**  
In case of CVCS, the central server needs to be powerful enough to serve requests of the  entire team. For smaller teams, it is not an issue, but as the team size grows, the hardware  limitations of the server can be a performance bottleneck. In case of DVCS, developers don’t  interact with the server unless they need to push or pull changes. All the heavy lifting  happens on the client side, so the server hardware can be very simple indeed.  
**Supports non-linear development:**Git supports rapid branching and merging, and includes specific tools for visualizing and navigating a non-linear development history. A core assumption in Git is that a change will be merged more often than it is written, as it is passed around various reviewers. Branches in Git are very lightweight. A branch in Git is only a reference to a single commit. With its parental commits, the full branch structure can be constructed.

 **Easy Branching:** Branch management with Git is very simple. It takes only few seconds to create, delete, and merge branches. Feature branches provide an isolated environment for every change to your codebase. When a developer wants to start working on something, no matter how big or small, they create a new branch. This ensures that the master branch always contains production-quality code.

**Distributed development:**  
Git gives each developer a local copy of the entire development history, and changes are copied from one such repository to another. These changes are imported as additional development branches, and can be merged in the same way as a locally developed branch.

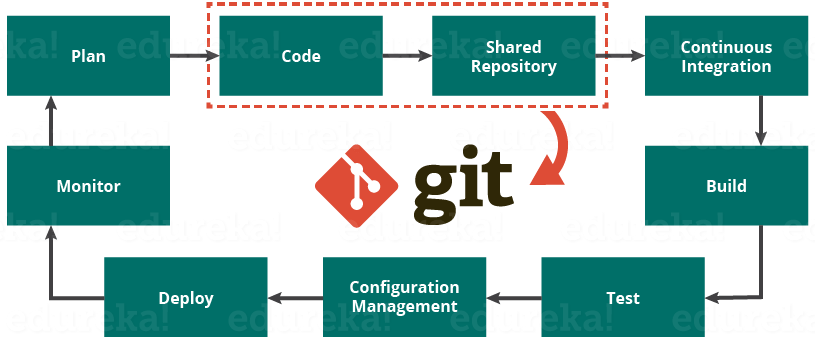
**Compatibility with existing systems or protocol**  
Repositories can be published via http, ftp or a Git protocol over either a plain socket, or ssh. Git also has a Concurrent Version Systems (CVS) server emulation, which enables the use of existing CVS clients and IDE plugins to access Git repositories. Apache SubVersion (SVN) and SVK repositories can be used directly with Git-SVN.

## ****What is Git – Role Of Git In DevOps?****

Now that you know what is Git, you should know Git is an integral part of DevOps.

DevOps is the practice of bringing agility to the process of development and operations. It’s an entirely new ideology that has swept IT organizations worldwide, boosting project life-cycles and in turn increasing profits. DevOps promotes communication between development engineers and operations, participating together in the entire service life-cycle, from design through the development process to production support.

The diagram below depicts the Devops life cycle and displays how Git fits in Devops.

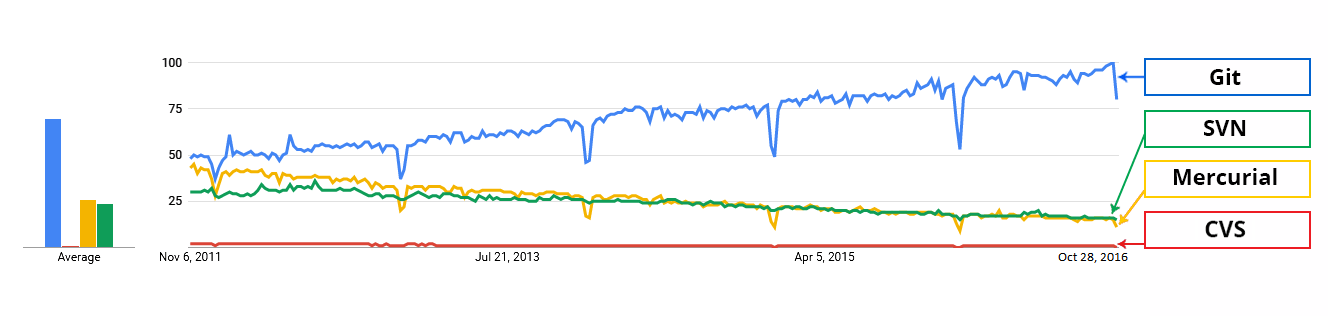


The diagram above shows the entire life cycle of Devops starting from planning the project to its deployment and monitoring. Git plays a vital role when it comes to managing the code that the collaborators contribute to the shared repository. This code is then extracted for performing continuous integration to create a build and test it on the test server and eventually deploy it on the production.

Tools like Git enable communication between the development and the operations team. When you are developing a large project with a huge number of collaborators, it is very important to have communication between the collaborators while making changes in the project. Commit messages in Git play a very important role in communicating among the team. The bits and pieces that we all deploy lies in the Version Control system like Git. To succeed in DevOps, you need to have all of the communication in Version Control. Hence, Git plays a vital role in succeeding at DevOps.

## ****Companies Using Git****

Git has earned way more popularity compared to other version control tools available in the market like Apache Subversion(SVN), Concurrent Version Systems(CVS), Mercurial etc. You can compare the interest of Git by time with other version control tools with the graph collected from Google Trends below:



In large companies, products are generally developed by developers located all around the world. To enable communication among them, Git is the solution.

Some companies that use Git for version control are: Facebook, Yahoo, Zynga, Quora, Twitter, eBay, Salesforce, Microsoft and many more.

Lately, all of Microsoft’s new development work has been in Git features. Microsoft is migrating .NET and many of its open source projects on GitHub which are managed by Git.

One of such projects is the LightGBM. It is a fast, distributed, high performance gradient boosting framework based on decision tree algorithms which is used for ranking, classification and many other machine learning tasks.

Here, Git plays an important role in managing this distributed version of LightGBM by providing speed and accuracy.

# Git Tutorial – Commands And Operations In Git

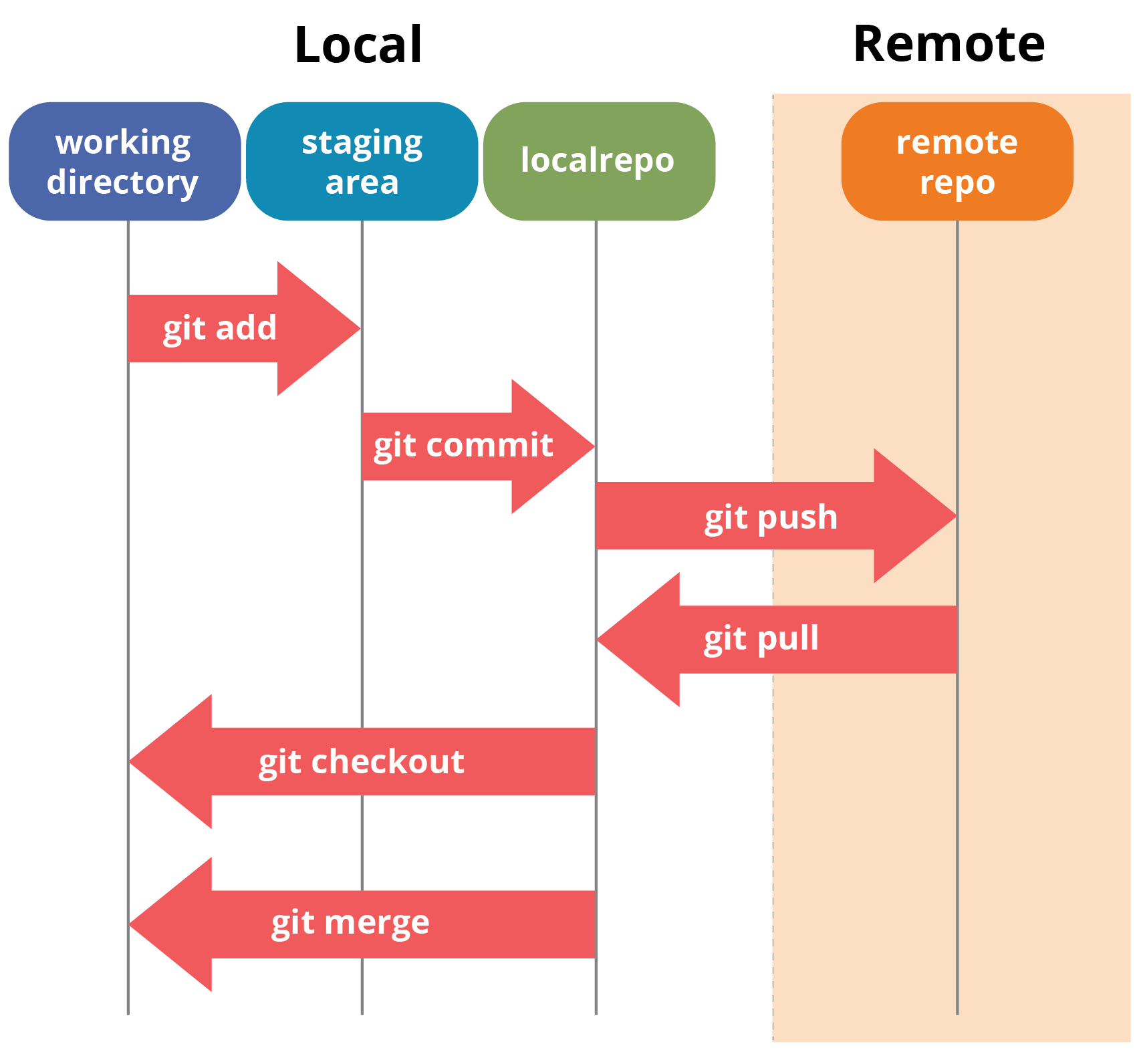
Some of the basic operations in Git are:

1. Initialize
2. Add
3. Commit
4. Pull
5. Push

Some advanced Git operations are:

1. Branching
2. Merging
3. Rebasing

Let me first give you a brief idea about how these operations work with the Git repositories. Take a look at the architecture of Git below:



If you understand the above diagram well and good, but if you don’t, you need not worry, I will be explaining these operations in this Git Tutorial one by one. Let us begin with the basic operations.

In this Git Tutorial, I will show you the commands and the operations using Git Bash. Git Bash is a text-only command line interface for using Git on Windows which provides features to run automated scripts.

After installing Git in your Windows system, just open your folder/directory where you want to store all your project files; right click and select ‘***Git Bash here***’.

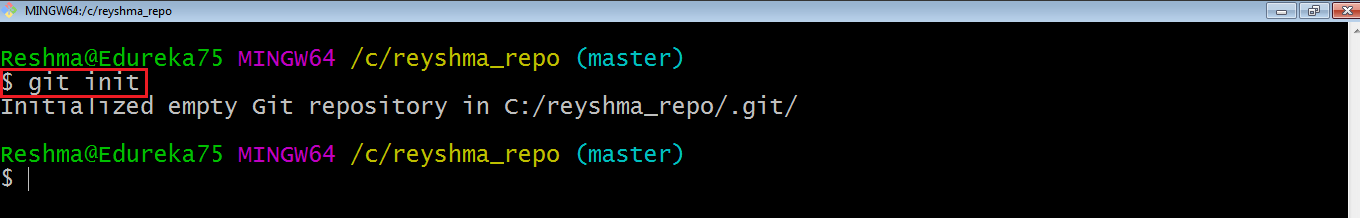


This will open up Git Bash terminal where you can enter commands to perform various Git operations.

Now, the next task is to initialize your repository.

## ****Initialize****

In order to do that, we use the command **git init.**Please refer to the below screenshot.



**git init** creates an empty Git repository or re-initializes an existing one. It basically creates a**.git** directory with sub directories and template files. Running a **git init** in an existing repository will not overwrite things that are already there. It rather picks up the newly added templates.

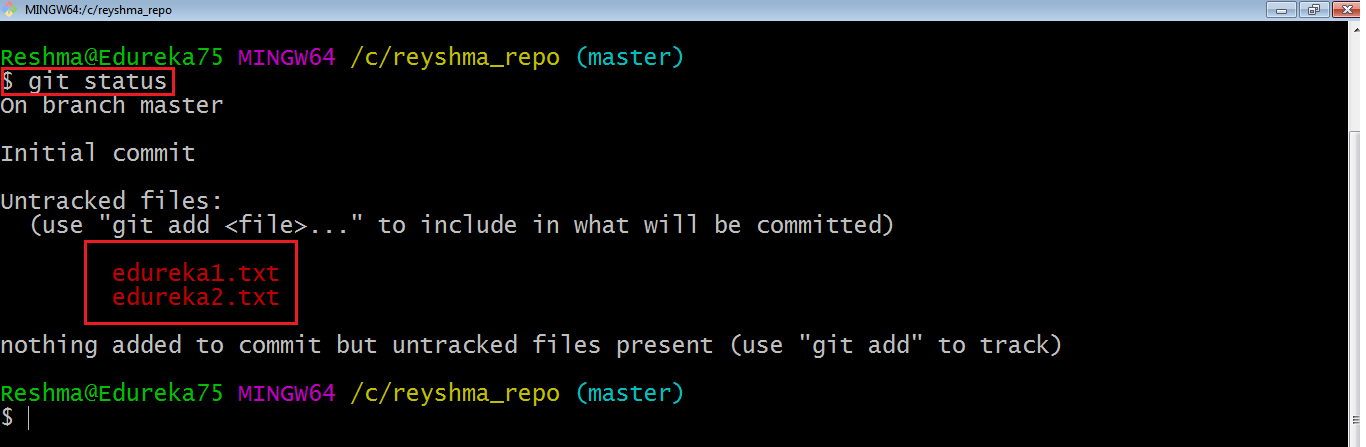
Now that my repository is initialized, let me create some files in the directory/repository. For e.g. I have created two text files namely edureka1.txt and edureka2.txt.

Let’s see if these files are in my index or not using the command **git status**. The index holds a snapshot of the content of the working tree/directory, and this snapshot is taken as the contents for the next change to be made in the local repository.

**Git status**

The **git status**command lists all the modified files which are ready to be added to the local repository.

Let us type in the command to see what happens:



This shows that I have two files which are not added to the index yet. This means I cannot commit changes with these files unless I have added them explicitly in the index.

**Add**

This command updates the index using the current content found in the working tree and then prepares the content in the staging area for the next commit.

Thus, after making changes to the working tree, and before running the **commit** command, you must use the **add** command to add any new or modified files to the index. For that, use the commands below:

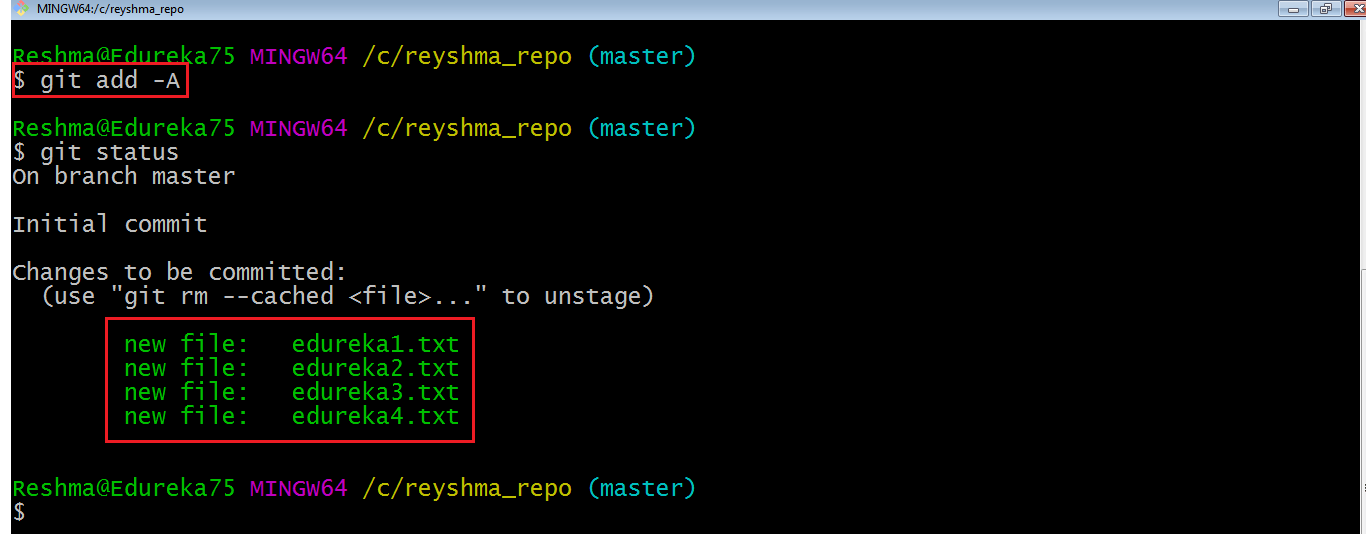
**git add <directory>**

or

**git add <file>**

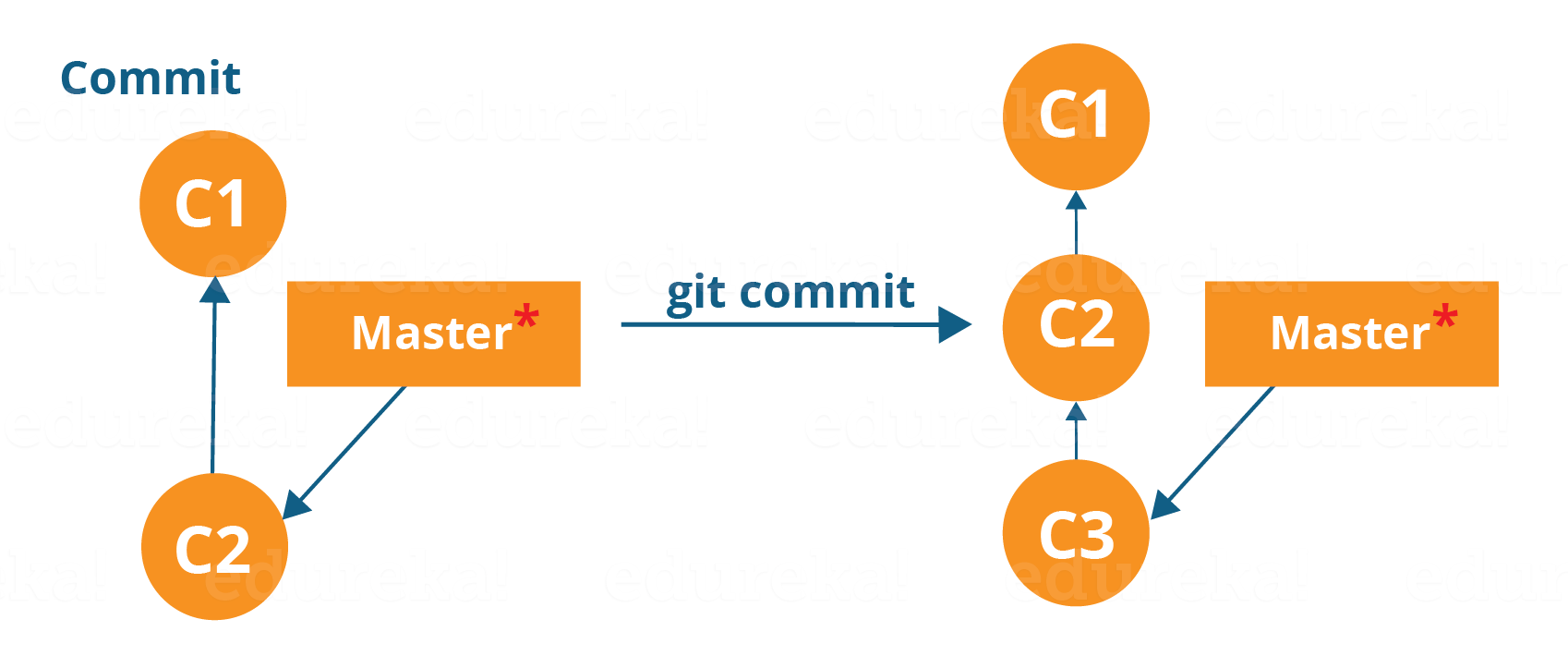
Let me demonstrate the **git add** for you so that you can understand it better.

I have created two more files edureka3.txt and edureka4.txt. Let us add the files using the command **git add -A**. This command will add all the files to the index which are in the directory but not updated in the index yet.



Now that the new files are added to the index, you are ready to commit them.

**Commit**

It refers to recording snapshots of the repository at a given time. Committed snapshots will never change unless done explicitly. Let me explain how commit works with the diagram below:

Here, C1 is the initial commit, i.e. the snapshot of the first change from which another snapshot is created with changes named C2. Note that the master points to the latest commit.

Now, when I commit again, another snapshot C3 is created and now the master points to C3 instead of C2.

Git aims to keep commits as lightweight as possible. So, it doesn’t blindly copy the entire directory every time you commit; it includes commit as a set of changes, or “delta” from one version of the repository to the other. In easy words, it only copies the changes made in the repository.

You can commit by using the command below:

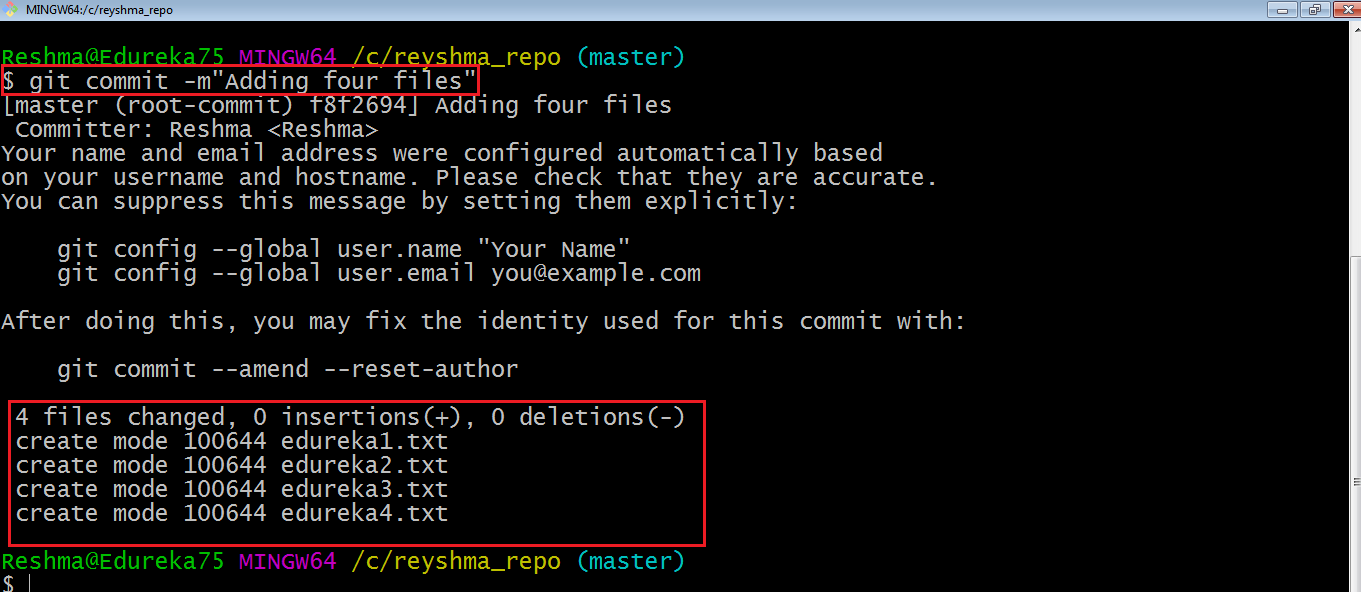
**git commit**

This will commit the staged snapshot and will launch a text editor prompting you for a commit message.

Or you can use:

**git commit -m “<message>”**

Let’s try it out.



As you can see above, the **git commit** command has committed the changes in the four files in the local repository.

Now, if you want to commit a snapshot of all the changes in the working directory at once, you can use the command below:

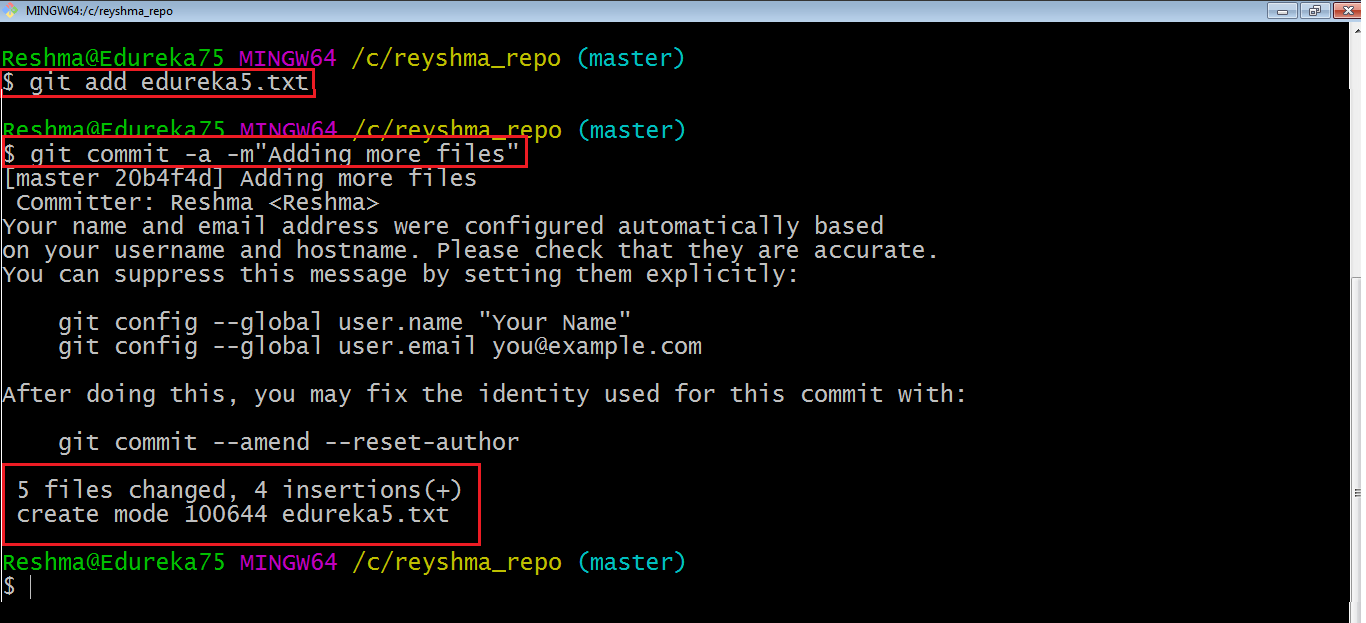
**git commit -a**

I have created two more text files in my working directory viz. edureka5.txt and edureka6.txt but they are not added to the index yet.

I am adding edureka5.txt using the command:

**git add edureka5.txt**

I have added edureka5.txt to the index explicitly but not edureka6.txt and made changes in the previous files. I want to commit all changes in the directory at once. Refer to the below snapshot.



This command will commit a snapshot of all changes in the working directory but only includes modifications to tracked files i.e. the files that have been added with **git add** at some point in their history. Hence, edureka6.txt was not committed because it was not added to the index yet. But changes in all previous files present in the repository were committed, i.e. edureka1.txt, edureka2.txt, edureka3.txt, edureka4.txt and edureka5.txt.  
Now I have made my desired commits in my local repository.

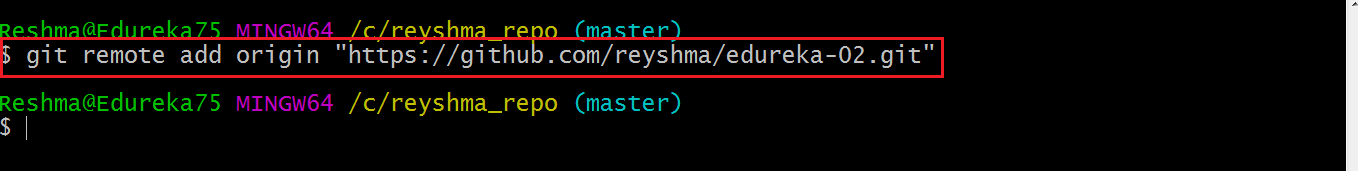
Note that before you affect changes to the central repository you should always pull changes from the central repository to your local repository to get updated with the work of all the collaborators that have been contributing in the central repository. For that we will use the **pull** command.

## ****Pull****

The **git pull** command fetches changes from a remote repository to a local repository. It merges upstream changes in your local repository, which is a common task in Git based collaborations.

But first, you need to set your central repository as origin using the command:

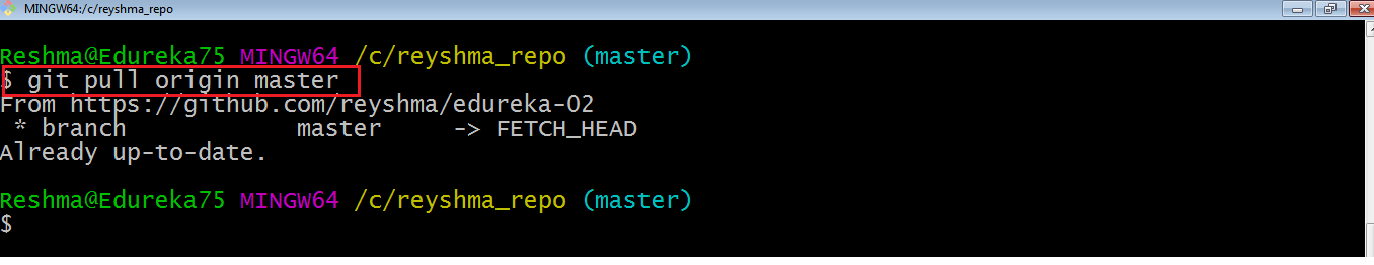
**git remote add origin <link of your central repository>**



Now that my origin is set, let us extract files from the origin using pull. For that use the command:

**git pull origin master**

This command will copy all the files from the master branch of remote repository to your local repository.



Since my local repository was already updated with files from master branch, hence the message is Already up-to-date. Refer to the screen shot above.

***Note:*** One can also try pulling files from a different branch using the following command:

***git pull origin <branch-name>***

Your local Git repository is now updated with all the recent changes. It is time you make changes in the central repository by using the**push** command.

**Push**

This command transfers commits from your local repository to your remote repository. It is the opposite of pull operation.

Pulling imports commits to local repositories whereas pushing exports commits to the remote repositories .

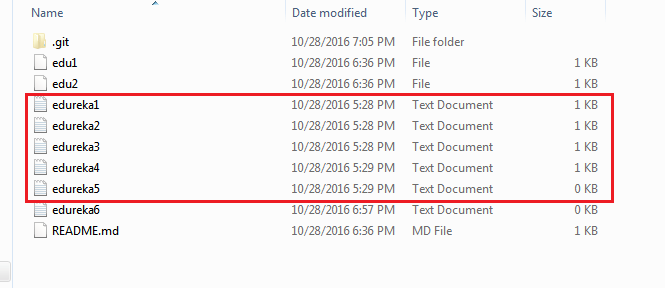
The use of **git push** is to publish your local changes to a central repository. After you’ve accumulated several local commits and are ready to share them with the rest of the team, you can then push them to the central repository by using the following command:

**git push <remote>**

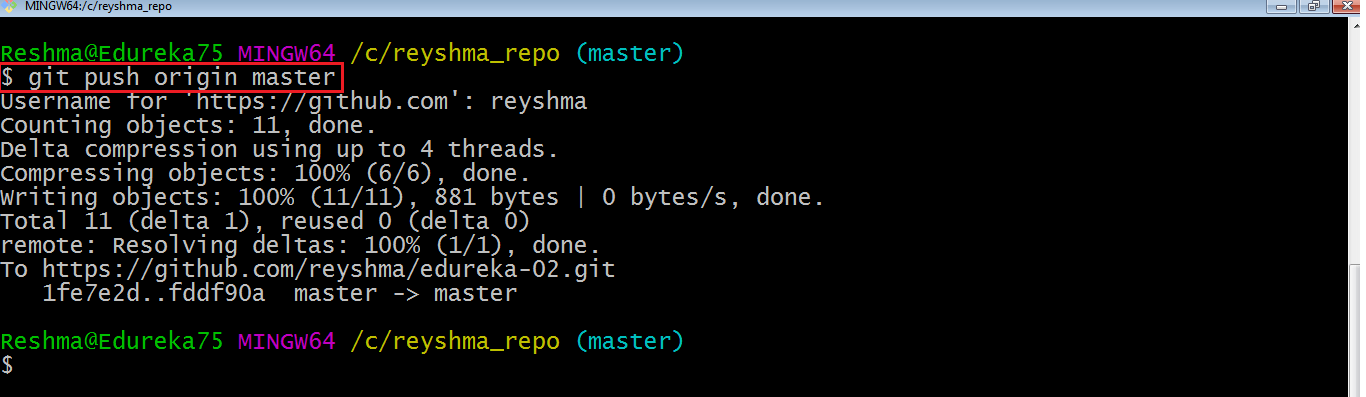
**Note** : This remote refers to the remote repository which had been set before using the pull command.

This pushes the changes from the local repository to the remote repository along with all the necessary commits and internal objects. This creates a local branch in the destination repository.

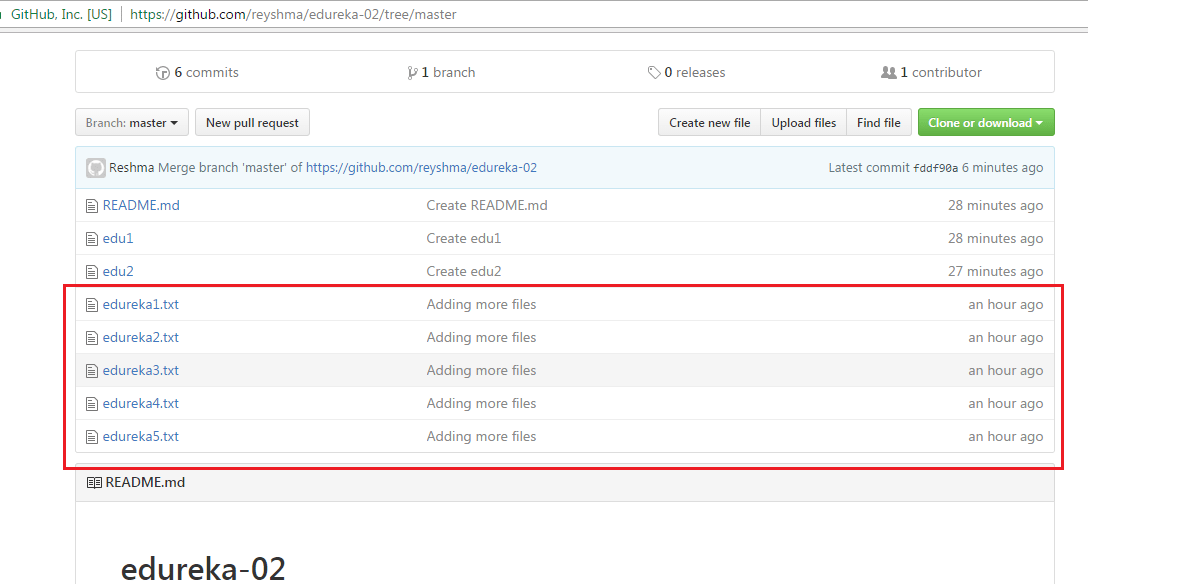
Let me demonstrate it for you.



The above files are the files which we have already committed previously in the commit section and they are all “push-ready“. I will use the command **git push origin master** to reflect these files in the master branch of my central repository.



Let us now check if the changes took place in my central repository.



Yes, it did. :-)

To prevent overwriting, Git does not allow push when it results in a non-fast forward merge in the destination repository.

**Note**: A non-fast forward merge means an upstream merge i.e. merging with ancestor or parent branches from a child branch.

To enable such merge, use the command below:

**git push <remote> –force**

The above command forces the push operation even if it results in a non-fast forward merge.

At this point of this Git Tutorial, I hope you have understood the basic commands of Git. Now, let’s take a step further to learn branching and merging in Git.

## ****Branching****

Branches in Git are nothing but pointers to a specific commit. Git generally prefers to keep its branches as lightweight as possible.

There are basically two types of branches viz. **local branches** and **remote tracking branches**.

A local branch is just another path of your working tree. On the other hand, remote tracking branches have special purposes. Some of them are:

* They link your work from the local repository to the work on central repository.
* They automatically detect which remote branches to get changes from, when you use **git pull**.

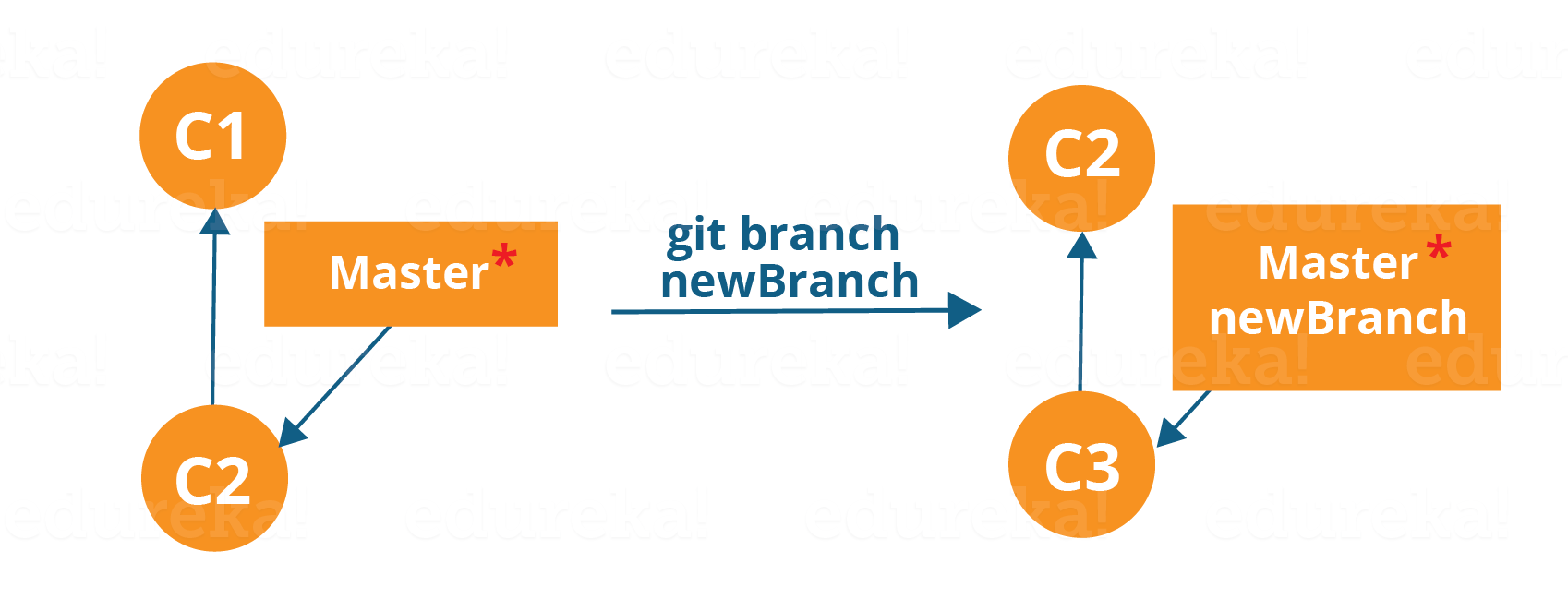
You can check what your current branch is by using the command:

**git branch**

The one mantra that you should always be chanting while branching is “branch early, and branch often”

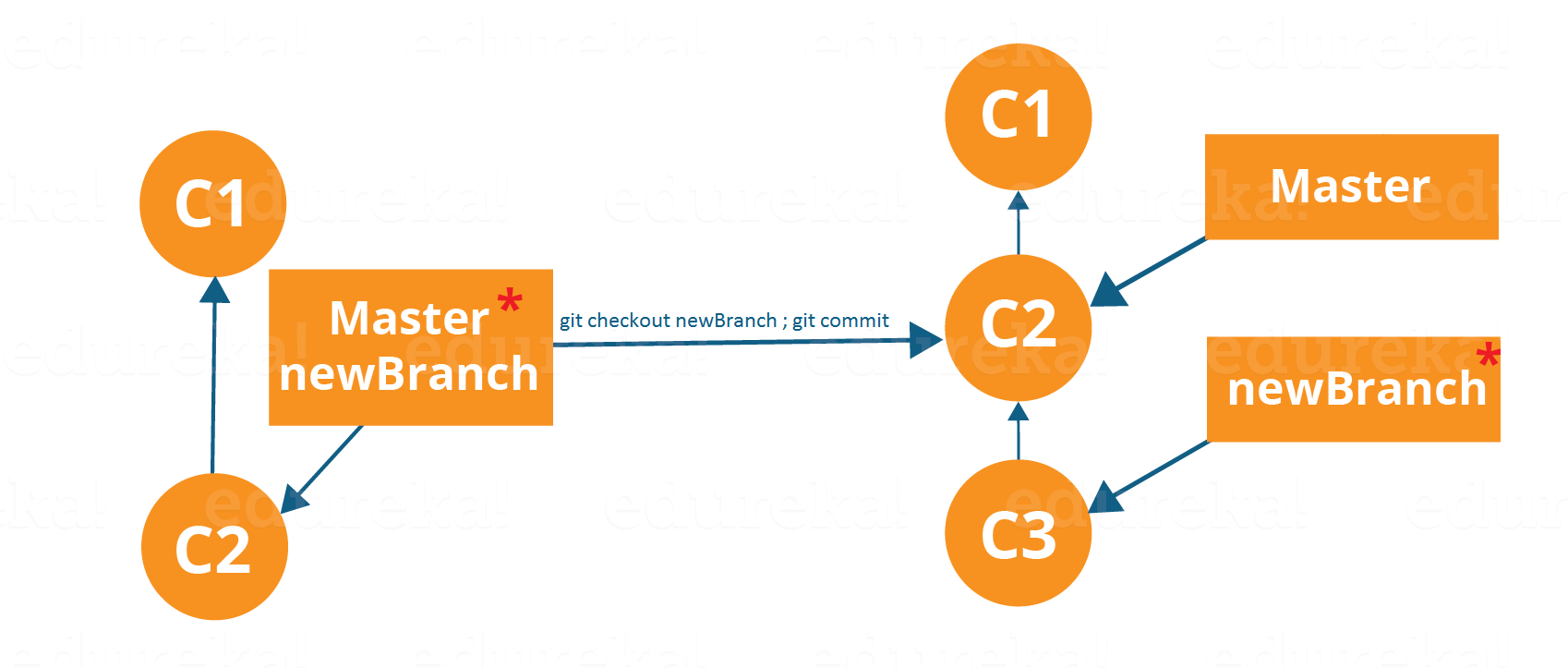
To create a new branch we use the following command:

**git branch <branch-name>**



The diagram above shows the workflow when a new branch is created.  When we create a new branch it originates from the master branch itself.

Since there is no storage/memory overhead with making many branches, it is easier to logically divide up your work rather than have big chunky branches.

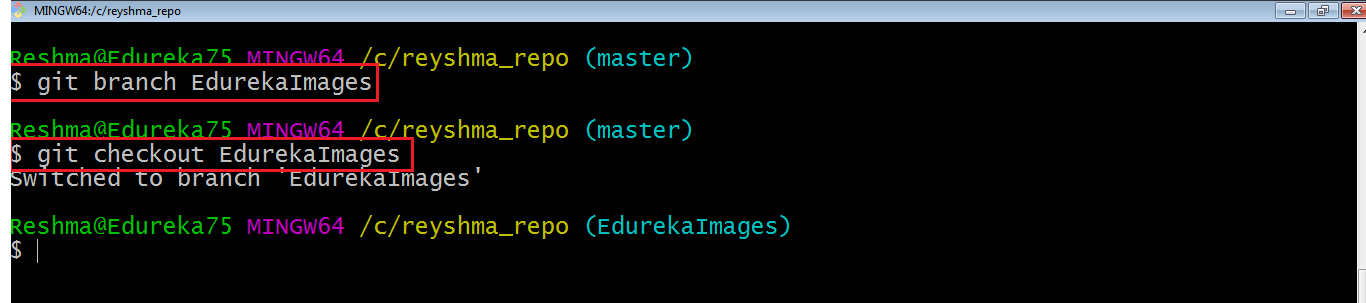
Now, let us see how to commit using branches.

Branching includes the work of a particular commit along with all parent commits. As you can see in the diagram above, the newBranch has detached itself from the master and hence will create a different path.

Use the command below:

**git checkout <branch\_name>**and then

**git commit**



Here, I have created a new branch named “EdurekaImages” and switched on to the new branch using the command **git checkout** .

One shortcut to the above commands is:

**git checkout -b[ branch\_name]**

This command will create a new branch and checkout the new branch at the same time.

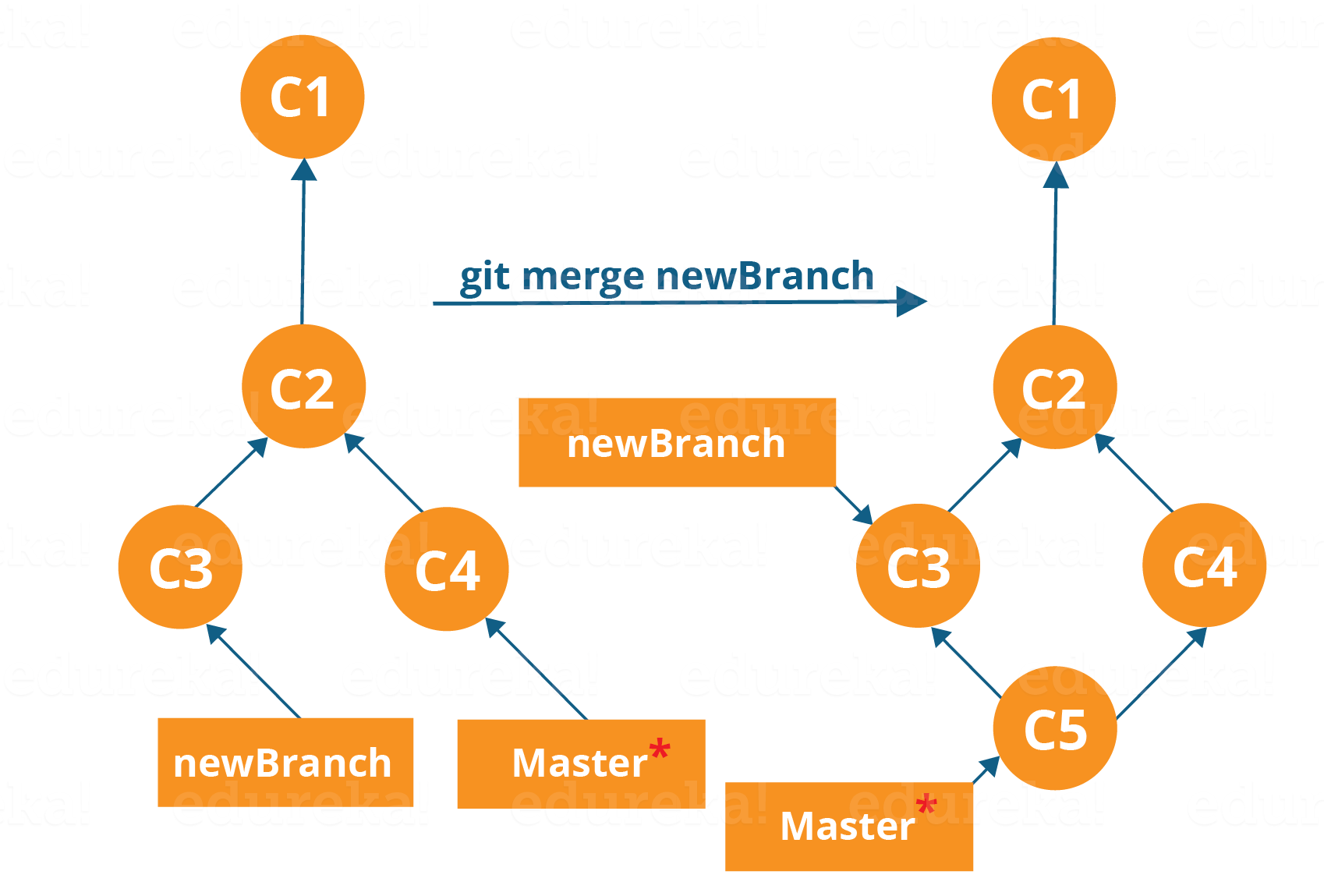
Now while we are in the branch EdurekaImages, add and commit the text file edureka6.txt using the following commands:

**git add edureka6.txt**

**git commit -m”adding edureka6.txt”**

## ****Merging****

Merging is the way to combine the work of different branches together. This will allow us to branch off, develop a new feature, and then combine it back in.



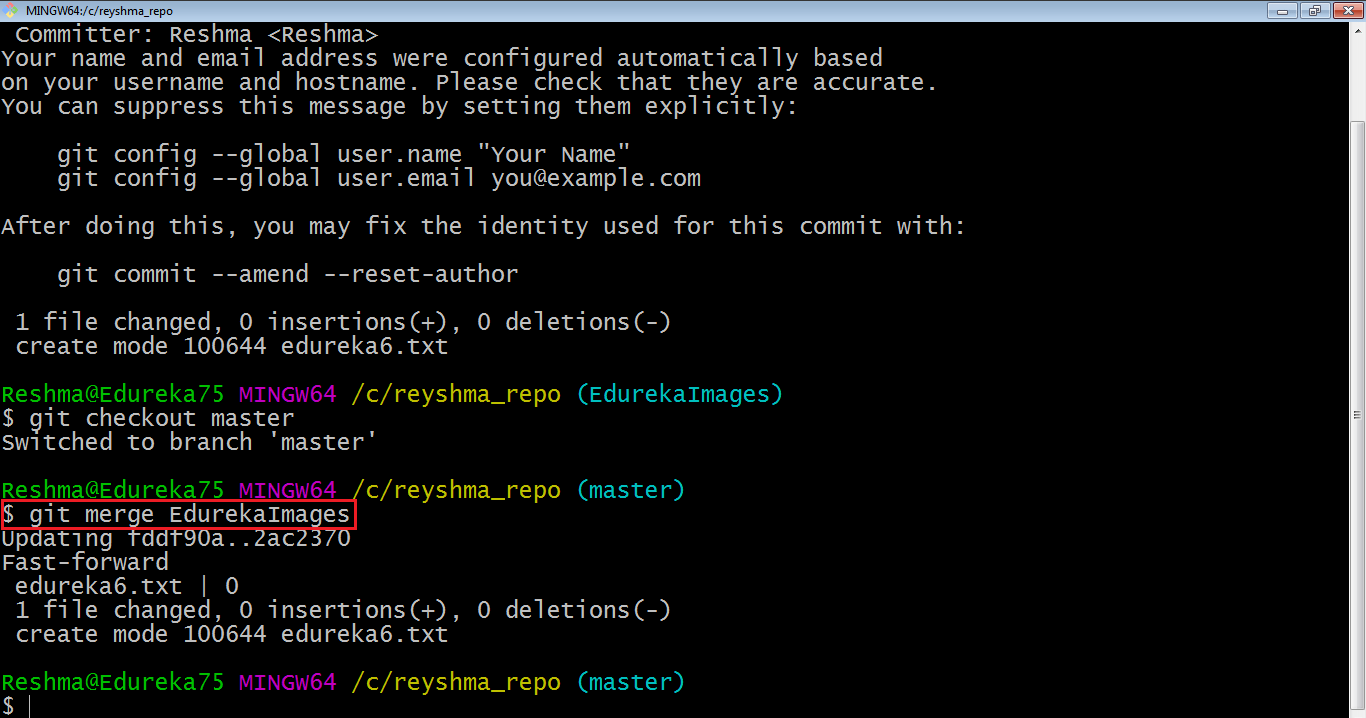
The diagram above shows us two different branches-> newBranch and master. Now, when we merge the work of newBranch into master, it creates a new commit which contains all the work of master and newBranch.

Now let us merge the two branches with the command below:

**git merge <branch\_name>**

It is important to know that the branch name in the above command should be the branch you want to merge into the branch you are currently checking out. So, make sure that you are checked out in the destination branch.

Now, let us merge all of the work of the branch EdurekaImages into the master branch. For that I will first checkout the master branch with the command **git checkout master** and merge EdurekaImages with the command **git merge EdurekaImages**



As you can see above, all the data from the branch name are merged to the master branch. Now, the text file edureka6.txt has been added to the master branch.

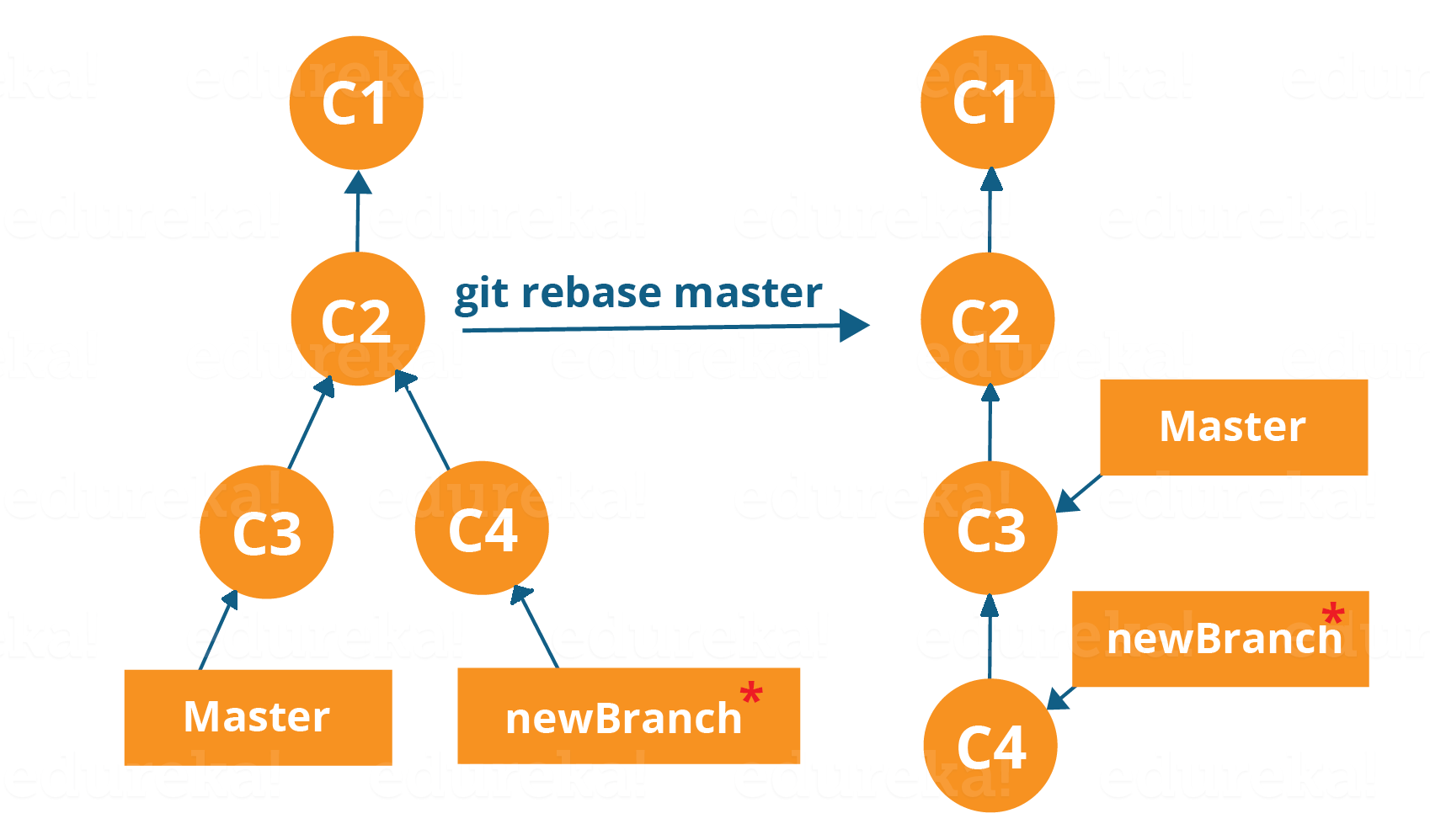
Merging in Git creates a special commit that has two unique parents.

## ****Rebasing****

This is also a way of combining the work between different branches. Rebasing takes a set of commits, copies them and stores them outside your repository.

The advantage of rebasing is that it can be used to make linear sequence of commits. The commit log or history of the repository stays clean if rebasing is done.

Let us see how it happens.

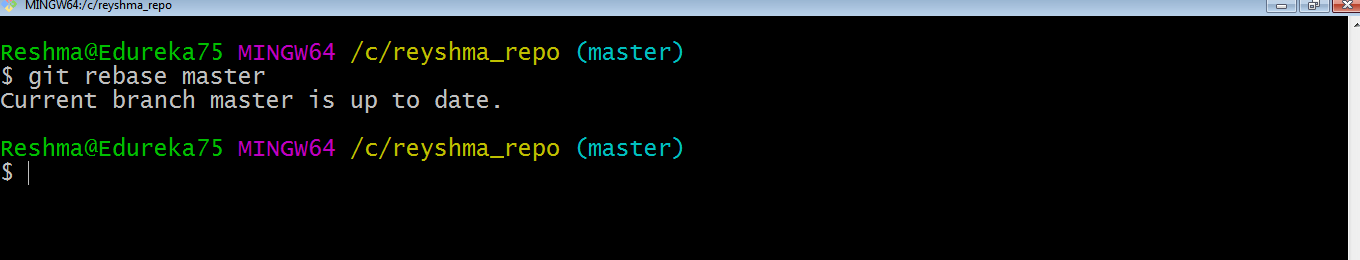


Now, our work from newBranch is placed right after master and we have a nice linear sequence of commits.

**Note**: Rebasing also prevents upstream merges, meaning you cannot place master right after newBranch.

Now, to rebase master, type the command below in your Git Bash:

**git rebase master**



This command will move all our work from current branch to the master. They look like as if they are developed sequentially, but they are developed parallelly.

## ****Git Tutorial – Tips And Tricks****

Now that you have gone through all the operations in this Git Tutorial, here are some tips and tricks you ought to know. :-)

* **Archive your repository**

Use the following command-

**git archive master –format=zip  –output= ../name-of-file.zip**

It stores all files and data in a zip file rather than the **.git** directory.

Note that this creates only a single snapshot omitting version control completely. This comes in handy when you want to send the files to a client for review who doesn’t have Git installed in their computer.

* **Bundle your repository**

It turns a repository into a single file.

Use the following command-

**git bundle create ../repo.bundler master**

This pushes the master branch to a remote branch, only contained in a file instead of a repository.

An alternate way to do it is:

**cd..**

**git clone repo.bundle repo-copy -b master**

**cd repo-copy**

**git log**

**cd.. /my-git-repo**

* **Stash uncommitted changes**

When we want to undo adding a feature or any kind of added data temporarily, we can “stash” them temporarily.

Use the command below:

**git status**

**git stash**

**git status**

And when you want to re-apply the changes you “stash”ed ,use the command below:

**git stash apply**

# Install Git – Git Installation On Windows And CentOS

**Install Git On Windows**

**Step 1**:

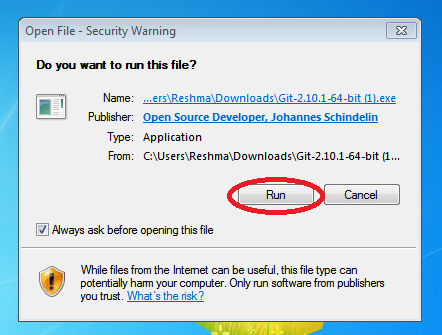
To download the latest version of Git, click on the link below:

[***Download Git for Windows***](https://git-scm.com/download/win/)

Great! Your file is being downloaded.

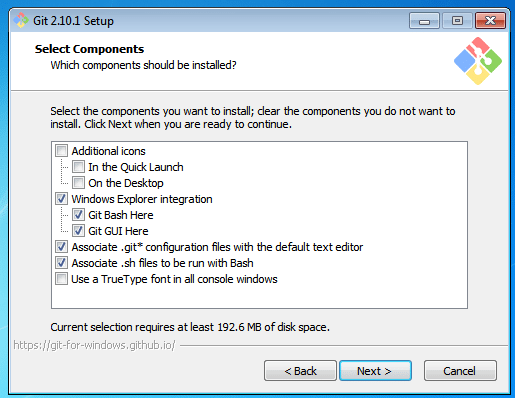
**Step 2:**

After your download is complete, **Run** the .exe file in your system.



**Step 3:**

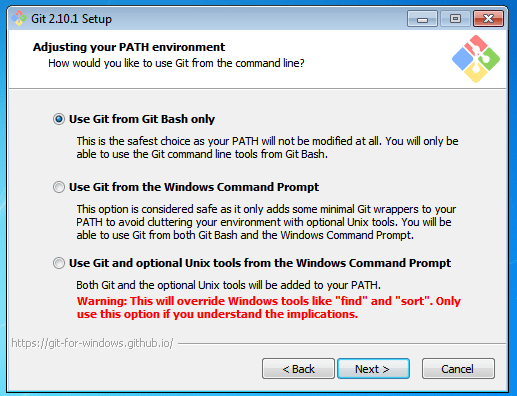
After you have pressed the **Run** button and agreed to the license, you will find a window prompt to select components to be installed.



After you have made selection of your desired components, click on **Next>**.

**Step 4:**

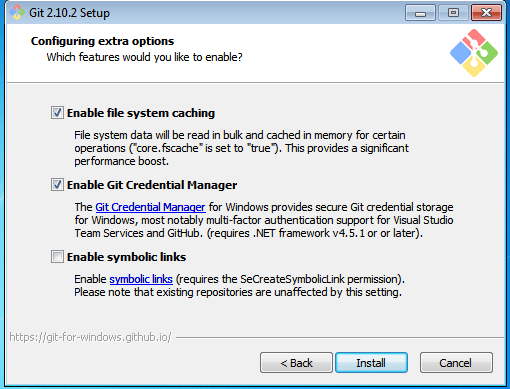
The next prompt window will let you choose the adjustment of your path environment. This is where you decide how do you want to use Git.



You can select any of the three options according to your needs. But for beginners, I recommend using **Use Git From Git Bash Only**

**Step 5:**

The next step is to choose features for your Git. You get three options and you can choose any of them, all of them or none of them as per your needs. Let me tell you what these features are:



The first is the option to **Enable file system caching**.

Caching is enabled through Cache manager, which operates continuously while Windows is running. File data in the system file cache is written to the disk at intervals determined by the operating system, and the memory previously used by that file data is freed.

The second option is to enable **Git Credential Manager**.

The **Git Credential Manager** for Windows (GCM) is a credential helper for Git. It securely stores your credentials in the Windows CM so that you only need to enter them once for each remote repository you access. All future Git commands will reuse the existing credentials.

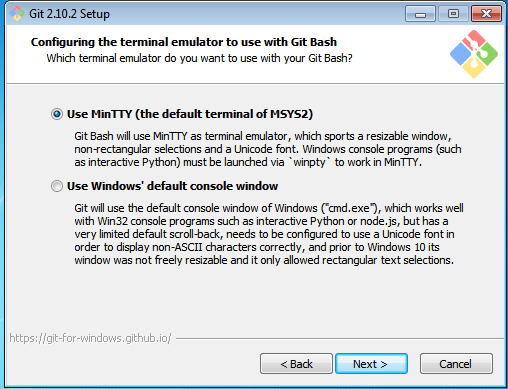
The third option is to **Enable symbolic links**.

Symbolic links or symlinks are nothing but advanced shortcuts. You can create symbolic links for each individual file or folder, and these will appear like they are stored in the folder with symbolic link.

I have selected the first two features only.

**Step 6:**

Choose your terminal.



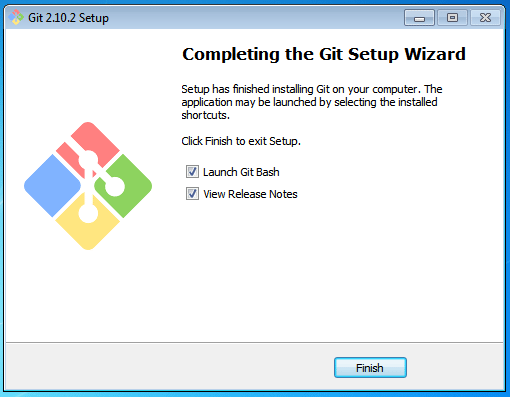
You can choose one from the options.

The default terminal of MYSYS2 which is a collection of GNU utilities like bash, make, gawk and grep to allow building of applications and programs which depend on traditionally UNIX tools to be present.

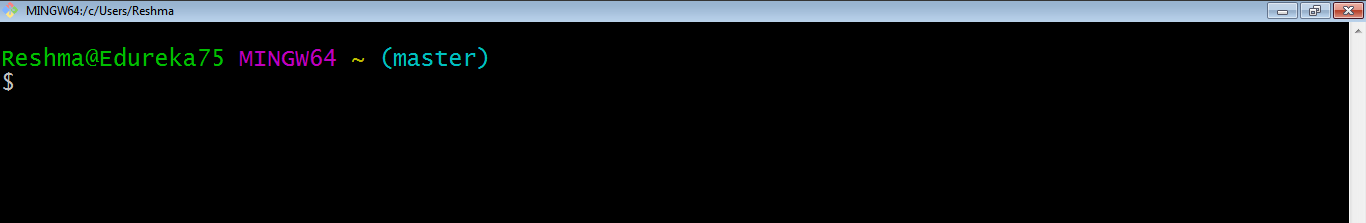
Or you can choose the window’s default console window (cmd.exe).

**Step 7:**

Now you have got all you need. Select **Launch Git Bash** and click on **Finish**.



This will launch Git Bash on your screen which looks like the snapshot below:

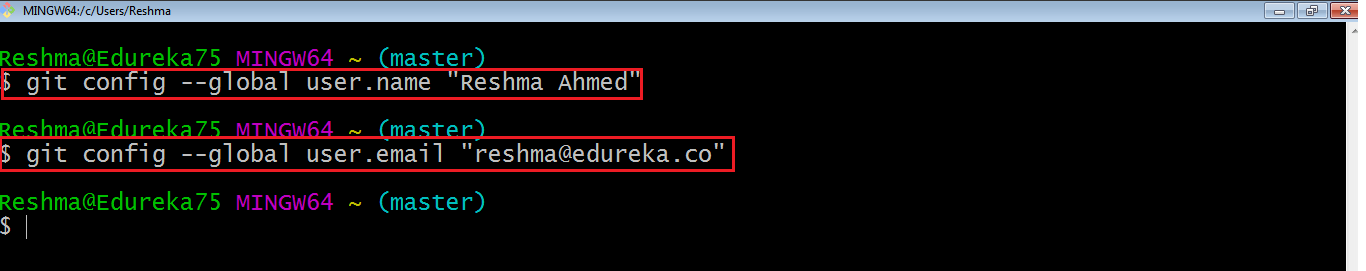


**Step 8:**

Let us proceed with configuring Git with your username and email. In order to do that, type the following commands in your Git Bash:

git config - - global user.name "<your name>"

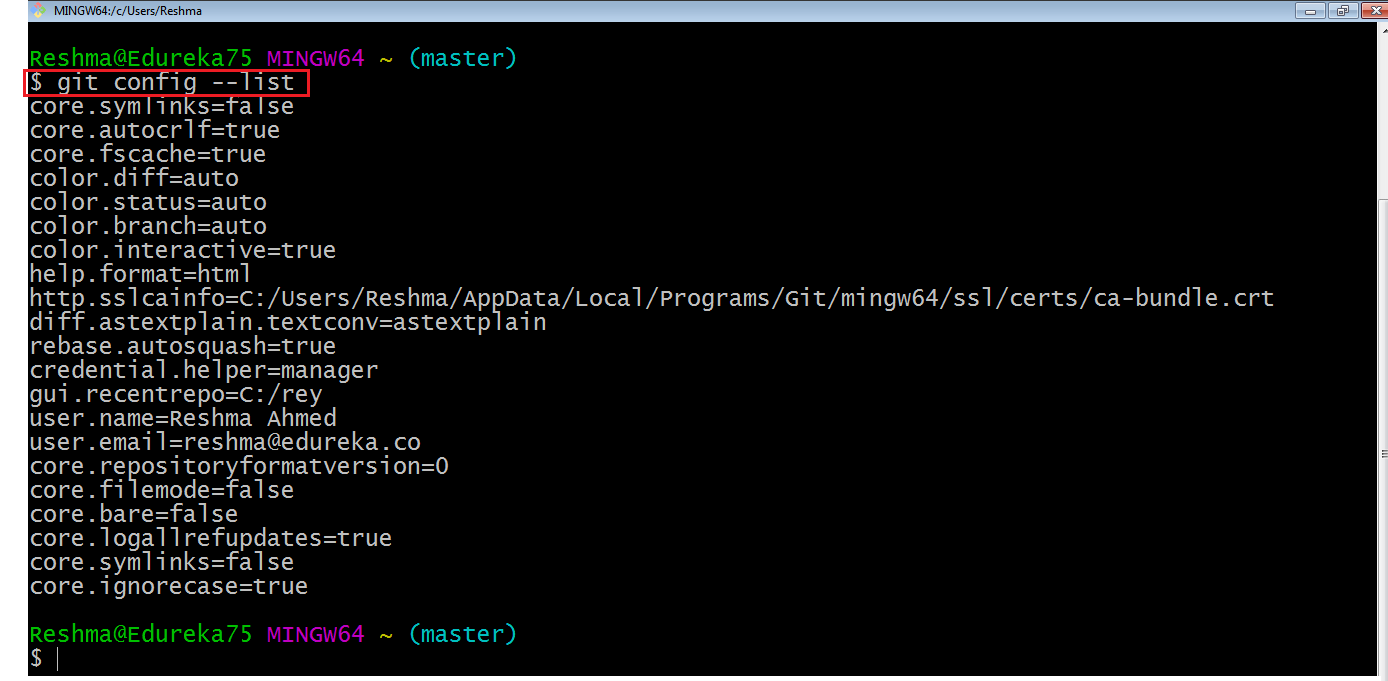
git config - - global user.email "<your email>"



It is important to configure your Git because any commits that you make are associated with your configuration details.

If you want to view all your configuration details, use the command below:

git config - - list

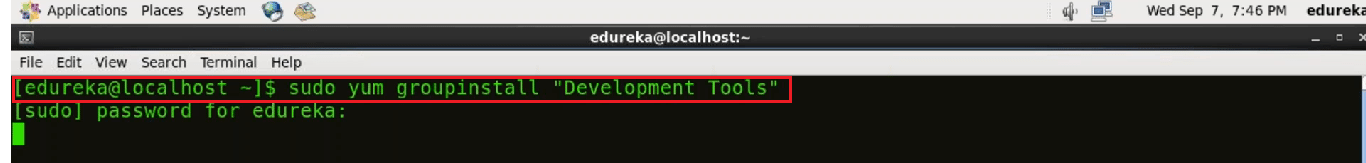


**Install Git on CentOS**

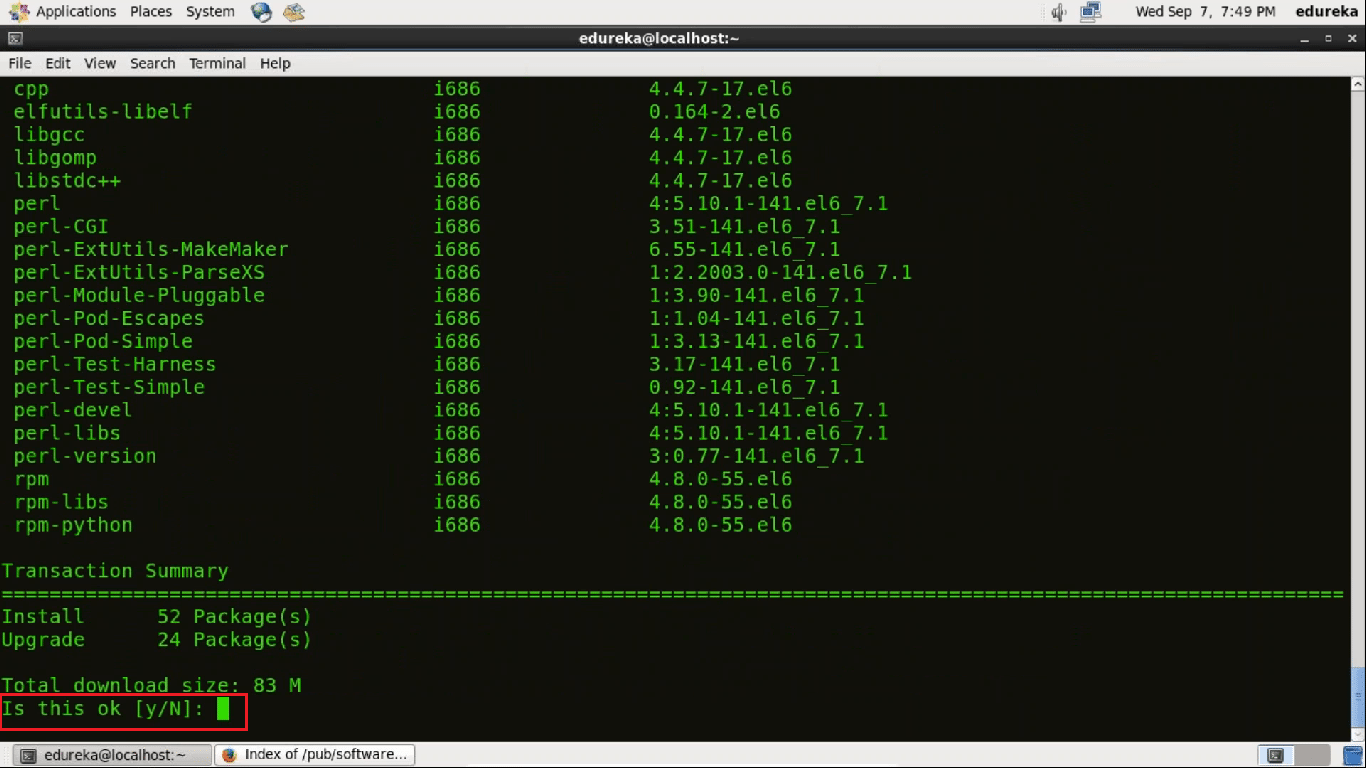
**Step 1:**  
First we need to install the software that Git depends on. These dependencies are all available in default CentOS repository.

Use the command:

sudo yum groupinstall "Development Tools"



It will ask for your confirmation to download the tools.

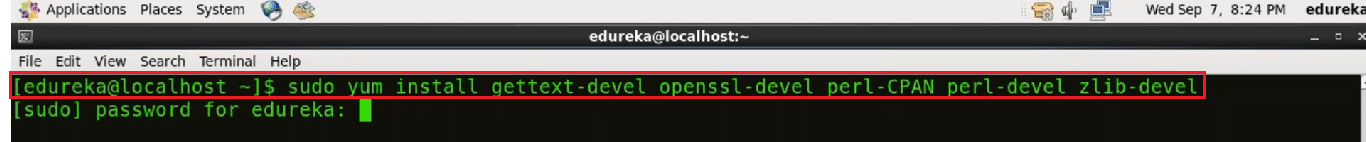


Press **Y** for Yes.

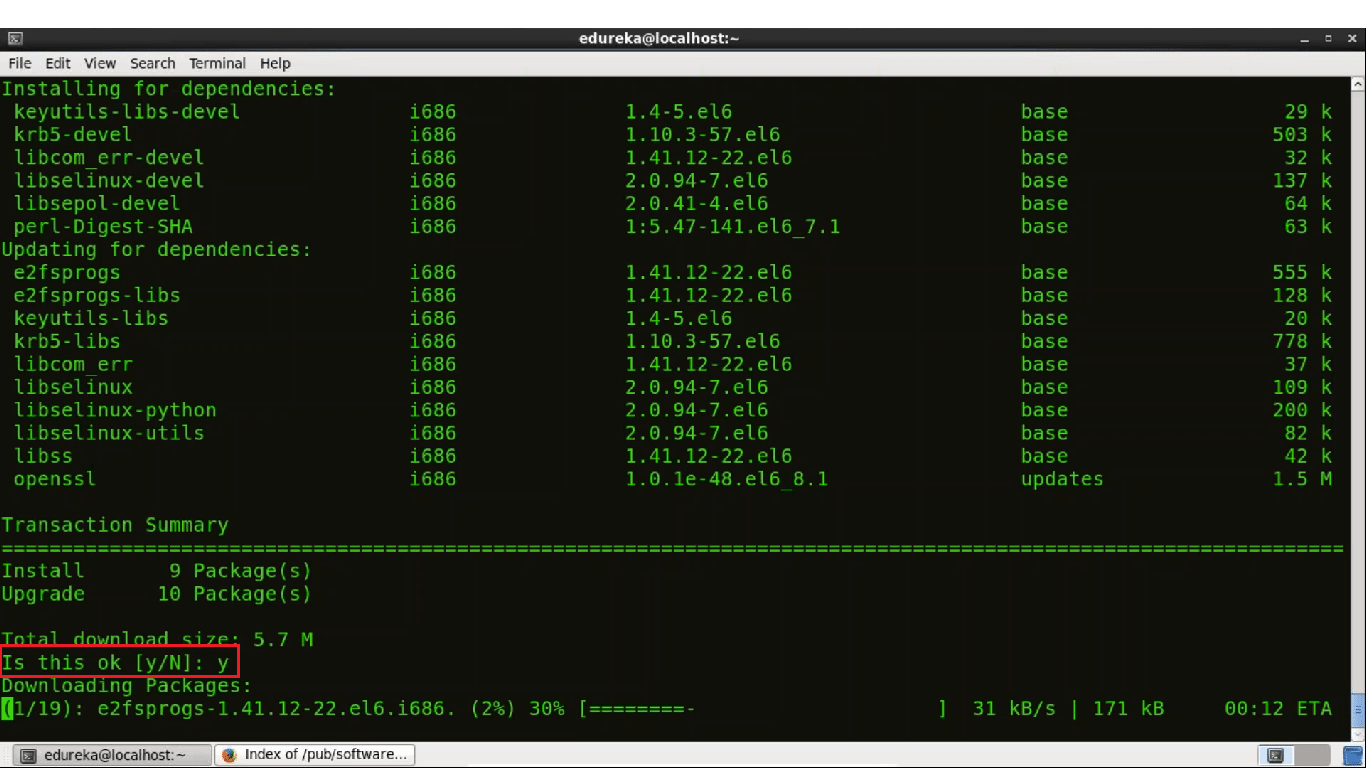
The “Development tools” which is a ***yum group***, is a predefined bundle of software that can be installed at once, instead of having to install each application separately. The Development tools will allow you to build and compile software from source code.

Now use the command:

sudo yum install gettext-devel openssl-devel perl-CPAN perl-devel zlib-devel



Enter your password. It will ask for your confirmation to download the package.

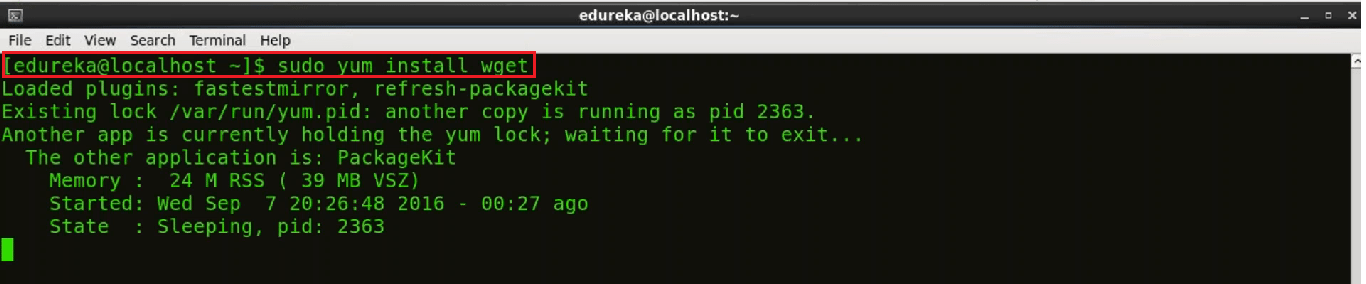


Press **y**.

Now we are ready with the prerequisites. Lets proceed towards Git installation.

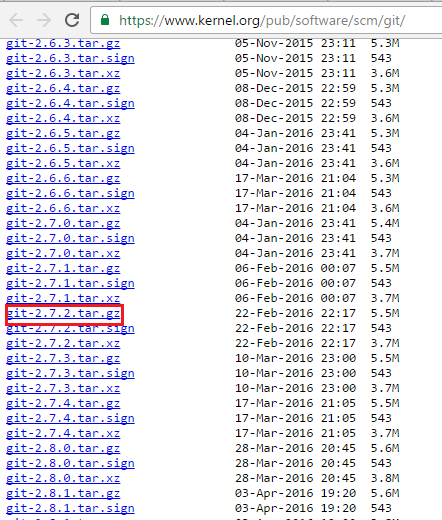
**Step 2:**

Now we are going to use **wget** command to download a specific version of Git.



 But first we need to copy the link of the version that we want to install. For that go to this [website](https://www.kernel.org/pub/software/scm/git/).

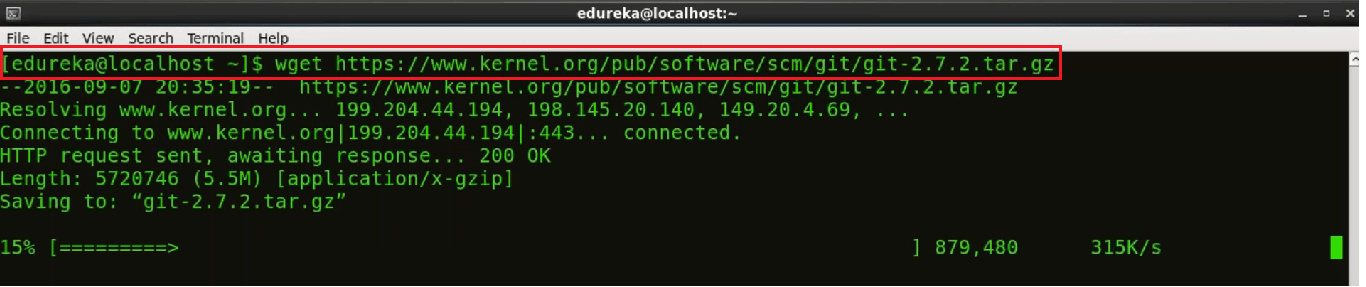
You will find the following webpage:



I am downloading git-2.7.2.tar.gz version of Git.

Now use the **wget** command with the link of the Git version you have chosen to install. Use the command below:

wget https://github.com/git/git/archive/v2.7.2.tar.gz -O git.tar.gz

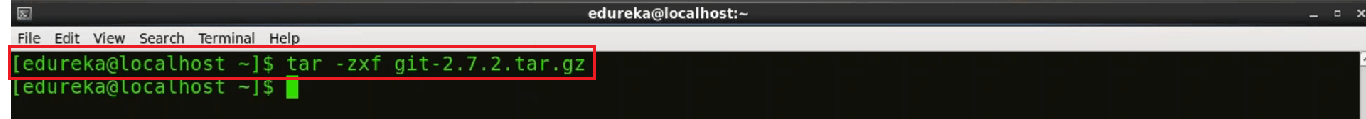


This downloaded file will be available in my directory.

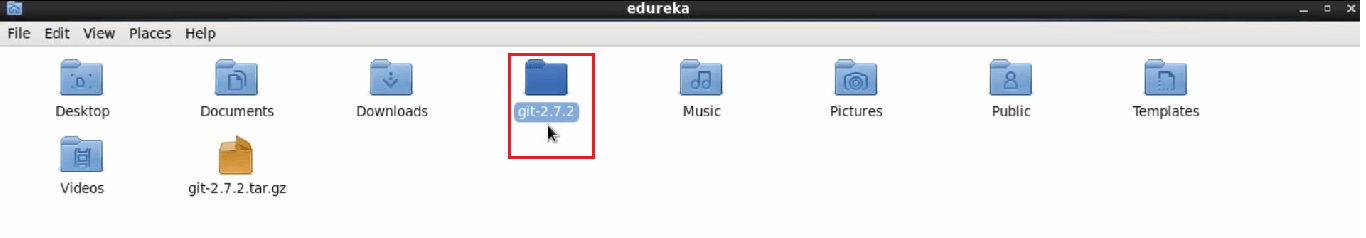
**Step 3:**

Once the download is complete we will extract the file from the downloaded Git Tar file. For that we will use Tar command.

tar -zxf git.tar.gz



Lets see the extracted folder.



**Step 4:**

Now lets change the directory to Git.

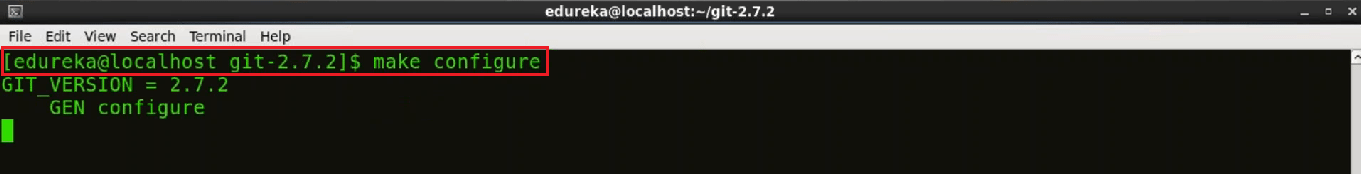
Use the command **cd git**



**Step 5:**

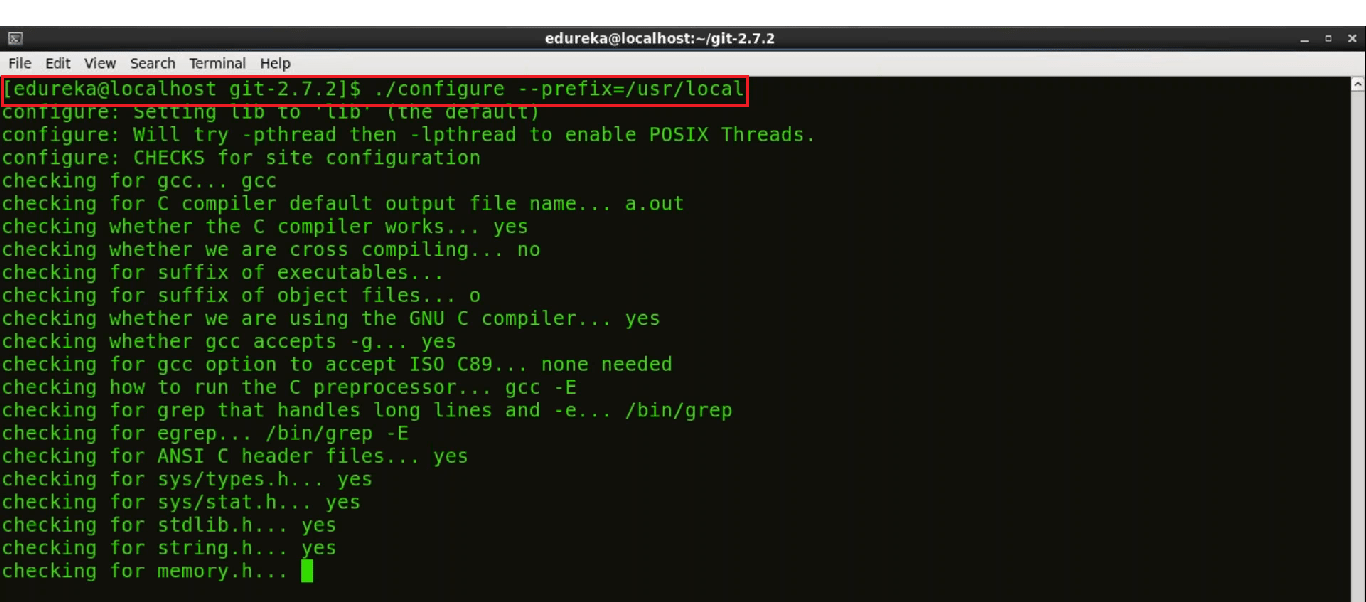
We are in the source folder we can begin the source build process. For that first type in the command:

make configure



Now use the following command:

./configure --prefix=/usr/local

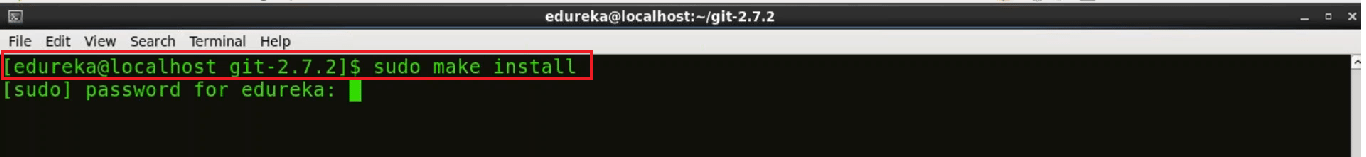


The configure script is responsible for getting ready to build the software on your specific system. It makes sure all of the dependencies for the rest of the build and install process are available once configure has done its job, we can invoke make to build the software.

**Step 6:**

Now that the software is built and ready to run, the files can be copied to their final destinations. Use the command below:

sudo make install

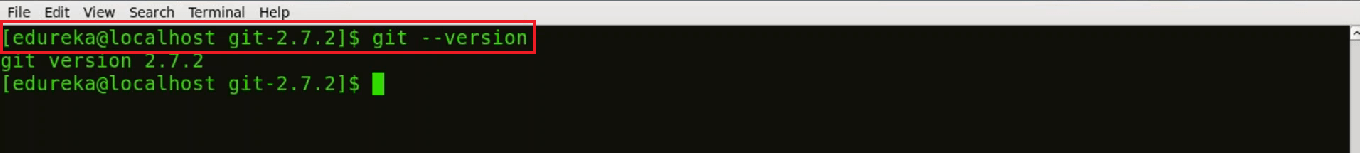


The make install command will copy the built program, and its libraries and documentation, to the correct locations.

**Step 7:**

Now to check the version of Git installed  we will use the command:

git --version

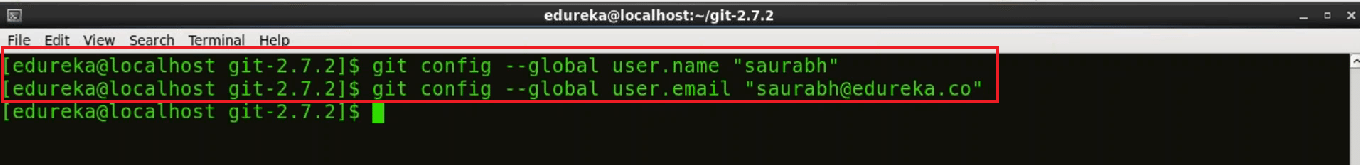


**Step 8:**

Before we go ahead you need to submit some information about yourself so that commit messages will be generated with the correct information attached.  
We need to provide Name and Email address that we would like to embed into our commits, to do that we will use following commands:

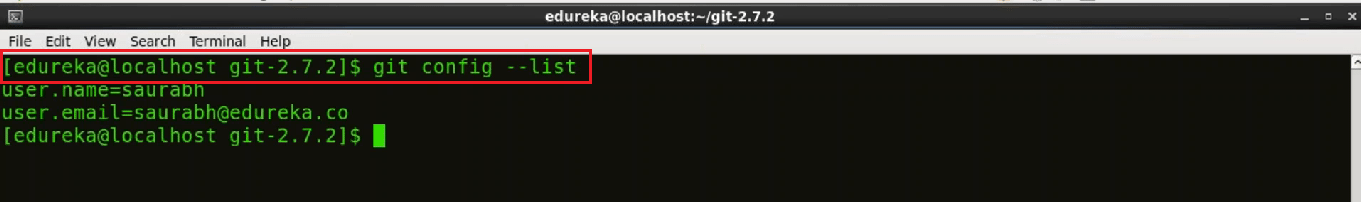
git config --global user.name "Your Name"

git config --global user.email "you@example.com"



To confirm that these configurations are added successfully we will use the command:

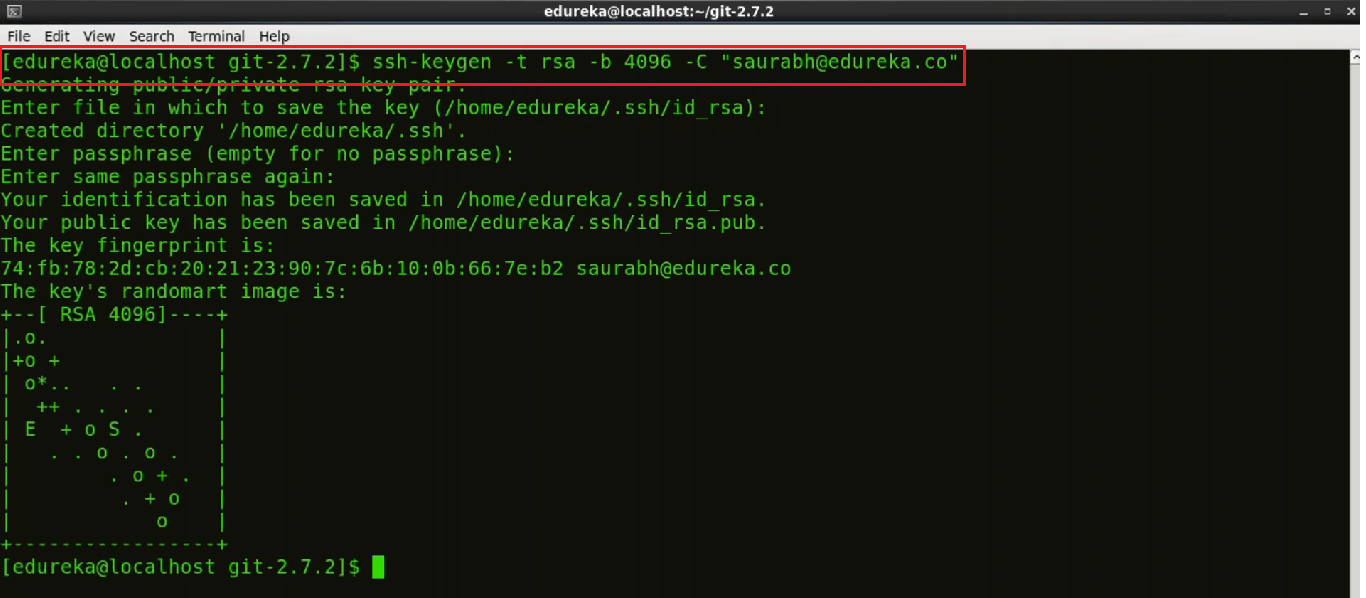
**git config --list**



**Step 9:**

Now we need to generate a **SSH** key.  
**SSH** is a secure protocol used as the primary means of connecting to Linux servers remotely. Now to generate a new SSH key we will use:

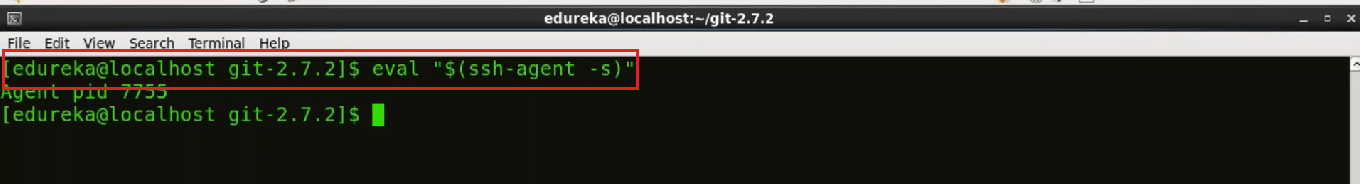
ssh-keygen -t rsa -b 4096 -C "your\_email@example.com"



It will ask you to enter the file name where you want to save the key. If you want it saved in your default directory press ‘Enter’. Enter blank passphrase if you want to and then enter the same again.

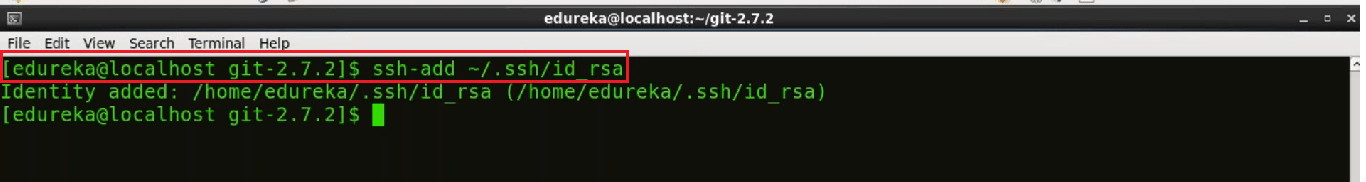
There is a program called **ssh-agent**that runs the duration of a local login session. It stores unencrypted keys in memory, and communicates with SSH clients using a Unix domain socket. So to ensure that SSH agent is enabled we will use this command below:

eval "$(ssh-agent -s)"



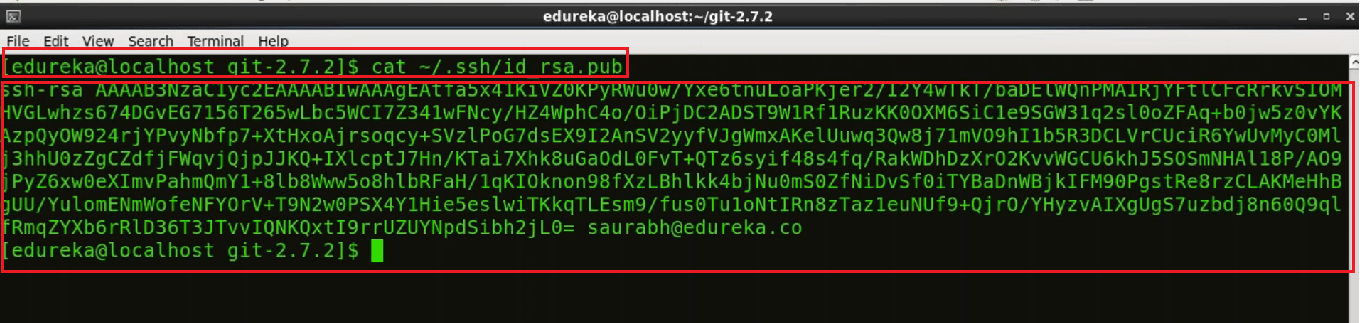
To add **SSH** key to the **SSH** agent we will use

ssh-add ~/.ssh/id\_rsa



To add **SSH** key to our GitHub account we will use:

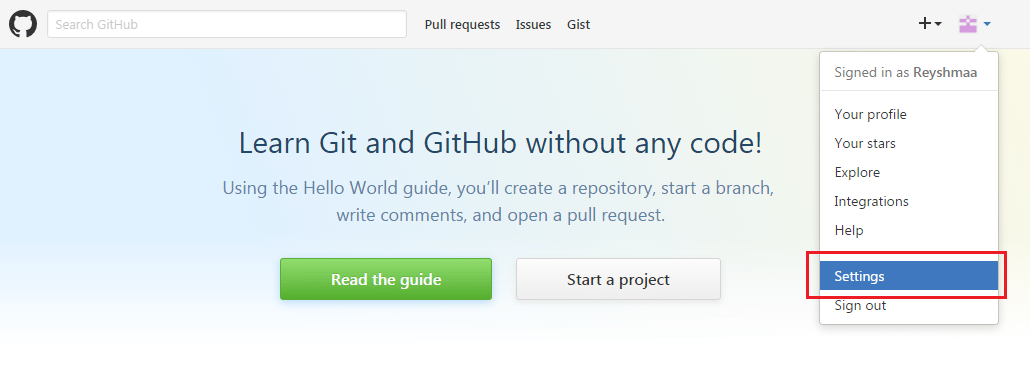
cat ~/.ssh/id\_rsa.pub



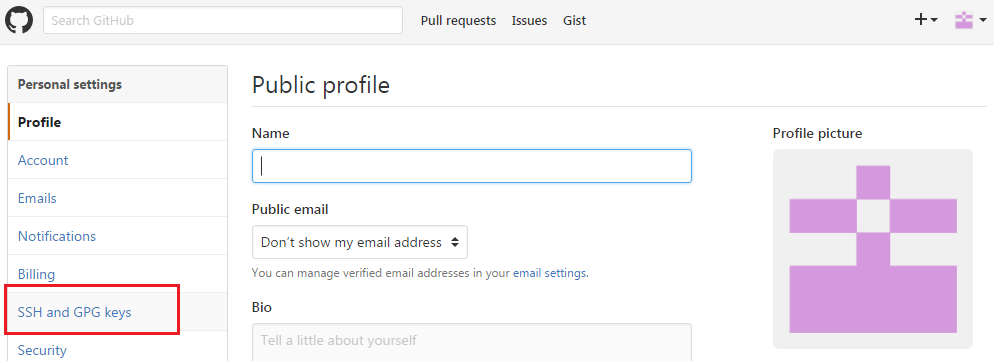
The gibberish you see on screen is actually the **SSH** key. ;-)

Finally we need to copy the **SSH** key and then we need to go to the GitHub account and click on settings.

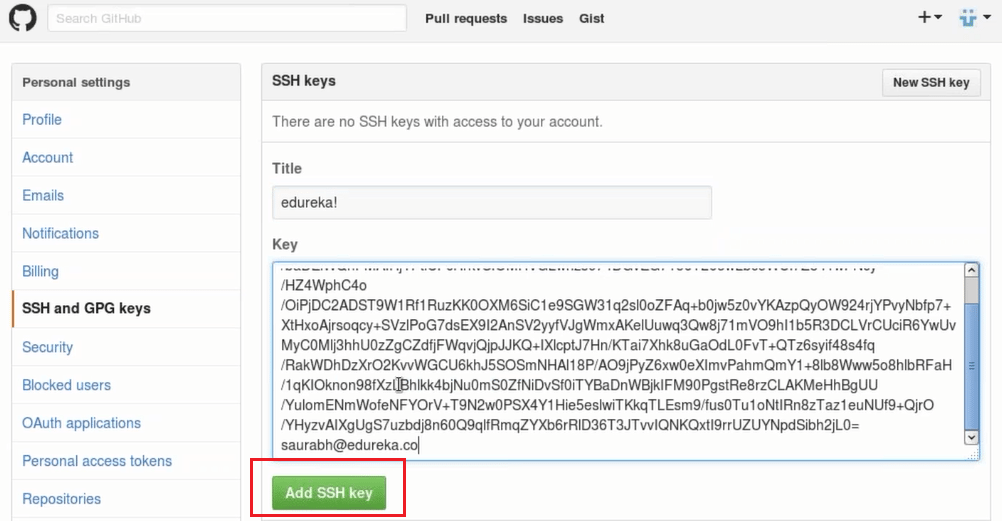
(P.S. If you don’t have a GitHub repository and want to learn how to create it , skip[here](https://www.edureka.co/blog/install-git/#github) )



and then go to **SSH** and **GPG** keys option on the left.

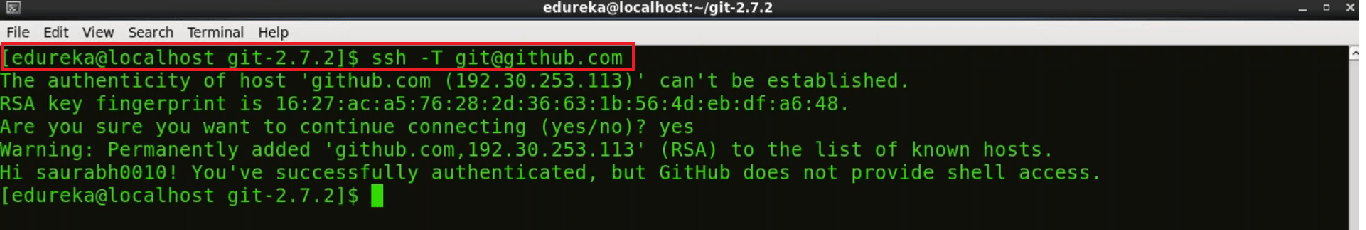


We will now click on **New SSH** key and add title to it  and then paste the copied key in the space provided. Now we will click on **add SSH key**

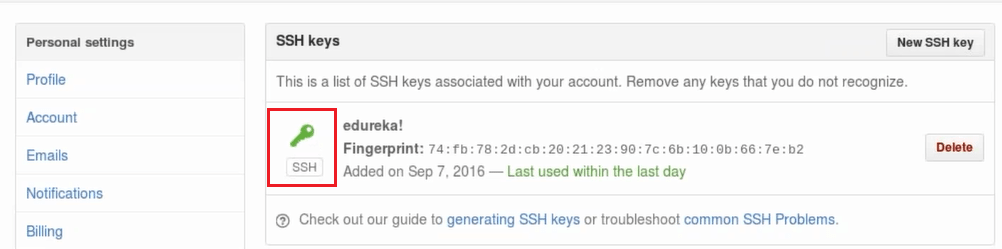


Now use the below command to test the **SSH key**:

ssh -T git@github.com



Now we can see in the snapshot below, that the colour of the key is green. It means we have successfully tested the key.



This is how you install Git and connect to your central repository on Git.

**Create GitHub Repositories**

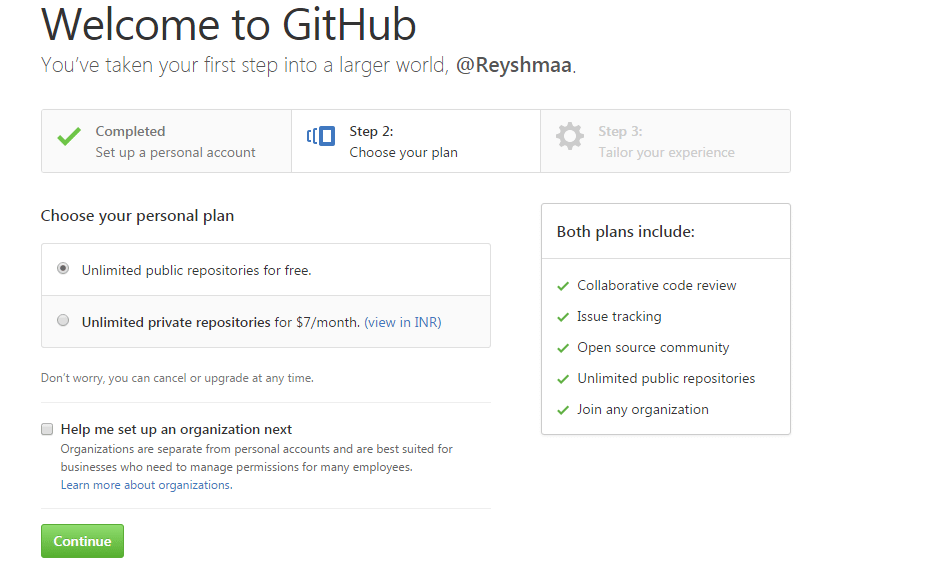
You have learnt to install Git in your system and now its time to make repositories on GitHub that will act as your remote repository.

**Step 1:**

Go to “**www.github.com**” and like a piece of cake, all you need to do to Sign Up is fill up the following form and click on **Sign Up**.

**Step 2:**

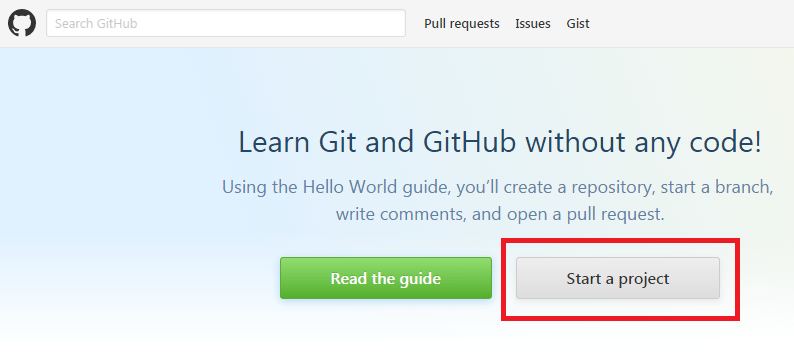
Choose if you want your repositories to be private or public.



After choosing your plan, click on **Continue**

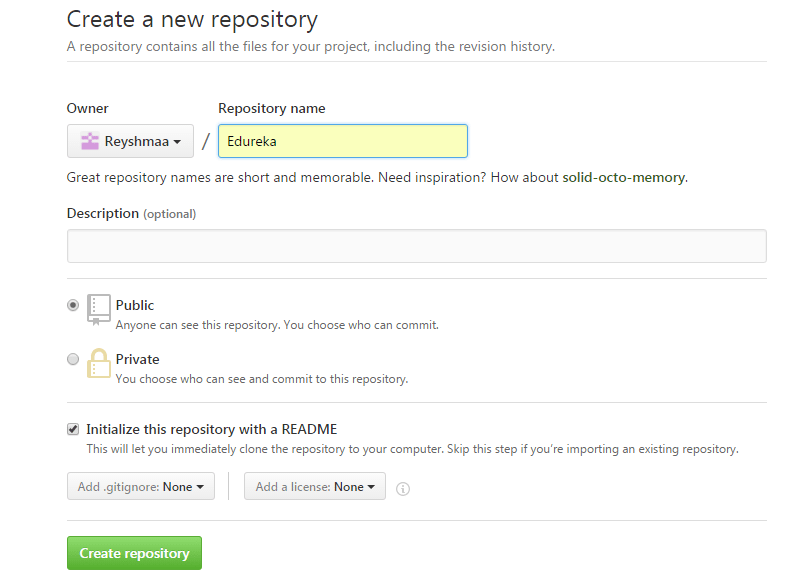
**Step 3:**

Confirm your email and then click on **Start a project**.

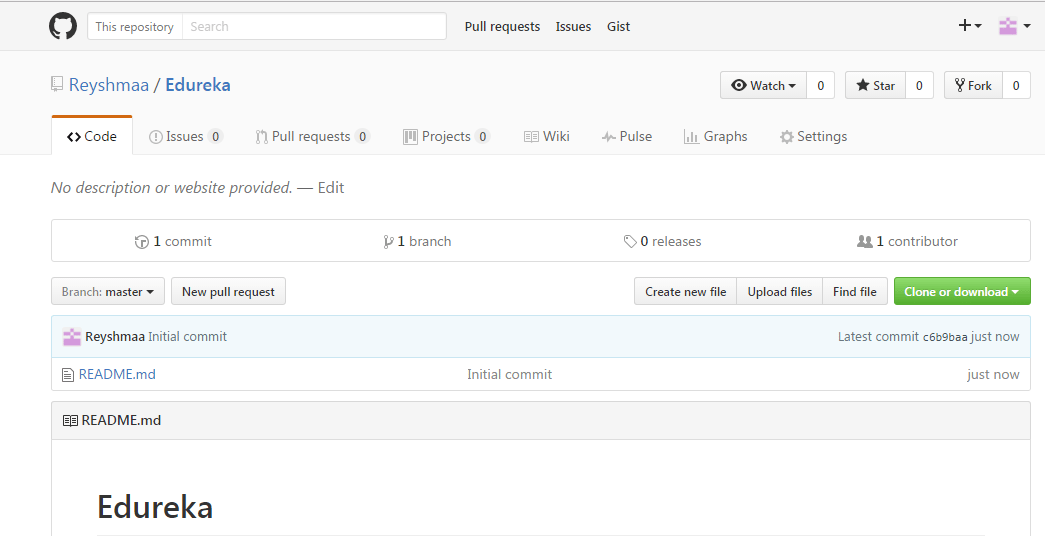


**Step 4:**

Name your repository and click on **Create repository**.



Your repository will look like this snapshot below:



# Top 20 Git Commands with Example

**Git Commands**

**git config**

**Usage: git config –global user.name “[name]”**

**Usage: git config –global user.email “[email address]”**

This command sets the author name and email address respectively to be used with your commits.

Git Config Command - Git Commands - Edureka

**git init**

**Usage: git init [repository name]**

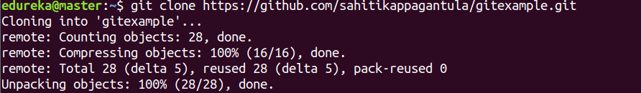
This command is used to start a new repository.

GitInit Command - Git Commands - Edureka

**git clone**

**Usage: git clone [url]**

This command is used to obtain a repository from an existing URL.



**git add**

**Usage: git add [file]**

This command adds a file to the staging area.

Git Add Command - Git Commands - Edureka

**Usage: git add \***

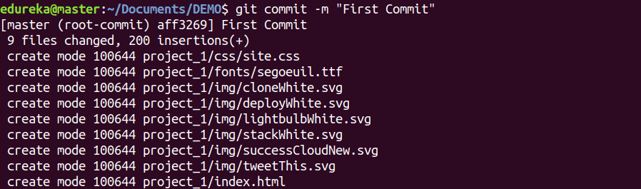
This command adds one or more to the staging area.

Git Add Command - Git Commands - Edureka

**git commit**

**Usage: git commit -m “[ Type in the commit message]”**

This command records or snapshots the file permanently in the version history.



**Usage: git commit -a**

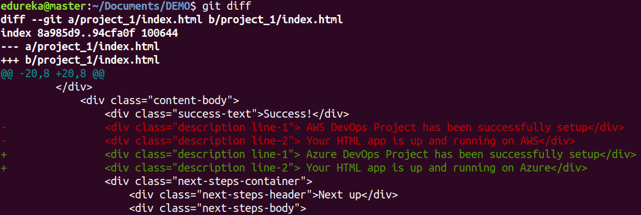
This command commits any files you’ve added with the git add command and also commits any files you’ve changed since then.

Git Commit Command - Git Commands - Edureka

**git diff**

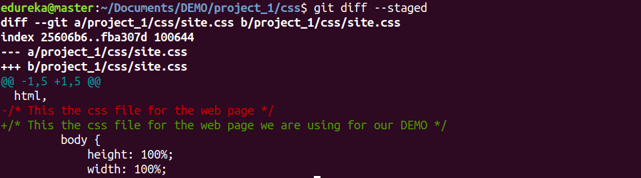
**Usage: git diff**

This command shows the file differences which are not yet staged.



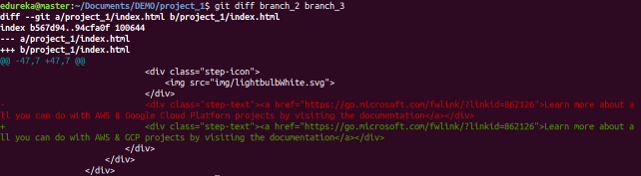
**Usage: git diff –staged**

This command shows the differences between the files in the staging area and the latest version present.



**Usage: git diff [first branch] [second branch]**

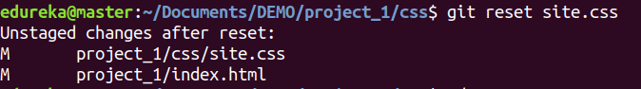
This command shows the differences between the two branches mentioned.



**git reset**

**Usage: git reset [file]**

This command unstages the file, but it preserves the file contents.



**Usage: git reset [commit]**

This command undoes all the commits after the specified commit and preserves the changes locally.

Git Reset Command - Git Commands - Edureka

**Usage: git reset –hard [commit]**

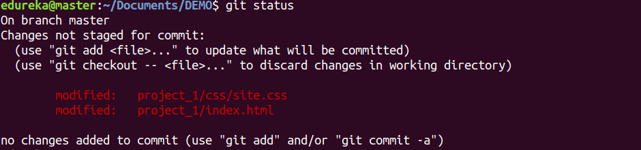
This command discards all history and goes back to the specified commit.

Git Reset Command - Git Commands - Edureka

**git status**

**Usage: git status**

This command lists all the files that have to be committed.



**git rm**

**Usage: git rm [file]**

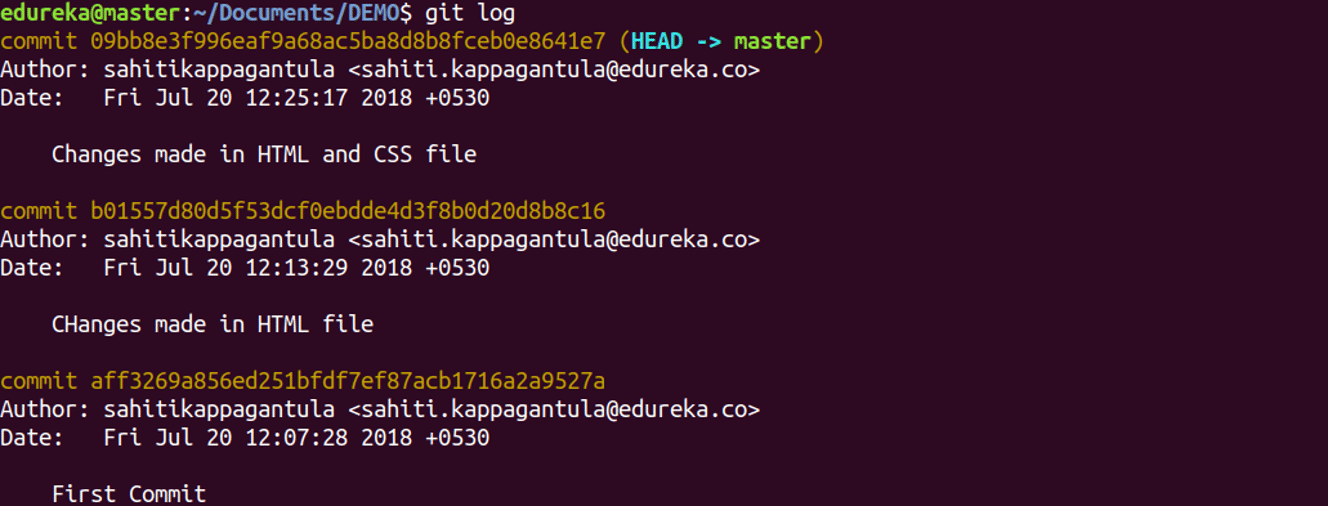
This command deletes the file from your working directory and stages the deletion.

Git Rm Command - Git Commands - Edureka

**git log**

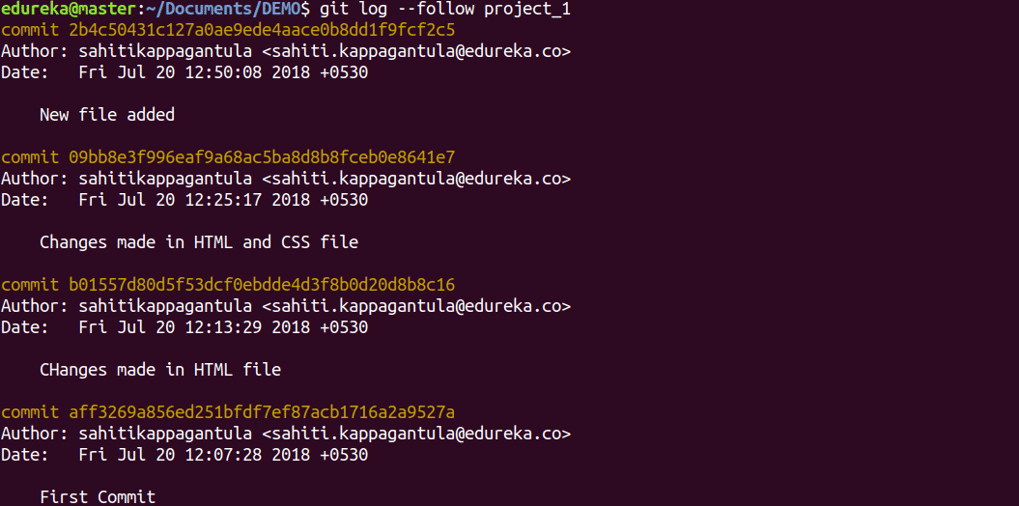
**Usage: git log**

This command is used to list the version history for the current branch.



**Usage: git log –follow[file]**

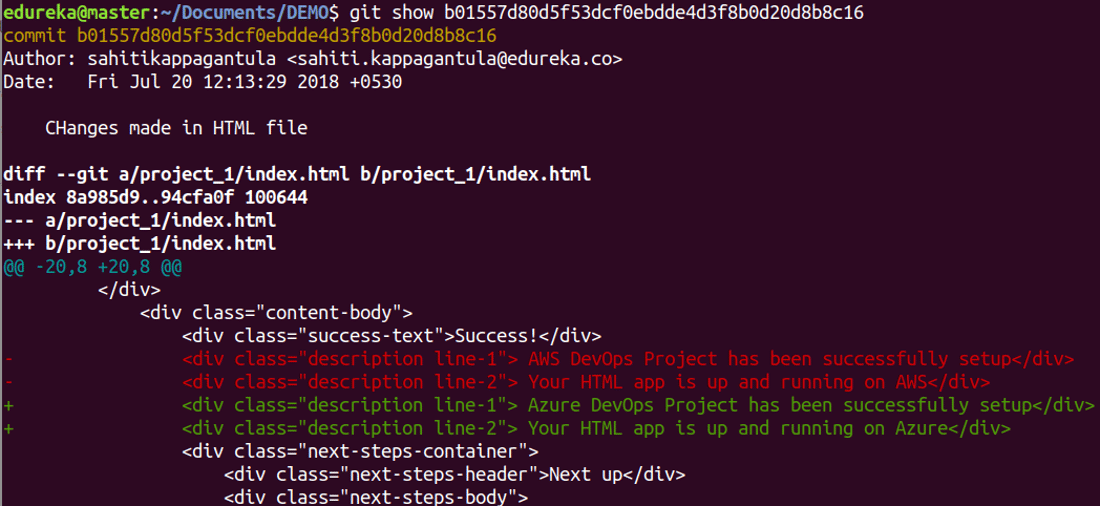
This command lists version history for a file, including the renaming of files also.



**git show**

**Usage: git show [commit]**

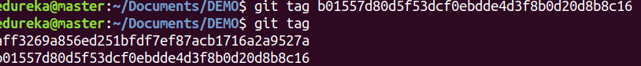
This command shows the metadata and content changes of the specified commit.



**git tag**

**Usage: git tag [commitID]**

This command is used to give tags to the specified commit.



**git branch**

**Usage: git branch**

This command lists all the local branches in the current repository.

Git Branch Command - Git Commands - Edureka

**Usage: git branch [branch name]**

This command creates a new branch.

Git Branch Command - Git Commands - Edureka

**Usage: git branch -d [branch name]**

This command deletes the feature branch.

Git Branch Command - Git Commands - Edureka

**git checkout**

**Usage: git checkout [branch name]**

This command is used to switch from one branch to another.

Git Checkout Command - Git Commands - Edureka

**Usage: git checkout -b [branch name]**

This command creates a new branch and also switches to it.

Git Checkout Command - Git Commands - Edureka

**git merge**

**Usage: git merge [branch name]**

This command merges the specified branch’s history into the current branch.

Git Merge Command - Git Commands - Edureka

**git remote**

**Usage: git remote add [variable name] [Remote Server Link]**

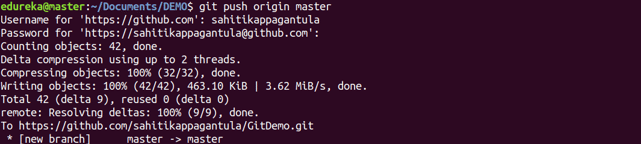
This command is used to connect your local repository to the remote server.

Git Remote Command - Git Commands - Edureka

**git push**

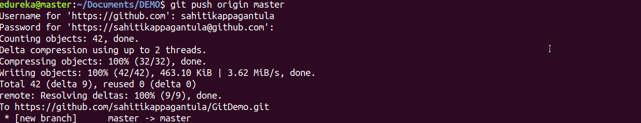
**Usage: git push [variable name] master**

This command sends the committed changes of master branch to your remote repository.



**Usage: git push [variable name] [branch]**

This command sends the branch commits to your remote repository.



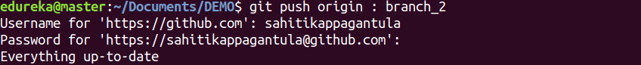
**Usage: git push –all [variable name]**

This command pushes all branches to your remote repository.



**Usage: git push [variable name] :[branch name]**

This command deletes a branch on your remote repository.



**git pull**

**Usage:  git pull [Repository Link]**

This command fetches and merges changes on the remote server to your working directory.



**git stash**

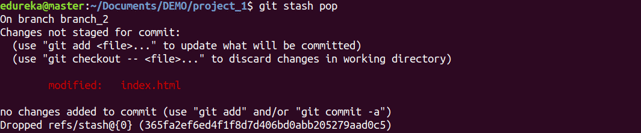
**Usage: git stash save**

This command temporarily stores all the modified tracked files.

Git Stash Command - Git Commands - Edureka

**Usage: git stash pop**

This command restores the most recently stashed files.



**Usage: git stash list**

This command lists all stashed changesets.

Git Stash Command - Git Commands - Edureka

**Usage: git stash drop**

This command discards the most recently stashed changeset.

Git Stash Command - Git Commands - Edureka

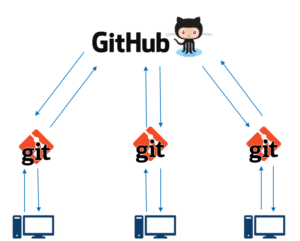
# How To Use GitHub – Developers Collaboration Using GitHub

## ****Step 1: Introduction to GitHub****

To be very crisp about it, GitHub is a file or code sharing service to collaborate with different people.

GitHub is a highly used software which is typically used for version control. It is helpful when more than just one person is working on a project. Say for example, a software developer team wants to build a website and everyone has to update their codes simultaneously while working on the project. In this case, Github helps them to build a centralized repository where everyone can upload, edit and manage the code files.

GitHub has various advantages but many people often have a doubt as to why not use dropbox or any cloud based system? Let me take the same example forward to answer this question. Say more than two software developers are working on the same file and they want to update it simultaneously. Unfortunately, the person who save the file first will get precedence over the others. While in Github, this is not the case. Github document the changes and reflect them in an organized manner to avoid any chaos between any of the files uploaded.  
Therefore using GitHub centralized repository, it avoids all the confusion and working on the same code becomes very easy.

If you look at the image on the left, GitHub is a central repository and Git is a tool which allows you to create a local repository. Now people usually get confused between git and GitHub but its actually very different. Git is a version control tool that will allow you to perform all kinds of operations to fetch data from the central server or push data to it whereas GitHub is a core hosting platform for version control collaboration. GitHub is a company that allows you to host a central repository in a remote server.

Now let me list down the ways in which GitHub makes git simple:

* GitHub provides you a beautiful visual interface which helps you to track or manage your version controlled projects locally.
* Once you register on GitHub, you can connect with social network and build a strong profile.

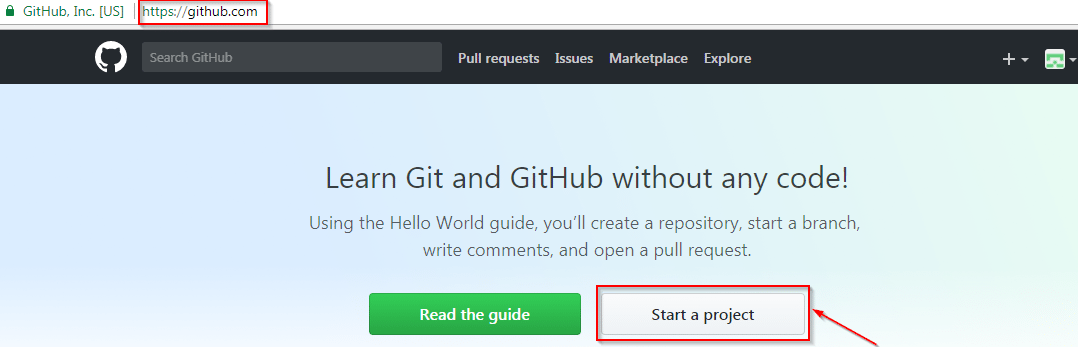
So let’s get started with GitHub.

## ****Step 2: Creating a GitHub Repository****

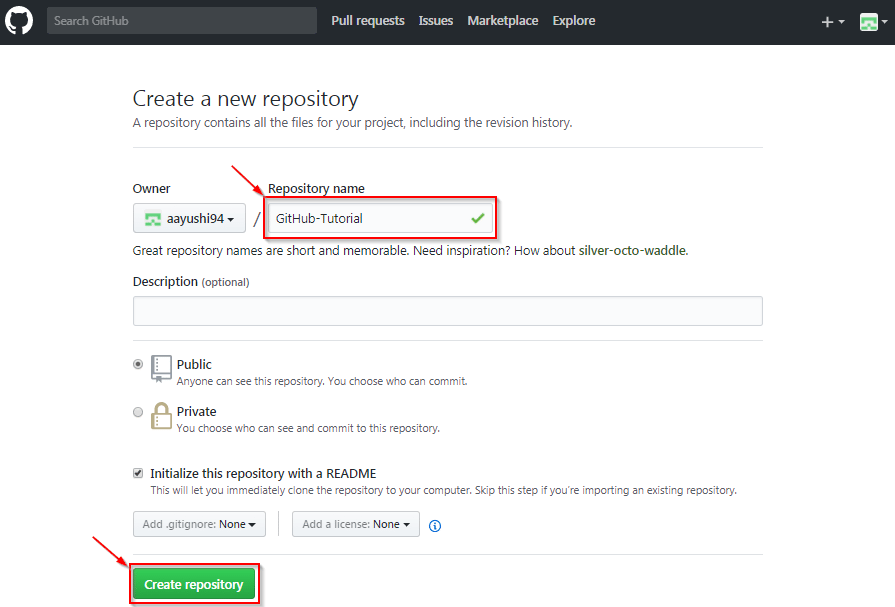
A repository is a storage space where your project lives. It can be local to a folder on your computer, or it can be a storage space on GitHub  or another online host. You can keep code files, text files, images or any kind of a file in a repository. You need a GitHub repository when you have done some changes and are ready to be uploaded. This GitHub repository acts as your remote repository. So let me make your task easy, just follow these simple steps to create a GitHub repository:

* Go to the link: <https://github.com/> . Fill the sign up form and click on “Sign up for Github”.
* Click on “Start a new project”.

Refer to the below screenshot to get a better understanding.

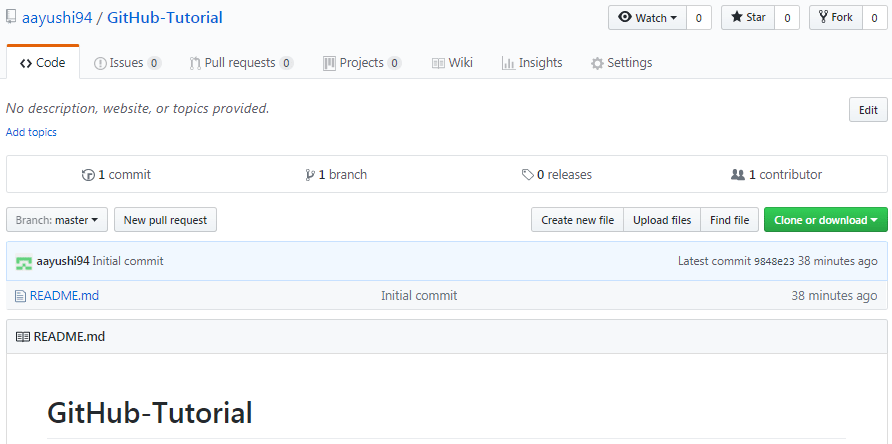


* Enter any repository name and click on “Create Repository”. You can also give a description to your repository (optional).



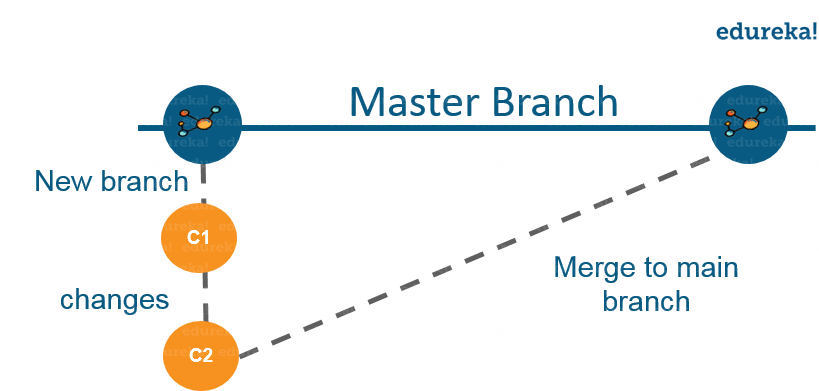
Now, if you noticed by default a GitHub repository is public which means that anyone can view the contents of this repository whereas in a private repository, you can choose who can view the content. Also, private repository is a paid version. Also, if you refer the above screenshot, initialize the repository with a README file. This file contains the description of the file and once you check this box, this will be the first file inside your repository.

Congratulations, your repository is successfully created! It will look like the below screenshot:



So now my central repository has been sucessfully created! Once this is done, you are ready to commit, pull, push and perform all the other operations. Now let’s move forward and understand branching in GitHub.

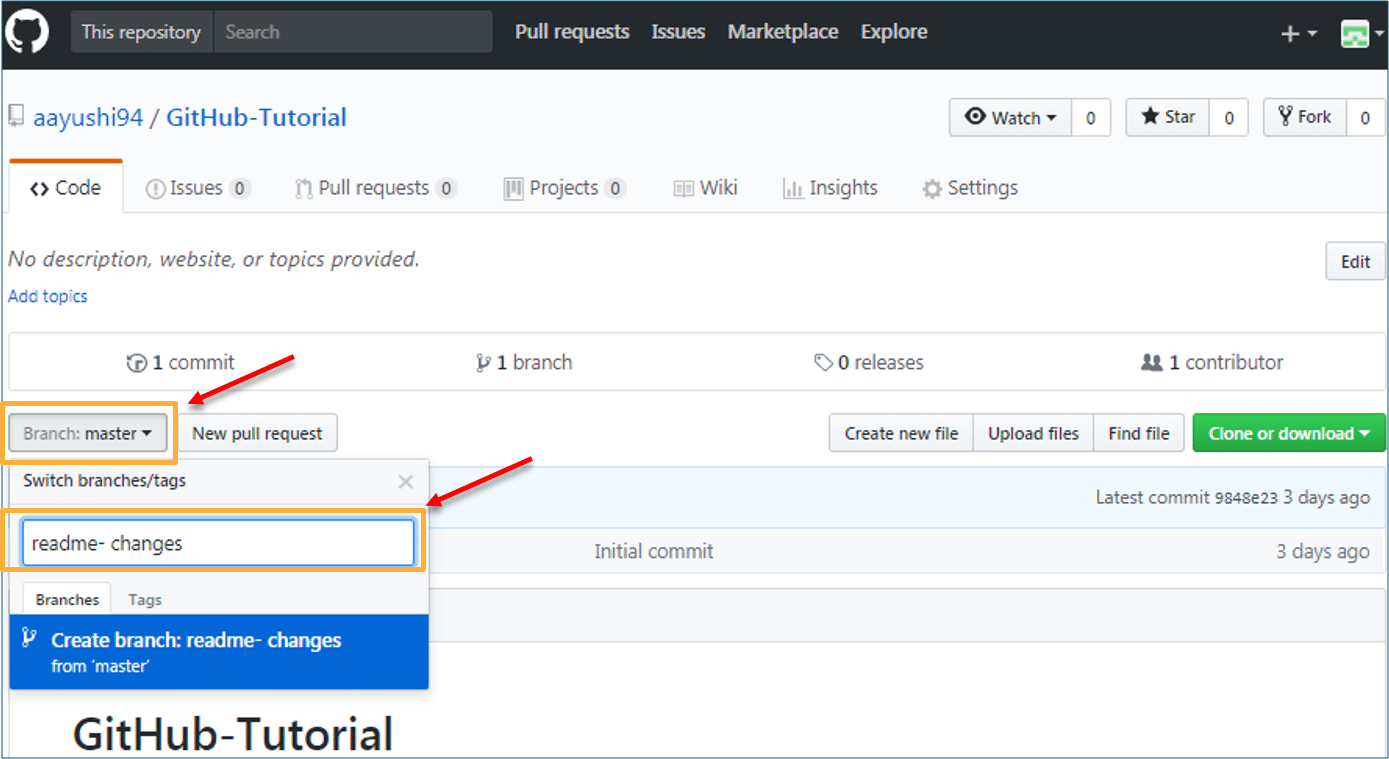
**Step 3: Create Branches and Perform Operations**

**Branching:**Branches help you to work on different versions of a repository at one time. Let’s say you want to add a new feature (which is in the development phase), and you are afraid at the same time whether to make changes to your main project or not. This is where git branching comes to rescue. Branches allow you to move back and forth between the different states/versions of a project. In the above scenario, you can create a new branch and test the new feature without affecting the main branch. Once you are done with it, you can merge the changes from new branch to the main branch. Here the main branch is the master branch, which is there in your repository by default. Refer to the below image for better understanding:

As depicted in the above image, there is a master/ production branch which has a new branch for testing. Under this branch, two set of changes are done and once it completed, it is merged back to the master branch. So this is how branching works!  
Let’s move ahead in ‘how to use GitHub’ blog, and learn how you can create a branch.

To create a branch in GitHub, follow the below steps:

* Click on the dropdown “Branch: master”
* As soon as you click on the branch, you can find an existing branch or you can create a new one. In my case, I am creating a new branch with a name “readme- changes”. Refer to the below screenshot for better understanding.



Once you have created a new branch, you have two branches in your repository now i.e. read-me (master branch) and readme- changes. The new branch is just the copy of master branch. So let’s perform some changes in our new branch and make it look different from the master branch.

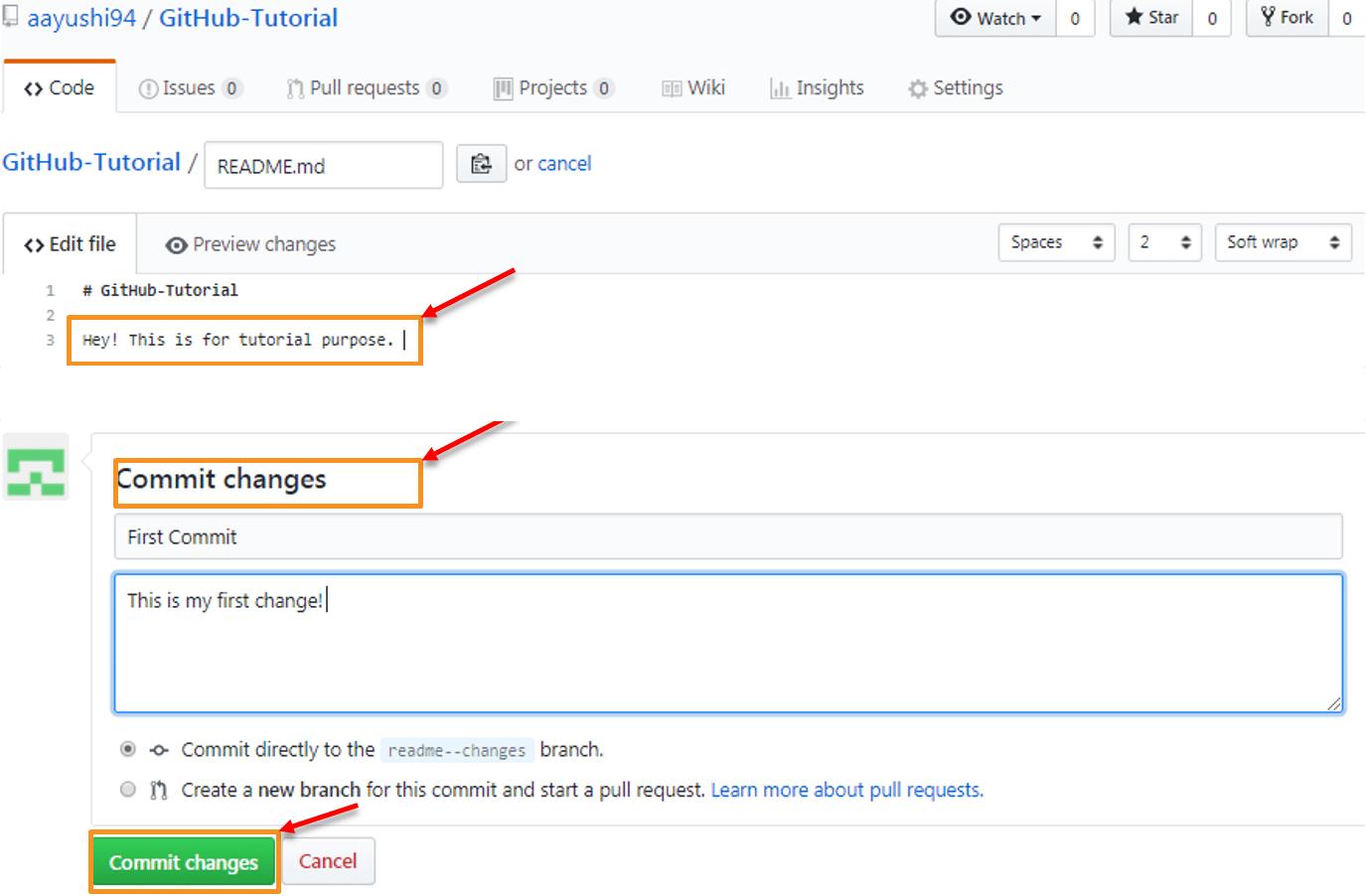
**How to use GitHub: Operations**

**Commit Command:**

This operation helps you to save the changes in your file. When you commit a file, you should always provide the message, just to keep in the mind the changes done by you. Though this message is not compulsory but it is always recommended so that it can differentiate the various versions or commits you have done so far to your repository. These commit messages maintain the history of changes which in turn help other contributors to understand the file better. Now let’s make our first commit, follow the below steps:

* Click on “readme- changes” file which we have just created.
* Click on the “edit” or a pencil icon in the righmost corner of the file.
* Once you click on that, an editor will open where you can type in the changes or anything.
* Write a commit message which identifies your changes.
* Click commit changes in the end.

Refer to the below screenshot for better understanding:

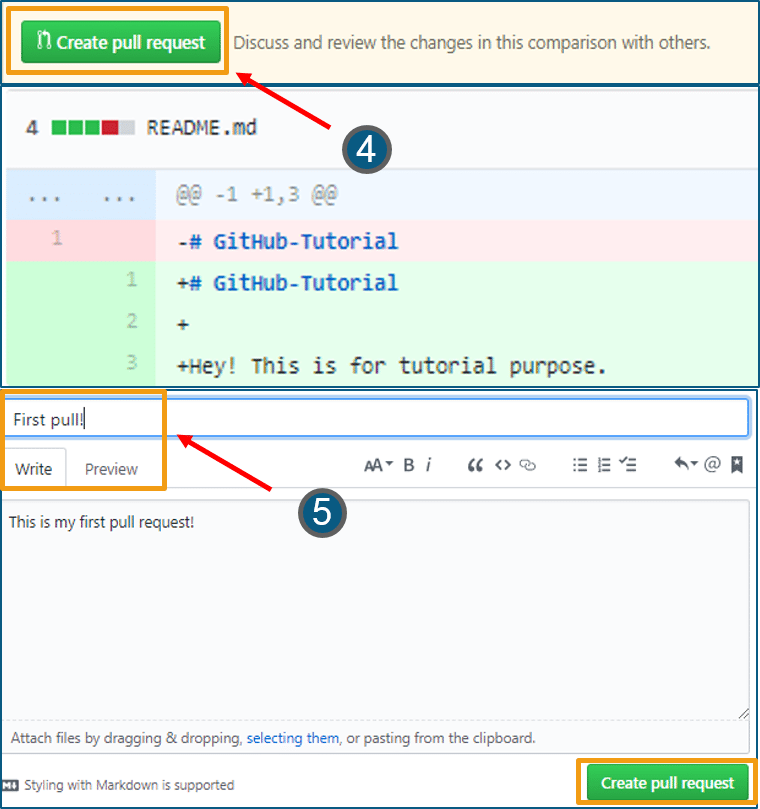
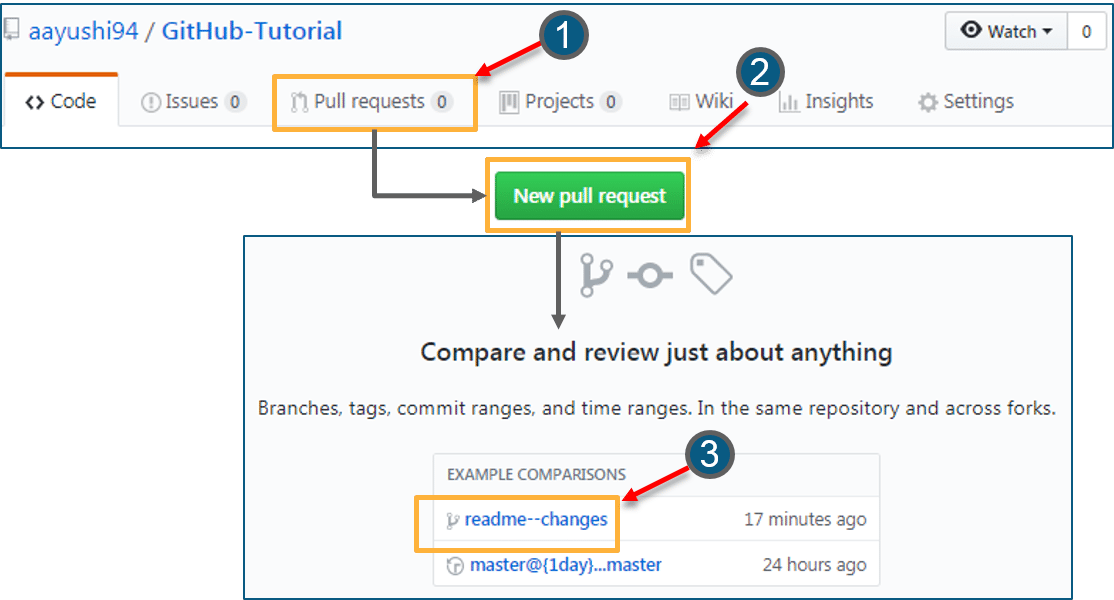


We have successfully made our first commit. Now this “readme- changes” file is different from the master branch. Next, let us see how can we open a pull request.

**Pull Command**

Pull command is the most important command in GitHub. It tell the changes done in the file and request other contributors to view it as well as merge it with the master branch. Once the commit is done, anyone can pull the file and can start a discussion over it. Once its all done, you can merge the file. Pull command compares the changes which are done in the file and if there are any conflicts, you can manually resolve it. Now let us see different steps involved to pull request in GitHub.

* Click the ‘Pull requests’ tab.
* Click ‘New pull request’.
* Once you click on pull request, select the branch and click ‘readme- changes’ file to view changes between the two files present in our repository.
* Click “Create pull request”.
* Enter any title, description to your changes and click on “Create pull request”. Refer to the below screenshots.

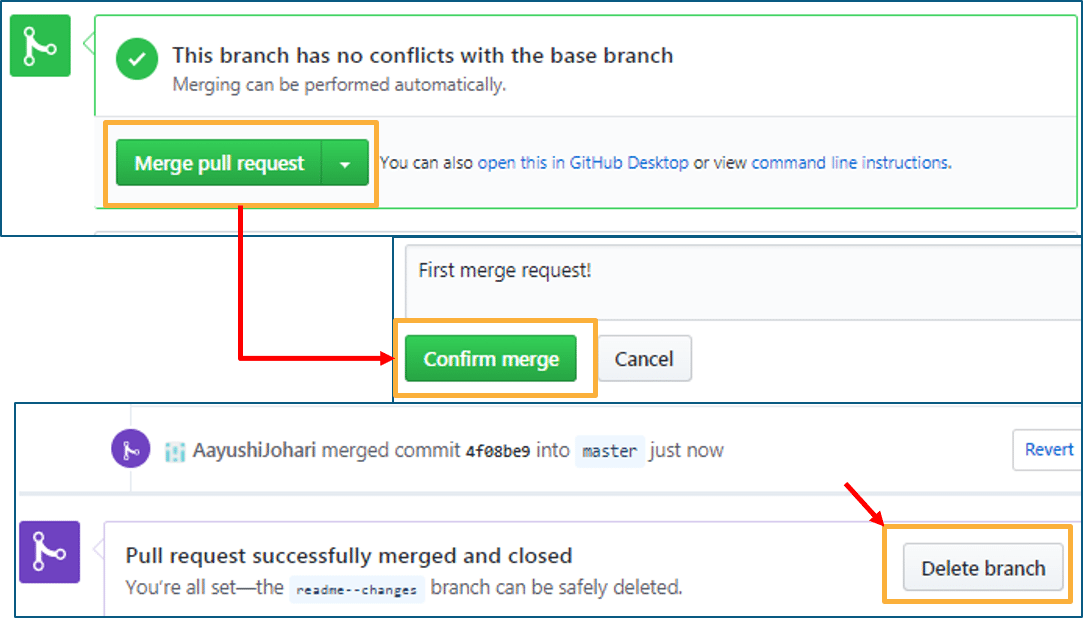


Next, let us move forward and see how can you merge your pull request.

**Merge Command**

Here comes the last command which merge the changes into the main master branch. We saw the changes in pink and green color, now let’s merge the “readme- changes” file with the master branch/ read-me. Go through the below steps to merge pull request.

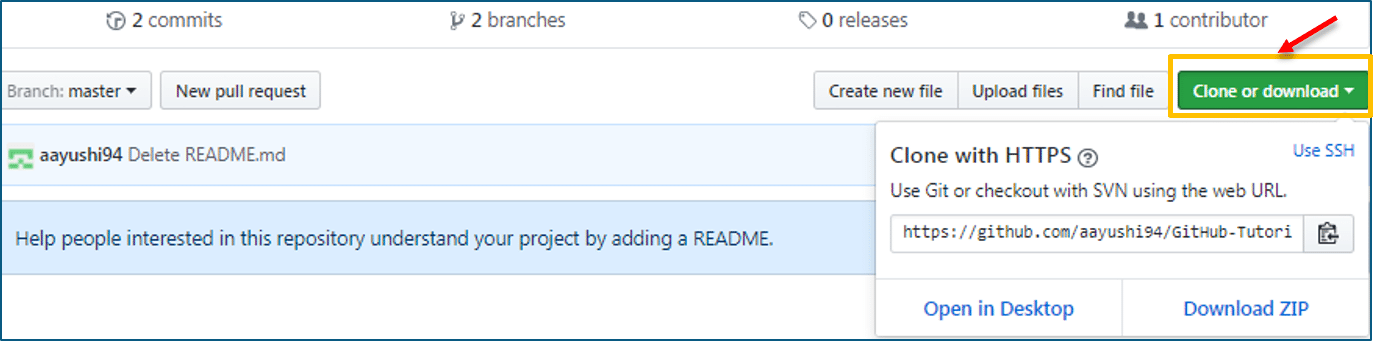
* Click on “Merge pull request” to merge the changes into master branch.
* Click “Confirm merge”.
* You can delete the branch once all the changes have been incorporated and if there are no conflicts. Refer to the below screenshots.



I hope you guys are trying these steps simultaneously while you are learning how to use GitHub. Next, let us move to our last topic in ‘how to use GitHub’ blog, i.e. Cloning and forking a GitHub repository.

**Step 4: Cloning and Forking GitHub Repository**

**Cloning:** Before I actually talk about cloning a GitHub repository, first let us understand why do we need to clone a repository. The answer is simple! Suppose you want to use some code which is present in a public repository, you can directly copy the contents by cloning or downloading. Refer to the below screenshot for a better understanding.



Cloning is really simple! In case you are facing any challenges on how to use GitHub, please comment your problems in the section below. Moving forward, let’s see what forking is.

**Forking:** First, let us talk about why do we need forking. Suppose, you need some code which is present in a public repository, under your repository and GitHub account. For this, we need to fork a repository.

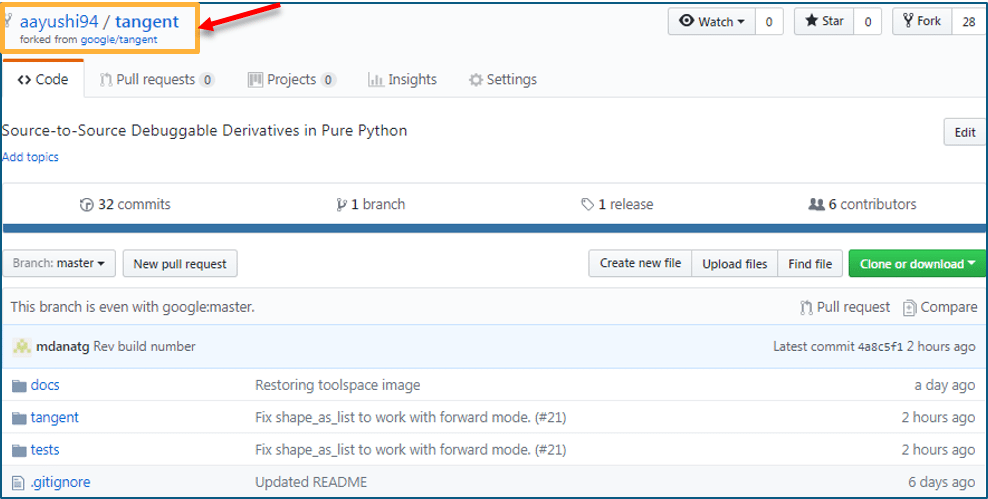
Before we get started with forking, there are some important points which you should always keep in mind.

* Changes done to the original repository will be reflected back to the forked repository.
* If you make a change in forked repository, it will not b reflected to the original repository until and unless you have made a pull request.

Now let’s see how can you want to fork a repository. For that, follow the below steps:

* Go to Explore and search for public repositories.
* Click “fork”. Note that this “tangent” repository is already forked 27 times and it is under “google”account. Refer the below image for better understanding.  
    
  

As soon as you click on “Fork”, it will take some time to fork the repository. Once done you will notice that the repository name is under your account. For reference, you can have a look at the below screenshot.



Congratulations! You have successfully forked an existing repository under your own account.

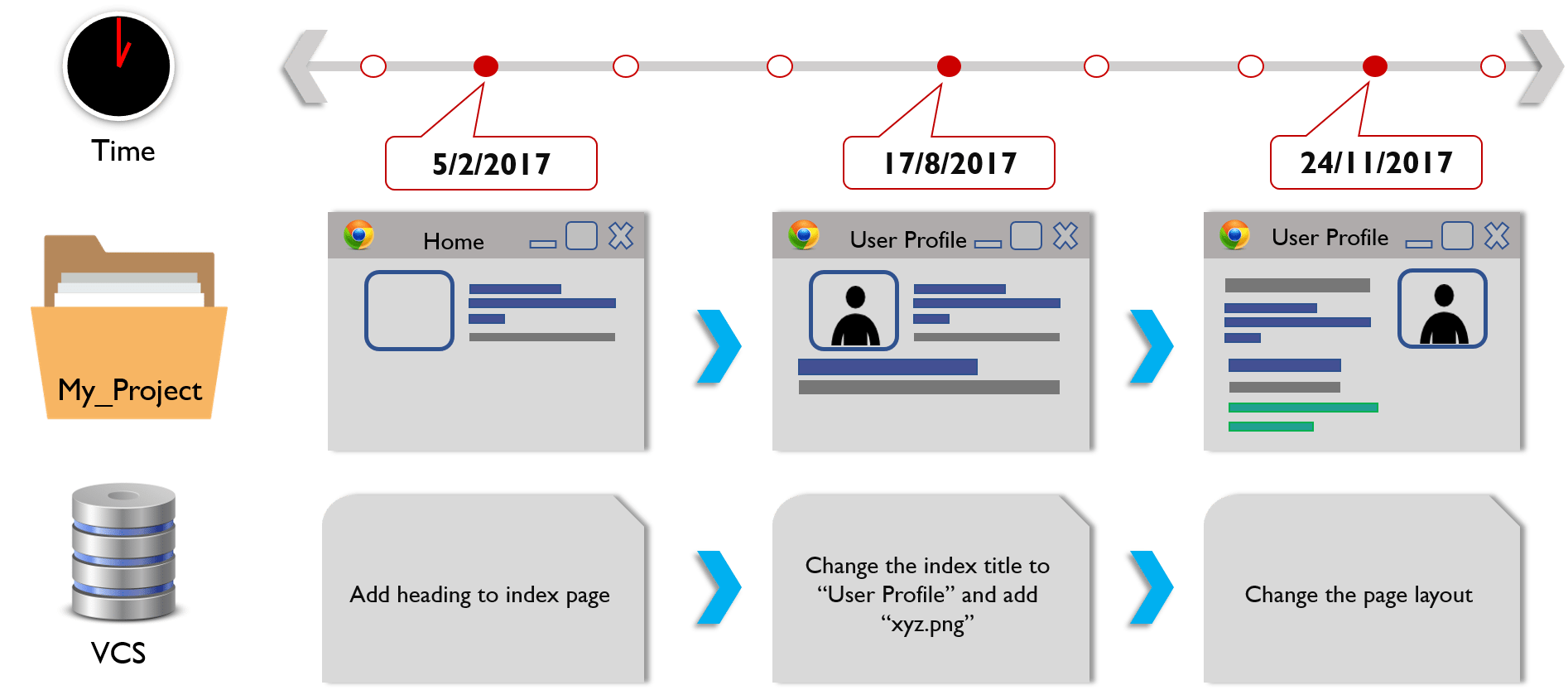
# Git vs Github – Demystifying The Differences

## ****Version Control – Git vs GitHub****

To understand the exact difference between git and GitHub you first need to know about version control. So, what do you understand by the term version control?

The term Version control refers to a system that records changes to a file or set of files over time called the ‘versions’. In other words, these versions will help you in tracking the changes in your codes/project and if necessary, undo those changes as well. This feature of being able to compare, differentiate and revert changes between two versions of a particular project becomes really helpful when working on a larger project. Larger projects mean more people working on the same code which increases the chances of conflicts. Using version control you can easily prevent these conflicts.

This system which controls the versioning of the codes is called a Version Control System (VCS). For a better picture, you can think it as a ‘database’.  Within this database, VCS takes snapshots of your entire project and stores them as versions. Now with the help of these snapshots, you can easily compare the versions and switch between them as per the need. Check out the below picture which shows the entire process:



I think now you clearly understand the concept of version control. One thing I would like to mention here is, don’t get confused between a VCS and a backup/ deployment system. This VCS totally eliminates the need for changing or replacing the toolchain completely unlike the backup systems. Below I have listed few of the advantages of using version control:

* Version control creates proper workflows which help in preventing the chaos among people using their own development process with different and incompatible tools.
* Each version has a description of what the changes in this version are done. These descriptions help in tracking the changes in the code by version.
* VCS synchronizes the versions and ensures that your changes don’t conflict with others using the same repository.
* VCS helps in keeping the history of changes as other people save new versions of your code.  This history can be reviewed to find out by whom, why, and when changes were done.
* The VCS automation features save yours as well as your teams time and generate consistent results by performing testing, code analysis, and deployment whenever any new versions are saved.

So now let’s proceed ahead with this blog on Git vs GitHub and understand what exactly is Git.

**What Is Git? – Git vs GitHub**

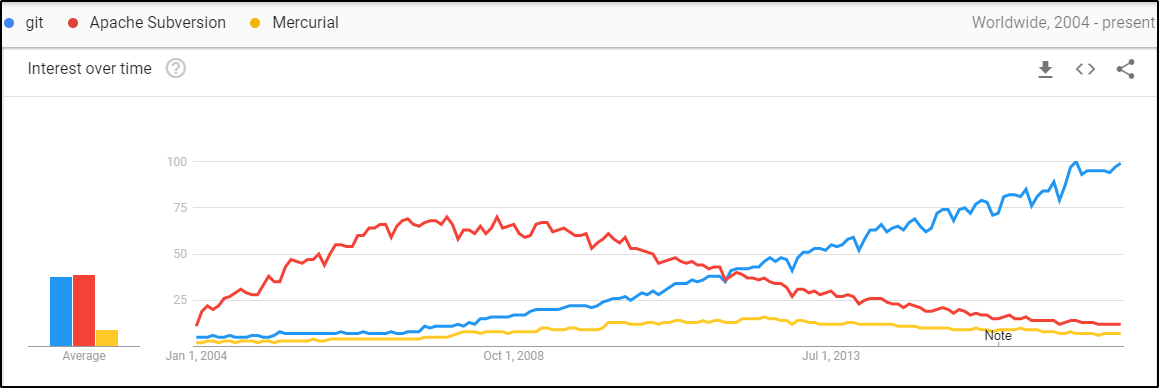
Now that you know about the version control, it will be really easy for you to understand about Git. So, what is Git?

Git is a version control software which you need to install on your local system in order to use it. For an individual working on a project alone, Git proves to be an excellent software. But what if you are working on a project with a large team. All will be working on the same project but each of you will be having a different version of the same project. Suppose you have made changes in the project directory on your machine and you want to send those changes to your collaborators. Also, you want the changes they make to appear directly in your machine’s project directory. As working on a project, being on the same page with your teammates is very crucial to avoid any type of conflict. So, how you will do that? Well, no worries Git takes care of all but the only condition here is, each of your team members must have Git installed on their systems.

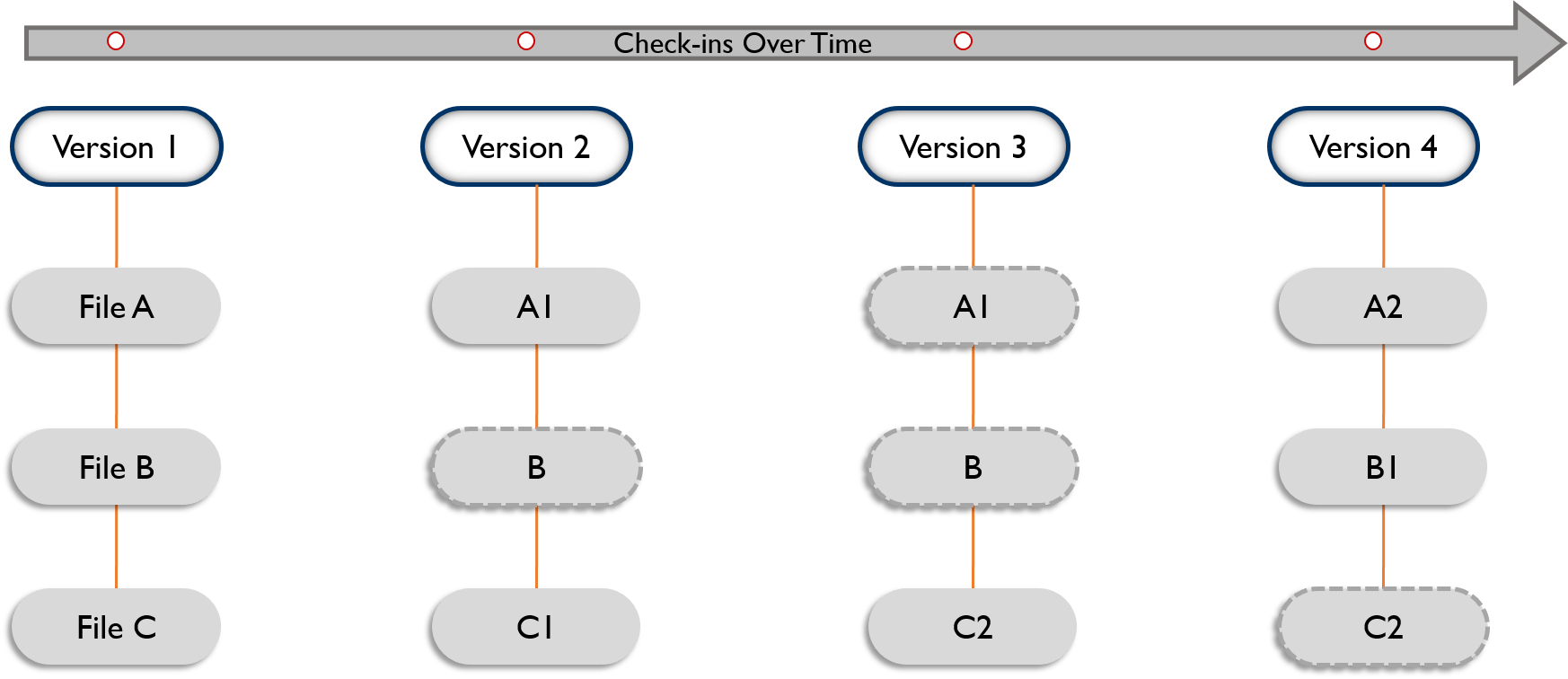
Git also is known as distributed version control system which means using Git you can push and pull yours as well as others changes to other people’s machines:



Now, using Git you can work on the same copy as your team member. But the copy on which you will be working, won’t reflect any of the changes in the main directory unless and until you pull your collaborator’s changes and push back your own changes.



In today’s market, Git is the most widely and popularly used modern VCS. It is quite matured and is a well maintained open source project. It was developed back in 2005, by Linus Torvalds who is also the creator of Linux OS kernel. If you check the above picture of the Google trends result, you will find that Git has continued to grow in the market since 2005. A varying number of software projects, including commercial as well as open source, completely rely on Git for their version control. So, how Git is different from the rest that makes it so popular?

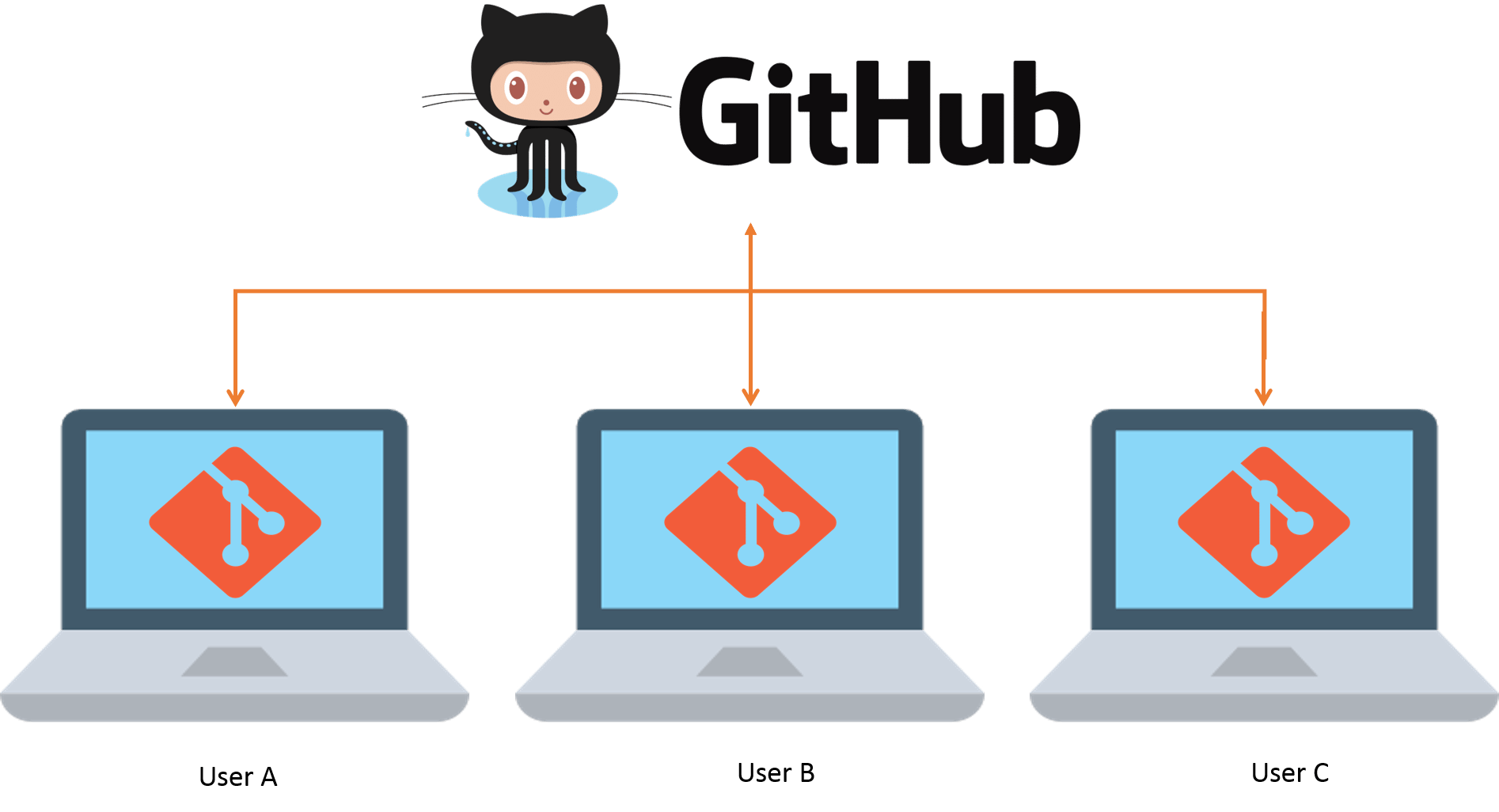
The major difference between Git and any other VCS like Subversion (SVN), Mercurial, TFS, Perforce, Bazaar etc., is the way Git stores its data. In other systems, information is stored as a list of file-based changes which is also known as delta-based version control. But in case of Git, it stores its data in the form of a stream of snapshots of a miniature filesystem. With Git, whenever you commit or save the state of your project, a snapshot is taken of the present file state and a reference is stored for it. If no changes or updates are done then Git stores a link to the previous file that it has already stored. Following diagram shows how Git stores the versions:

But Git will only be helpful when you know when is your collaborator’s system is turned on and is connected to a network. So what will you do when your team members are not online? In these situations, a third party having an identical copy of your project, from where you can easily push and pull the changes, will come in handy.

Well, you will be glad to know that, this is what GitHub does and in the next section of this blog on Git vs GitHub, I will explain all about it.

**What Is GitHub? – Git vs GitHub**

As told, Git is a version control system which tracks the changes when working with computer codes while GitHub is a Web-based Git version control repository hosting service. It provides all of the distributed version control and source code management (SCM) functionalities of Git while topping it with a few of its own features. It is a heaven for the developers where they can store their projects and get connected with like-minded people. You can think it as “cloud for codes”.



So basically it is a place to store your identical working directories or repositories. It literally is a hub for the Git repositories which you can use, simply by creating a free account on the GitHub. These accounts come with an abundant storage space where you can store your repositories and build a proper profile which holds a great value. By default the repositories are public i.e., everyone can see your codes but you can make it private as well. So if you are a good coder you can upload your own applications and programs and get recognition from others.

It works along with the collaboration of Git. While Git is a command line tool, GitHub is a Web-based graphical interface which provides you with the access control, basic task management tools along with several collaboration features. Moreover, GitHub can host your project’s source codes in various programming languages and keep the track of the each of the changes made to every iteration. GitHub’s functionality doesn’t end here. It provides following 3 extraordinary features which makes it so powerful:

1. ***fork:*** Or commonly known as forking is copying a repository from one user’s account when you don’t have the write access to it. So you can just copy it and modify it under your own account.
2. ***pull:***When you have made the changes in codes that you have copied and want to share them with its original order. Then you can send a notification called a “pull request” to them.
3. ***merge:*** Now the user who is the owner of those codes, if, finds your changes relevant can merge the changes found in your repo with the original repo, by just button click.

If your pull request is accepted by the owner then you get its credit on the original site and shows up in your user profile. It is a kind of a resume that helps the GitHub project maintainer to determine your reputation. So, more the people and projects on GitHub, the better idea a project maintainer gets of its potential contributors. This encourages the young developers and projects to grow more in the industry. I hope now you understand what is Github. So let me now summarize the basic differences between Git and Github.

## ****Git vs GitHub****

|  |  |
| --- | --- |
| git logo - Git vs GitHub - Edureka | GitHub logo - Git vs GitHub - Edureka |
| 1. It is a software | 1. It is a service |
| 2. It is installed locally on the system | 2. It is hosted on  Web |
| 3. It is a command line tool | 3. It provides a graphical interface |
| 4. It is a tool to manage different versions of edits, made to files in a git repository | 4. It is a space to upload a copy of the **Git** repository |
| 5. It provides functionalities like Version Control System Source Code Management | 5. It provides functionalities of Git like VCS, Source Code Management as well as adding few of its own features |

This concludes this blog on Git vs GitHub. Hope you enjoyed reading this blog and learned something new.

# [Reverting a git commit after pushing to remote](https://willwarren.com/2013/04/20/reverting-a-git-commit-after-pushing-to-remote/)

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

Imagine a scenario where you have a git repo with 2 branches; master, the production-ready branch and dev, the branch where all the development occurs.

Now imagine that you accidentally made a commit on master, when really it should have been on dev. If you have not yet pushed to a remote repository (like Github), you can undo that commit using git reset like so:

git reset --soft HEAD~1

This will bring your repository back to the state it was in right before you did your git commit. Now you can switch to your dev branch and re-commit the changes in the right place.

The --soft option tells git to leave your index (or “staging area”) and your working tree alone. If you were to run this same command with --hard it would trash all your local changes. This is fine if you want to throw all your work away, but if the work is good, just the commit was bad, then use --soft.

The HEAD~1 just means “the latest commit’s parent”. It could also be written as HEAD~ or HEAD^.

This is all well and good, but what if you had git pushed right after doing the erroneous commit? If you just try the steps outlined above, and then try and push to your remote repo, you will get an error because the tip of your local repo is behind that of the remote and it will reject your push.

This is where you need to use git revert.

git revert HEAD

This command essentially says, “I want to create a new commit that undoes the commit pointed to by HEAD”. Once the command has been executed, it creates a new commit which you can push back to your remote repo which will effectively create a patch which undoes all the changes in the last commit.

# Difference between ‘git pull’, ‘git fetch’ & ‘git clone’

Git was designed to support a more distributed model with no need for a central repository.Git was designed so that people on an unreliable link could exchange code via email, even. In order to support this model git maintains a local repository with your code and also an additional local repository that mirrors the state of the remote repository. By keeping a copy of the remote repository locally, git can figure out the changes needed even when the remote repository is not reachable. Later when you need to send the changes to someone else, git can transfer them as a set of changes from a point in time known to the remote repository.

**Git fetch**

$ git fetch origin

**git fetch** only downloads new data from a remote repository – but it doesn’t integrate any of this new data into working files/directory. Fetch is good for getting a fresh view on all the things that happened in a remote repository. It is harmless and doesnt manipulate local changes.

**Git Pull**

$ git pull origin master

git pull does a git fetch followed by a git merge. Since “git pull” tries to merge remote changes with your local ones, a so-called “merge conflict” can occur. It’s highly recommended to start a “git pull” only with a clean working copy.

**Git clone**

$git clone git@github.com:whatever/something.git

Git clone will clone a repo int a newly created directory. Git clone additionally creates a remote called ‘origin’ for the repo cloned from, sets up a local branch based on the remote’s active branch (generally master), and creates remote-tracking branches for all the branches in the repo

**Git Pull**

From what I understand, git pull will pull down from a remote whatever you ask (so, whatever trunk you’re asking for) and instantly merge it into the branch you’re in when you make the request. Pull is a high-level request that runs ‘fetch’ then a ‘merge’ by default, or a rebase with ‘–rebase’. You could do without it, it’s just a convenience.

|  |  |
| --- | --- |
| 1  2  3  4  5 | %> git checkout localBranch  %> git pull origin master  %> git branch  master  \* localBranch |

The above will merge the remote “master” branch into the local “localBranch”.

**Git fetch**

Fetch is similar to pull, except it won’t do any merging.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | %> git checkout localBranch   %> git fetch origin remoteBranch  %> git branch  master  \* localBranch  remoteBranch |

So, the fetch will have pulled down the remoteBranch and . creates a local copy of a remote branch which you shouldn’t manipulate directly; instead create a proper local branch and work on that. ‘git checkout’ has a confusing feature though. If you ‘checkout’ a local copy of a remote branch, it creates a local copy and sets up a merge to it by default.

**Git clone**

Git clone will clone a repo int a newly created directory. It’s useful for when you’re setting up your local doodah

|  |  |
| --- | --- |
| 1  2  3  4  5 | %> cd newfolder  %> git clone git@github.com:whatever/something.git  %> git branch   \* master  remoteBranch |

Git clone additionally creates a remote called ‘origin’ for the repo cloned from, sets up a local branch based on the remote’s active branch (generally master), and creates remote-tracking branches for all the branches in the repo

# [How to delete a Git branch both locally and remotely?](http://www.ranjeetvimal.com/delete-git-branch-locally-remotely-2/)

**Delete Remote Branch**

$ git push origin --delete <branch\_name>

$ git branch -d <branch\_name>

**Delete Local Branch**

To delete the *local* branch use:

$ git branch -d branch\_name

**Note:** The -d option is an alias for --delete, which only deletes the branch if it has already been fully merged in its upstream branch. You could also use -D, which is an alias for --delete --force, which deletes the branch “irrespective of its merged status.

# [Your local changes to the following files would be overwritten by merge](http://www.ranjeetvimal.com/local-changes-following-files-overwritten-merge/)

This is very common problem we face while taking pull since pull tries to merge with local change.

How to ignore error on git pull about my local changes would be overwritten by merge?

If you want remove all local changes from your working copy, simply stash them:

git stash save --keep-index

Or

override all local changes

git reset –hard git pull

# [How to undo last commit in Git?](http://www.ranjeetvimal.com/how-to-undo-last-commit-git-github/)

One of the most useful features of any git or any version control is the ability to “undo” your mistakes. When you make a new commit, Git stores a snapshot of your repository at that specific moment in time; later, you can use Git to go back to an earlier version of your project.

git reset --soft HEAD~1

# Set HEAD to point to an earlier commit, make the changes

git status #you can see changed file here.

git add . #add all files

git commit -c ORIG\_HEAD

# Commit the changes, reusing the old commit message. reset copied the old head to .git/ORIG\_HEAD; commit with -c ORIG\_HEAD will open an editor, which initially contains the log message from the old commit and allows you to edit it. If you do not need to edit the message, you could use the -C option

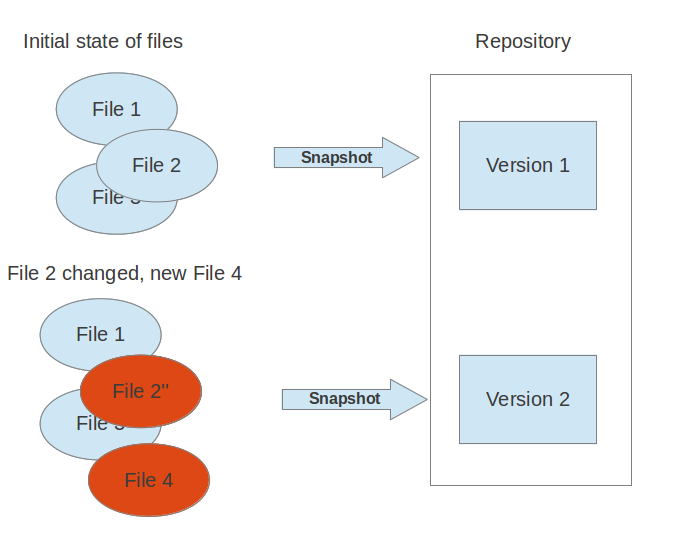
*---------------------------------------------------------------------------------------------------------------------------*

*This tutorial explains the usage of the distributed version control system Git via the command line. The examples were done on Linux (Ubuntu), but should also work on other operating systems like Microsoft Windows.*

[**1. What is a version control system?**](http://www.vogella.com/tutorials/Git/article.html#versioncontrolssystems)

A version control system (VCS) allows you to track the history of a collection of files. It supports creating different versions of this collection. Each version captures a snapshot of the files at a certain point in time and the VCS allows you to switch between these versions. These versions are stored in a specific place, typically called a *repository*.

You may, for example, revert the collection of files to a state from 2 days ago. Or you may switch between versions of your files for experimental features. The process of creating different versions (snapshots) in the repository is depicted in the following graphic. Please note that this picture fits primarily to Git. Other version control systems like *Concurrent Versions System* (CVS) don’t create snapshots of the files but store file deltas.



VCS are typically used to track changes in text files. These text files can for example be source code for a programming language, HTML or configuration files. Of course, version control systems are not limited to text files, they can also handle other types of files. For example, you may use a VCS to track the different versions of a png file.

## [2. Localized and centralized version control systems](http://www.vogella.com/tutorials/Git/article.html#cvcs_definition)

A localized version control system keeps local copies of the files. This approach can be as simple as creating a manual copy of the relevant files.

A centralized version control system provides a server software component which stores and manages the different versions of the files. A developer can copy (checkout) a certain version from the central sever onto their individual computer.

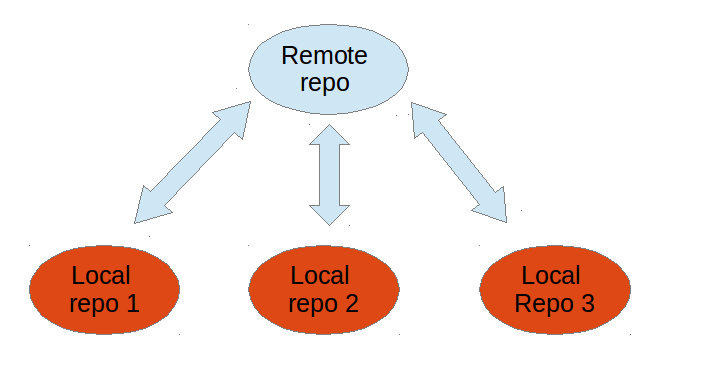
Both approaches have the drawback that they have one single point of failure. In a localized version control systems it is the individual computer and in a centralized version control systems it is the server machine. Both system makes it also harder to work in parallel on different features.

## [3. Distributed version control systems](http://www.vogella.com/tutorials/Git/article.html#dvcs_definition)

In a distributed version control system each user has a complete local copy of a repository on his individual computer. The user can copy an existing repository. This copying process is typically called cloning and the resulting repository can be referred to as a clone.

Every clone contains the full history of the collection of files and a cloned repository has the same functionality as the original repository.

Every repository can exchange versions of the files with other repositories by transporting these changes. This is typically done via a repository running on a server which is, unlike the local machine of a developer, always online. Typically, there is a central server for keeping a repository but each cloned repository is a full copy of this repository. The decision which of the copies is considered to be the central server repository is pure convention.



## [4. What is Git?](http://www.vogella.com/tutorials/Git/article.html#gitterminlogy)

Git is currently the most popular implementation of a distributed version control system.

Git originates from the Linux kernel development and was founded in 2005 by Linus Torvalds. Nowadays it is used by many popular open source projects, e.g., the Android or the Eclipse developer teams, as well as many commercial organizations.

The core of Git was originally written in the programming language \_C, but Git has also been re-implemented in other languages, e.g., Java, Ruby and Python.

## [5. Git repositories](http://www.vogella.com/tutorials/Git/article.html#gitdefintion_localrepositories)

A Git repository contains the history of a collection of files starting from a certain directory. The process of copying an existing Git repository via the Git tooling is called \_cloning. After cloning a repository the user has the complete repository with its history on his local machine. Of course, Git also supports the creation of new repositories.

If you want to delete a Git repository, you can simply delete the folder which contains the repository.

If you clone a Git repository, by default, Git assumes that you want to work in this repository as a user. Git also supports the creation of repositories targeting the usage on a server.

* bare repositories are supposed to be used on a server for sharing changes coming from different developers. Such repositories do not allow the user to modify locally files and to create new versions for the repository based on these modifications.
* non-bare repositories target the user. They allow you to create new changes through modification of files and to create new versions in the repository. This is the default type which is created if you do not specify any parameter during the clone operation.

A local non-bare Git repository is typically called local repository.

## [6. Working tree](http://www.vogella.com/tutorials/Git/article.html#workingtree)

A local repository provides at least one collection of files which originate from a certain version of the repository. This collection of files is called the working tree. It corresponds to a checkout of one version of the repository with potential changes done by the user.

The user can change the files in the working tree by modifying existing files and by creating and removing files. A file in the working tree of a Git repository can have different states. These states are the following:

* untracked: the file is not tracked by the Git repository. This means that the file never staged nor committed.
* tracked: committed and not staged
* staged: staged to be included in the next commit
* dirty / modified: the file has changed but the change is not staged

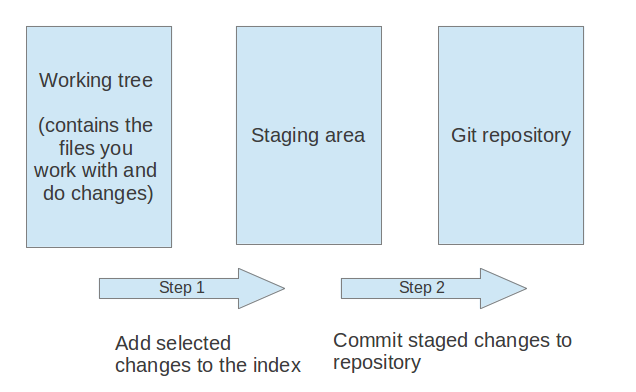
After doing changes in the working tree, the user can add these changes to the Git repository or revert these changes.

## [7. Adding to a Git repository via staging and committing](http://www.vogella.com/tutorials/Git/article.html#gitaddingprocess)

After modifying your working tree you need to perform the following two steps to persist these changes in your local repository:

* add the selected changes to the staging area (also known as index) via the git addcommand
* commit the staged changes into the Git repository via the git commit command

This process is depicted in the following graphic.



The git add command stores a snapshot of the specified files in the staging area. It allows you to incrementally modify files, stage them, modify and stage them again until you are satisfied with your changes.

Some tools and Git user prefer the usage of the index instead of staging area. Both terms mean the same thing.

After adding the selected files to the staging area, you can commit these files to add them permanently to the Git repository. \_ Committing\_ creates a new persistent snapshot (called commit or commit object) of the staging area in the Git repository. A commit object, like all objects in Git, is immutable.

The staging area keeps track of the snapshots of the files until the staged changes are committed.

For committing the staged changes you use the git commit command.

If you commit changes to your Git repository, you create a new commit object in the Git repository. See [Commit object (commit)](http://www.vogella.com/tutorials/Git/article.html#commit_object) for information about the commit object.

## [8. Synchronizing with other Git repositories (remote repositories)](http://www.vogella.com/tutorials/Git/article.html#gitdefintion_remoterepositories)

Git allows the user to synchronize the local repository with other (remote) repositories.

Users with sufficient authorization can send new version in their local repository to to remote repositories via the push operation. They can also integrate changes from other repositories into their local repository via the fetch and pull operation.

## [9. The concept of branches](http://www.vogella.com/tutorials/Git/article.html#gitdefinition_branching)

Git supports branching which means that you can work on different versions of your collection of files. A branch allows the user to switch between these versions so that he can work on different changes independently from each other.

For example, if you want to develop a new feature, you can create a branch and make the changes in this branch. This does not affect the state of your files in other branches. For example, you can work independently on a branch called production for bugfixes and on another branch called feature\_123 for implementing a new feature.

Branches in Git are local to the repository. A branch created in a local repository does not need to have a counterpart in a remote repository. Local branches can be compared with other local branches and with \_remote-tracking branches. A remote-tracking branch proxies the state of a branch in another remote repository.

Git supports the combination of changes from different branches. The developer can use Git commands to combine the changes at a later point in time.

## [10. Summary of the core Git terminology](http://www.vogella.com/tutorials/Git/article.html#gitterminology)

The following table provides a summary of important Git terminology discussed in this section

| *Table 1. Git terminology* | |
| --- | --- |
| **Term** | **Definition** |
| Branch | A *branch* is a named pointer to a commit. Selecting a branch in Git terminology is called *to checkout a branch. If you are working in a certain branch, the creation of a new commit advances this pointer to the newly created commit.*  *Each commit knows their parents (predecessors). Successors are retrieved by traversing the commit graph starting from branches or other refs, symbolic references (for example: HEAD) or explicit commit objects. This way a branch defines its own line of descendants in the overall version graph formed by all commits in the repository.*  *You can create a new branch from an existing one and change the code independently from other branches. One of the branches is the default (typically named \_master* ). The default branch is the one for which a local branch is automatically created when cloning the repository. |
| Commit | When you commit your changes into a repository this creates a new *commit object* in the Git repository. This *commit object* uniquely identifies a new revision of the content of the repository.  This revision can be retrieved later, for example, if you want to see the source code of an older version. Each commit object contains the author and the committer. This makes it possible to identify who did the change. The author and committer might be different people. The author did the change and the committer applied the change to the Git repository. This is common for contributions to open source projects. |
| HEAD | *HEAD* is a symbolic reference most often pointing to the currently checked out branch.  Sometimes the *HEAD* points directly to a commit object, this is called *detached HEAD mode*. In that state creation of a commit will not move any branch.  If you switch branches, the *HEAD* pointer points to the branch pointer which in turn points to a commit. If you checkout a specific commit, the*HEAD* points to this commit directly. |
| Index | *Index* is an alternative term for the *staging area*. |
| Repository | A *repository* contains the history, the different versions over time and all different branches and tags. In Git each copy of the repository is a complete repository. If the repository is not a bare repository, it allows you to checkout revisions into your working tree and to capture changes by creating new commits. Bare repositories are only changed by transporting changes from other repositories.  This description uses the term *repository* to talk about a non-bare repository. If it talks about a bare repository, this is explicitly mentioned. |
| Revision | Represents a version of the source code. Git implements revisions as *commit objects* (or short *commits* ). These are identified by an SHA-1 hash. |
| Staging area | The *staging area* is the place to store changes in the working tree before the commit. The *staging area* contains a snapshot of the changes in the working tree (changed or new files) relevant to create the next commit and stores their mode (file type, executable bit). |
| Tag | A *tag* points to a commit which uniquely identifies a version of the Git repository. With a tag, you can have a named point to which you can always revert to. You can revert to any point in a Git repository, but tags make it easier. The benefit of tags is to mark the repository for a specific reason, e.g., with a release.  Branches and tags are named pointers, the difference is that branches move when a new commit is created while tags always point to the same commit. Tags can have a timestamp and a message associated with them. |
| URL | A URL in Git determines the location of the repository. Git distinguishes between *fetchurl* for getting new data from other repositories and *pushurl* for pushing data to another repository. |
| Working tree | The *working tree* contains the set of working files for the repository. You can modify the content and commit the changes as new commits to the repository. |

## [11. The details of the commit objects](http://www.vogella.com/tutorials/Git/article.html#git)

### [11.1. Commit object (commit)](http://www.vogella.com/tutorials/Git/article.html#commit_object)

Conceptually a commit object (short:commit) represents a version of all files tracked in the repository at the time the commit was created. Commits know their parent(s) and this way capture the version history of the repository.

### [11.2. Technical details of a commit object](http://www.vogella.com/tutorials/Git/article.html#commit_objects)

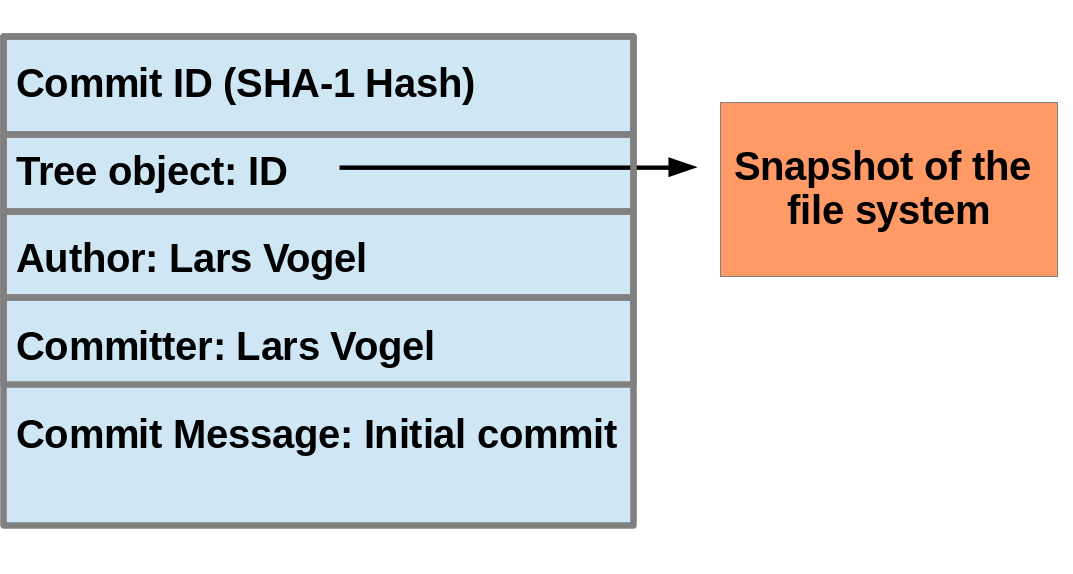
This commit object is addressable via a hash ( SHA-1 checksum ). This hash is calculated based on the content of the files, the content of the directories, the complete history of up to the new commit, the committer, the commit message, and several other factors.

This means that Git is safe, you cannot manipulate a file or the commit message in the Git repository without Git noticing that corresponding hash does not fit anymore to the content.

The commit object points to the individual files in this commit via a tree object. The files are stored in the Git repository as blob objects and might be packed by Git for better performance and more compact storage. Blobs are addressed via their SHA-1 hash.

Packing involves storing changes as deltas, compression and storage of many objects in a single pack file. Pack files are accompanied by one or multiple index files which speedup access to individual objects stored in these packs.

A commit object is depicted in the following picture.



The above picture is simplified. Tree objects point to other tree objects and file blobs. Objects which didn’t change between commits are reused by multiple commits.

### [11.3. Hash and abbreviated commit hash](http://www.vogella.com/tutorials/Git/article.html#commits_sha_checksum)

A Git commit object is identified by its hash (SHA-1 checksum). SHA-1 produces a 160-bit (20-byte) hash value. A SHA-1 hash value is typically rendered as a hexadecimal number, 40 digits long.

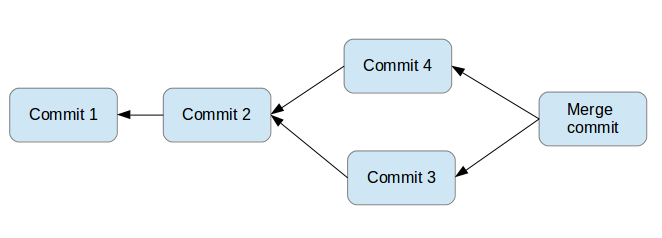
In a typical Git repository you need fewer characters to uniquely identify a commit object. As a minimum you need 4 characters and in a typical Git repository 5 or 6 are sufficient. This short form is called the abbreviated commit hash or abbreviated hash. Sometimes it is also called the shortened SHA-1 or abbreviated SHA-1.

Several commands, e.g., the git log command can be instructed to use the shortened SHA-1 for their output.

## [12. Commit references](http://www.vogella.com/tutorials/Git/article.html#commitreference)

### [12.1. Predecessor commits, parents and commit references](http://www.vogella.com/tutorials/Git/article.html#commitreference_whatis)

Each commit has zero or more direct predecessor commits. The first commit has zero parents, merge commits have two or more parents, most commits have one parent.



In Git you frequently want to refer to certain commits. For example, you want to tell Git to show you all changes which were done in the last three commits. Or you want to see the differences introduced between two different branches.

Git allows addressing commits via commit reference for this purpose.

A commit reference can be a simple reference (simple ref), in this case it points directly to a commit. This is the case for a commit hash or a tag. A commit reference can also be symbolic reference (symbolic ref, symref). In this case it points to another reference (either simple or symbolic). For example HEAD is a symbolic ref for a branch, if it points to a branch. HEAD points to the branch pointer and the branch pointer points to a commit.

### [12.2. Branch references and the HEAD reference](http://www.vogella.com/tutorials/Git/article.html#commitreference_branchheadreferences)

A branch points to a specific commit. You can use the branch name as reference to the corresponding commit. You can also use HEAD to reference the corresponding commit.

### [12.3. Parent and ancestor commits](http://www.vogella.com/tutorials/Git/article.html#commitreference_parentancestor)

You can use ^ (caret) and ~ (tilde) to reference predecessor commit objects from other references. You can also combine the ^ and ~ operators. See [Using caret and tilde for commit references](http://www.vogella.com/tutorials/Git/article.html#commitreference_carettilde) for their usage.

The Git terminology is parent for ^ and ancestor for ~.

### [12.4. Using caret and tilde for commit references](http://www.vogella.com/tutorials/Git/article.html#commitreference_carettilde)

[reference]~1 describes the first predecessor of the commit object accessed via [reference]. [reference]~2 is the first predecessor of the first predecessor of the [reference] commit. [reference]~3 is the first predecessor of the first predecessor of the first predecessor of the [reference] commit, etc.

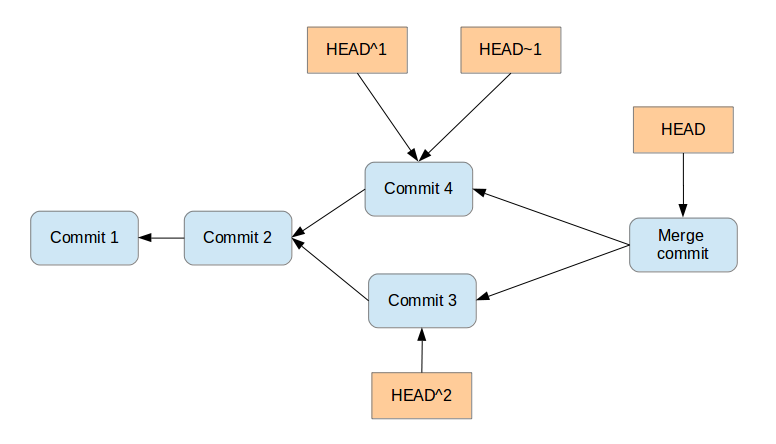
[reference]~ is an abbreviation for [reference]~1.

For example, you can use the HEAD~1 or HEAD~ reference to access the first parent of the commit to which the HEAD pointer currently points.

[reference]^1 also describes the first predecessor of the commit object accessed via [reference].

For example HEAD^ is the same as HEAD~ and is the same as HEAD~3.

The difference is that [reference]^2 describes the second parent of a commit. A merge commit typically has two predecessors. HEAD^3 means ‘the third parent of a merge’ and in most cases this won’t exist (merges are generally between two commits, though more is possible).



[reference]^ is an abbreviation for [reference]^1.

### [12.5. Commit ranges with the double dot operator](http://www.vogella.com/tutorials/Git/article.html#commitreference_ranges_doubledot)

You can also specify ranges of commits. This is useful for certain Git commands, for example, for seeing the changes between a series of commits.

The double dot operator allows you to select all commits which are reachable from a commit c2 but not from commit c1. The syntax for this is "c1..c2". A commit A is reachable from another commit B if A is a direct or indirect parent of B.

Think of c1..c2 as all commits as of c1 (not including c1) until commit c2.

For example, you can ask Git to show all commits which happened between HEAD and HEAD~4.

git log HEAD~4..HEAD

This also works for branches. To list all commits which are in the "master" branch but not in the "testing" branch, use the following command.

git log testing..master

You can also list all commits which are in the "testing" but not in the "master" branch.

git log master..testing

### [12.6. Commit ranges with the triple dot operator](http://www.vogella.com/tutorials/Git/article.html#commitreference_ranges_tripledot)

The triple dot operator allows you to select all commits which are reachable either from commit c1 or commit c2 but not from both of them.

This is useful to show all commits in two branches which have not yet been combined.

# show all commits which

# can be reached by master or testing

# but not both

git log master...testing

## [13. Git tooling](http://www.vogella.com/tutorials/Git/article.html#git_tooling)

### [13.1. The Git command line tools](http://www.vogella.com/tutorials/Git/article.html#git_tools_commmandline)

The core Git development team provides tooling for the command line via the the gitcommand. Without any arguments, this command lists its options and the most common commands. You can get help for a certain Git command via the help command online option followed by the command.

git help [command to get help for]

See all possible commands, use the git help --all command.

Git supports for several commands a short and a long version, similar to other Unix commands. The short version uses a single hyphen and the long version uses two hyphen. The following two commands are equivalent.

git commit -m "This is a message"

git commit --message "This is a message"

### [13.2. Separating parameters and file arguments in Git commands](http://www.vogella.com/tutorials/Git/article.html#git_tools_commandline_doublehyphens)

The double hyphens (--) in Git separates out any references or other options from a path (usually file names). For example, HEAD has a special meaning in Git. Using double hyphens allows you to distinguish between looking at a file called HEAD from a Git commit reference called HEAD.

In case Git can determine the correct parameters and options automatically the double hyphens can be avoided.

# seeing the git log for the HEAD file

git log -- HEAD

# seeing the git log for the HEAD reference

git log HEAD --

# if there is no HEAD file you can use HEAD as commit reference

git log HEAD

### [13.3. Graphical tools for Git](http://www.vogella.com/tutorials/Git/article.html#graphical-tools-for-git)

You can also use graphical tools. For example, the [Eclipse IDE](https://www.eclipse.org/downloads/) provides excellent support for working with Git repositories.

See [GUI Clients](http://git-scm.com/downloads/guis) for an overview of the available tools

## [14. Installation of the Git command line tooling](http://www.vogella.com/tutorials/Git/article.html#installation)

### [14.1. Ubuntu, Debian and derived systems](http://www.vogella.com/tutorials/Git/article.html#installation_ubuntu)

On Ubuntu and similar systems you can install the Git command line tool via the following command:

sudo apt-get install git

### [14.2. Fedora, Red Hat and derived systems](http://www.vogella.com/tutorials/Git/article.html#installation_fedora)

On Fedora, Red Hat and similar systems you can install the Git command line tool via the following command:

dnf install git

### [14.3. Other Linux systems](http://www.vogella.com/tutorials/Git/article.html#installation_linux)

To install Git on other Linux distributions please check the documentation of your distribution. The following listing contains the commands for the most popular ones.

# Arch Linux

sudo pacman -S git

# Gentoo

sudo emerge -av git

# SUSE

sudo zypper install git

### [14.4. Windows](http://www.vogella.com/tutorials/Git/article.html#installation_windows)

A Windows version of Git can be found on the [Git download page](http://git-scm.com/downloads). This website provides native installers for each operating system. The homepage of the Windows Git project is [git for window](https://git-for-windows.github.io/).

### [14.5. Mac OS](http://www.vogella.com/tutorials/Git/article.html#installation_mac)

The easiest way to install Git on a Mac is via the [Git download page](http://git-scm.com/downloads) and to download and run the installer for Mac OS X.

Git is also installed by default with the Apple Developer Tools on Mac OS X.

## [15. Git configuration](http://www.vogella.com/tutorials/Git/article.html#setup)

### [15.1. Git configuration levels](http://www.vogella.com/tutorials/Git/article.html#setup_configurationlevels)

The git config command allows you to configure your Git settings. These settings can be system wide, user or repository specific.

A more specific setting overwrites values in the previous level. A setting for the repository overrides the user setting and a user setting overrides a system wide setting.

#### [15.1.1. Git system-wide configuration](http://www.vogella.com/tutorials/Git/article.html#setup_systemwideconfiguration)

You can provide a system wide configuration for your Git settings. A system wide configuration is not very common. Most settings are user specific or repository specific as described in the next chapters.

On a Unix based system, Git uses the /etc/gitconfig file for this system-wide configuration. To set this up, ensure you have sufficient rights, i.e. root rights, in your OS and use the --system option for the git config command.

#### [15.1.2. Git user configuration](http://www.vogella.com/tutorials/Git/article.html#setup_userconfiguration)

Git allows you to store user settings in the .gitconfig file located in the user home directory. This is also called the global Git configuration.

For example Git stores the committer and author of a change in each commit. This and additional information can be stored in the Git user settings.

In each Git repository you can also configure the settings for this repository. User configuration is done if you include the --global option in the git config command.

#### [15.1.3. Repository specific configuration](http://www.vogella.com/tutorials/Git/article.html#setup_configuration)

You can also store repository specific settings in the .git/config file of a repository. Use the --local or use no flag at all. If neither the --system not the --global parameter is used, the setting is specific for the current Git repository.

### [15.2. User credential configuration](http://www.vogella.com/tutorials/Git/article.html#gitsetup_user)

You have to configure at least your user and email address to be able to commit to a Git repository because this information is stored in each commit.

# configure the user which will be used by Git

# this should be not an acronym but your full name

git config --global user.name "Firstname Lastname"

# configure the email address

git config --global user.email "your.email@example.org"

### [15.3. Push configuration](http://www.vogella.com/tutorials/Git/article.html#gitsetup_pushconfiguration)

If your are using Git in a version below 2.0 you should also execute the following command.

# set default so that only the current branch is pushed

git config --global push.default simple

This configures Git so that the git push command pushes only the active branch to your Git remote repository. As of Git version 2.0 this is the default and therefore it is good practice to configure this behavior.

You learn about the push command in [Push changes to another repository](http://www.vogella.com/tutorials/Git/article.html#cloneremotes_push).

### [15.4. Avoid merge commits for pulling](http://www.vogella.com/tutorials/Git/article.html#autosetuprebase)

By default, Git runs the git fetch followed by the git merge command if you use the git pull command. You can configure git to use git rebase instead of git merge for the pull command via the following setting.

# set default so that you avoid unnecessary commits

git config --global branch.autosetuprebase always

### [15.5. Allow rebasing with uncommited changes](http://www.vogella.com/tutorials/Git/article.html#allow-rebasing-with-uncommited-changes)

If you want Git to automatically save your uncommited changes before a rebase you can activate autoStash. After the rebase is done your changes will get reapplied. For an explanation of git stash please see [Stashing changes in Git](http://www.vogella.com/tutorials/Git/article.html#stash_usage).

git config --global rebase.autoStash true

Before Git v2.6 git pull --rebase didn’t respected this setting

### [15.6. Color Highlighting](http://www.vogella.com/tutorials/Git/article.html#setup_highlighting)

The following commands enables color highlighting for Git in the console.

git config --global color.ui auto

### [15.7. Setting the default editor](http://www.vogella.com/tutorials/Git/article.html#setup_editor)

By default Git uses the system default editor which is taken from the VISUAL or EDITORenvironment variables if set. You can configure a different one via the following setting.

# setup vim as default editor for Git (Linux)

git config --global core.editor vim

### [15.8. Setting the default merge tool](http://www.vogella.com/tutorials/Git/article.html#setup_mergetool)

File conflicts might occur in Git during an operation which combines different versions of the same files. In this case the user can directly edit the file to resolve the conflict.

Git allows also to configure a merge tool for solving these conflicts. You have to use third party visual merge tools like tortoisemerge, p4merge, kdiff3 etc. A Google search for these tools help you to install them on your platform. Keep in mind that such tools are not required, you can always edit the files directly in a text editor.

Once you have installed them you can set your selected tool as default merge tool with the following command.

# setup kdiff3 as default merge tool (Linux)

git config --global merge.tool kdiff3

# to install it under Ubuntu use

sudo apt-get install kdiff3

### [15.9. More settings](http://www.vogella.com/tutorials/Git/article.html#setup_allsetttings)

All possible Git settings are described under the following link: [git-config manual page](https://www.kernel.org/pub/software/scm/git/docs/git-config.html)

### [15.10. Query Git settings](http://www.vogella.com/tutorials/Git/article.html#setup_seesettings)

To query your Git settings, execute the following command:

git config --list

If you want to query the global settings you can use the following command.

git config --global --list

## [16. Configure files and directories to ignore](http://www.vogella.com/tutorials/Git/article.html#configure-files-and-directories-to-ignore)

### [16.1. Ignoring files and directories with a .gitignore file](http://www.vogella.com/tutorials/Git/article.html#ignoring-files-and-directories-with-a-.gitignore-file)

Git can be configured to ignore certain files and directories for repository operations. This is configured via one or several .gitignore files. Typically, this file is located at the root of your Git repository but it can also be located in sub-directories. In the second case the defined rules are only valid for the sub-directory and below.

You can use certain wildcards in this file. \* matches several characters. More patterns are possible and described under the following URL: [gitignore manpage](https://www.kernel.org/pub/software/scm/git/docs/gitignore.html)

For example, the following .gitignore file tells Git to ignore the bin and target directories and all files ending with a ~.

# ignore all bin directories

# matches "bin" in any subfolder

bin/

# ignore all target directories

target/

# ignore all files ending with ~

\*~

You can create the .gitignore file in the root directory of the working tree to make it specific for the Git repository.

The .gitignore file tells Git to ignore the specified files in Git commands. You can still add ignored files to the staging area of the Git repository by using the --force parameter, i.e. with the git add --force [paths] command.

This is useful if you want to add, for example, auto-generated binaries, but you need to have a fine control about the version which is added and want to exclude them from the normal workflow.

It is good practice to commit the local .gitignore file into the Git repository so that everyone who clones this repository have it.

### [16.2. Stop tracking files based on the .gitignore file](http://www.vogella.com/tutorials/Git/article.html#gitignore_untrack)

Files that are tracked by Git are not automatically removed if you add them to a .gitignorefile. Git never ignores files which are already tracked, so changes in the .gitignore file only affect new files. If you want to ignore files which are already tracked you need to explicitly remove them.

The following command demonstrates how to remove the .metadata directory and the doNotTrackFile.txt file from being tracked. This is example code, as you did not commit the corresponding files in your example, the command will not work in your Git repository.

# remove directory .metadata from git repo

git rm -r --cached .metadata

# remove file test.txt from repo

git rm --cached doNotTrackFile.txt

Adding a file to the .gitignore file does not remove the file from the repository history. If the file should also be removed from the history, have a look at the git filter-branch command which allows you to rewrite the commit history. See [Using the git filter branch command (filter-branch)](http://www.vogella.com/tutorials/Git/article.html#filterbranch_definition) for details.

### [16.3. Global (cross-repository) .gitignore settings](http://www.vogella.com/tutorials/Git/article.html#global-cross-repository-gitignore-settings)

You can also setup a global .gitignore file valid for all Git repositories via the core.excludesfile setting. The setup of this setting is demonstrated in the following code snippet.

# Create a ~/.gitignore in your user directory

cd ~/

touch .gitignore

# Exclude bin and .metadata directories

echo "bin" >> .gitignore

echo ".metadata" >> .gitignore

echo "\*~" >> .gitignore

echo "target/" >> .gitignore

# for Mac

echo ".DS\_Store" >> .gitignore

echo ".\_\*" >> .gitignore

# Configure Git to use this file

# as global .gitignore

git config --global core.excludesfile ~/.gitignore

The global .gitignore file is only locally available.

### [16.4. Local per-repository ignore rules](http://www.vogella.com/tutorials/Git/article.html#local-per-repository-ignore-rules)

You can also create local per-repository rules by editing the .git/info/exclude file in your repository. These rules are not committed with the repository so they are not shared with others.

This allows you to exclude, for example, locally generated files.

### [16.5. Tracking empty directories with Git](http://www.vogella.com/tutorials/Git/article.html#tracking_empty_directories)

Git ignores empty directories, i.e., it does not put them under version control. If you want to track an empty directory in your Git repository, it is a good practice to put a file called .gitignore in the directory. As the directory now contains a file, Git includes it into its version control mechanism.

The file could be called anything. Some people suggest to call the file .gitkeep. One problem with

this approach is that .gitkeep is unlikely to be ignored by build systems. This may result in the .gitkeep file being copied to the output repository, which is typically not desired.

## [17. Exercise - Setting up Git](http://www.vogella.com/tutorials/Git/article.html#exercise_gitsetup)

In this exercise you configure your user and email which is a required setup for Git. You also configure Git to use rebase during a pull operation which is also a common setting for Git.

Configure your user and email for Git via the following command.

# configure the user which will be used by Git

# this should be not an acronym but your full name

git config --global user.name "Firstname Lastname"

# configure the email address

git config --global user.email "your.email@example.org"

# use rebase instead of merge in the `git pull` command.

# this avoids merge commits during the pull operation

git config --global branch.autosetuprebase always

## [18. Exercise: Performing a local Git workflow](http://www.vogella.com/tutorials/Git/article.html#firstgit)

In this exercise, you learn how to create and work with a local Git repository.

Open a command shell for the operations. Some commands are Linux specific, e.g., appending to a file or creating a directory. Substitute these commands with the commands of your operating system. The comments (marked with #) before the commands explain the specific actions.

### [18.1. Create a directory](http://www.vogella.com/tutorials/Git/article.html#firstgit_directory)

The following commands create an empty directory which is used later in this exercise to contain the working tree and the Git repository.

# switch to the home directory

cd

# create a directory and switch into it

mkdir repo01

cd repo01

# create a new directory

mkdir datafiles

### [18.2. Create a new Git repository](http://www.vogella.com/tutorials/Git/article.html#firstgit_repository_creation)

You now create a new Git repository with a working tree.

Every Git repository is stored in the .git folder of the directory in which the Git repository has been created. This directory contains the complete history of the repository. The .git/configfile contains the configuration for the repository.

Use the git init command to create a Git repository in the current directory. Git does not care whether you start with an empty directory or if it contains already files.

# you should still be in the repo01 directory

cd ~/repo01

# initialize the Git repository

# for the current directory

git init

All files inside the repository folder, excluding the .git folder, are the working tree for a Git repository.

### [18.3. Create new content](http://www.vogella.com/tutorials/Git/article.html#firstgit_content)

Use the following commands to create several new files.

# switch to your Git repository

cd ~/repo01

# create an empty file in a new directory

touch datafiles/data.txt

# create a few files with content

ls > test01

echo "bar" > test02

echo "foo" > test03

### [18.4. See the current status of your repository](http://www.vogella.com/tutorials/Git/article.html#firstgit_repostatus)

The git status command shows the status of the working tree, i.e. which files have changed, which are staged and which are not part of the staging area. It also shows which files have conflicts and gives an indication what the user can do with these changes, e.g., add them to the staging area or remove them, etc.

Run it via the following command.

git status

The output looks similar to the following listing.

On branch master

Initial commit

Untracked files:

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

datafiles/

test01

test02

test03

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

### [18.5. Add changes to the staging area](http://www.vogella.com/tutorials/Git/article.html#firstgit_repoadd)

Before committing changes to a Git repository, you need to mark the changes that should be committed with the git add command. This command allows adding changes in the file system to the staging area. It creates a snapshot of the affected files. You can add all changes to the staging area with the . option or changes in individual files but specifying a file pattern as option.

# add all files to the index of the Git repository

git add .

Afterwards run the git status command again to see the current status. The following listing shows the output of this command.

On branch master

Initial commit

Changes to be committed:

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

new file: datafiles/data.txt

new file: test01

new file: test02

new file: test03

### [18.6. Change files that are staged](http://www.vogella.com/tutorials/Git/article.html#firstgit_changefiles)

In case you change one of the staged files before committing, you need to add the changes again to the staging area, to commit the new changes. This is because Git creates a snapshot of the content of a staged file. All new changes must again be staged.

# append a string to the test03 file

echo "foo2" >> test03

# see the result

git status

Validate that the new changes are not yet staged.

On branch master

Initial commit

Changes to be committed:

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

new file: datafiles/data.txt

new file: test01

new file: test02

new file: test03

Changes not staged for commit:

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

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

modified: test03

Add the new changes to the staging area.

# add all files to the index of the Git repository

git add .

Use the git status command again to see that all changes are staged.

On branch master

Initial commit

Changes to be committed:

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

new file: datafiles/data.txt

new file: test01

new file: test02

new file: test03

### [18.7. Commit staged changes to the repository](http://www.vogella.com/tutorials/Git/article.html#firstgit_repocommit)

After adding the files to the Git staging area, you can commit them to the Git repository with the git commit command. This creates a new commit object with the staged changes in the Git repository and the HEAD reference points to the new commit. The -m parameter (or its long version: --message) allows you to specify the commit message. If you leave this parameter out, your default editor is started and you can enter the message in the editor.

# commit your file to the local repository

git commit -m "Initial commit"

Git also offers a mode that lets you choose interactively which changes you want to commit. After you quit the mode you will be asked to provide a commit message in your $EDITOR.

git commit --interactive

### [18.8. Viewing the Git commit history](http://www.vogella.com/tutorials/Git/article.html#viewing-the-git-commit-history)

The Git operations you performed have created a local Git repository in the .git folder and added all files to this repository via one commit. Run the git log command to see the history.

# show the Git log for the change

git log

You see an output similar to the following.

commit 30605803fcbd507df36a3108945e02908c823828

Author: Lars Vogel <Lars.Vogel@vogella.com>

Date: Mon Dec 1 10:43:42 2014 +0100

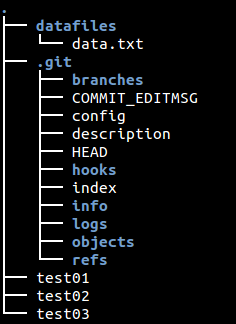
Initial commit

### [18.9. Viewing the changes of a commit](http://www.vogella.com/tutorials/Git/article.html#viewing-the-changes-of-a-commit)

Use the git show command to see the changes of a commit. If you specify a commit reference as third parameter, this is used to determine the changes, otherwise the HEAD reference is used.

### [18.10. Review the resulting directory structure](http://www.vogella.com/tutorials/Git/article.html#firstgitresult_directory)

Review the resulting directory structure. Your directory contains the Git repository as well as the Git working tree for your files. This directory structure is depicted in the following screenshot.



### [18.11. Remove files](http://www.vogella.com/tutorials/Git/article.html#firstgit_deletefile)

If you delete a file, you use the git add . command to add the deletion of a file to the staging area.

# remove the "test03" file

rm test03

# add and commit the removal

git add .

# if you use Git version < 2.0 use: git add -A .

git commit -m "Removes the test03 file"

Alternatively you can use the git rm command to delete the file from your working tree and record the deletion of the file in the staging area.

### [18.12. Revert changes in files in the working tree](http://www.vogella.com/tutorials/Git/article.html#firstgit_checkout)

Use the git checkout command to reset a tracked file (a file that was once staged or committed) to its latest staged or commit state. The command removes the changes of the file in the working tree. This command cannot be applied to files which are not yet staged or committed.

echo "useless data" >> test02

echo "another unwanted file" >> unwantedfile.txt

# see the status

git status

# remove unwanted changes from the working tree

# CAREFUL this deletes the local changes in the tracked file

git checkout test02

# unwantedstaged.txt is not tracked by Git simply delete it

rm unwantedfile.txt

If you use git status command to see that there are no changes left in the working directory.

On branch master

nothing to commit, working directory clean

Use this command carefully. The git checkout command deletes the unstaged and uncommitted

changes of tracked files in the working tree and it is not possible to restore this deletion via Git.

### [18.13. Correct the changes of the commit with git amend](http://www.vogella.com/tutorials/Git/article.html#firstgit_amend)

The git commit --amend command makes it possible to rework the changes of the last commit. It creates a new commit with the adjusted changes.

The amended commit is still available until a clean-up job removes it. But it is not included in the

git log output hence it does not distract the user. See [git reflog](http://www.vogella.com/tutorials/Git/article.html#gitreflog_definition) for details

Assume the last commit message was incorrect as it contained a typo. The following command corrects this via the --amend parameter.

# assuming you have something to commit

git commit -m "message with a tpyo here"

# amend the last commit

git commit --amend -m "More changes - now correct"

You should use the git --amend command only for commits which have not been pushed to a public branch of another Git repository. The git --amend command creates a new commit ID and people may have based their work already on the existing commit. If that would be the case, they would need to migrate their work based on the new commit.

### [18.14. Ignore files and directories with the .gitignore file](http://www.vogella.com/tutorials/Git/article.html#usegitignore)

Create the following .gitignore file in the root of your Git directory to ignore the specified directory and file.

cd ~/repo01

touch .gitignore

echo ".metadata/" >> .gitignore

echo "doNotTrackFile.txt" >> .gitignore

The above command creates the file via the command line. A more common approach is to use your

favorite text editor to create the file. This editor must save the file as plain text. Editors which do this are

for example gedit under Ubuntu or Notepad under Windows.

The resulting file looks like the following listing.

.metadata/

doNotTrackFile.txt

### [18.15. Commit the .gitignore file](http://www.vogella.com/tutorials/Git/article.html#commitgitignore)

It is good practice to commit the .gitignore file into the Git repository. Use the following commands for this.

# add the .gitignore file to the staging area

git add .gitignore

# commit the change

git commit -m "Adds .gitignore file"

## [19. Remote repositories](http://www.vogella.com/tutorials/Git/article.html#remotes)

### [19.1. What are remotes?](http://www.vogella.com/tutorials/Git/article.html#remotes_definition)

Git allows that you can synchronize your repository with more than one remote repository.

In the local repository you can address each remote repository by a shortcut. This shortcut is simply called remote. Such a remote repository point to another remote repository that can hosted on the Internet, locally or on the network.

You can specify properties for the remove, e.g. URL, branches to fetch or branches to push.

Think of remotes as shorter bookmarks for repositories. You can always connect to a remote repository if

you know its URL and if you have access to it. Without remotes the user would have to type the URL for

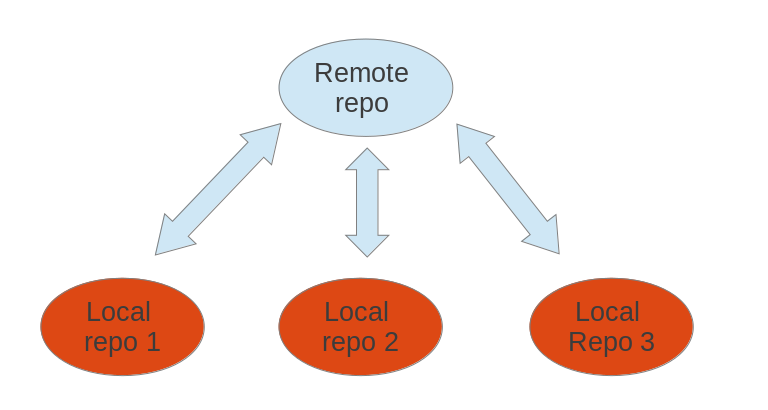
each and every command which communicates with another repository.

It is possible that users connect their individual repositories directly, but a typically Git workflow involves

one or more remote repositories which are used to synchronize the individual repository.

Typically the remote repository which is used for synchronization is located on a server which is

always available.



### [19.2. Bare repositories](http://www.vogella.com/tutorials/Git/article.html#bareremotes_definition)

A remote repository on a server typically does not require a working tree. A Git repository without a working tree is called a bare repository. You can create such a repository with the --bare option. The command to create a new empty bare remote repository is displayed below.

# create a bare repository

git init --bare

By convention the name of a bare repository should end with the .git extension.

To create a bare Git repository in the Internet you would, for example, connect to your server via the SSH protocol or you use some Git hosting platform, e.g., GitHub.com.

### [19.3. Convert a Git repository to a bare repository](http://www.vogella.com/tutorials/Git/article.html#bareremotes_convert)

Converting a normal Git repository to a bare repository is not directly support by Git.

You can convert it manually by moving the content of the .git folder into the root of the repository and by removing all others files from the working tree. Afterwards you need to update the Git repository configuration with the git config core.bare true command.

As this is officially not supported, you should prefer cloning a repository with the --bareoption.

### [19.4. Cloning a repository](http://www.vogella.com/tutorials/Git/article.html#remotes_cloneoperation)

The git clone command copies an existing Git repository. This copy is a working Git repository with the complete history of the cloned repository. It can be used completely isolated from the original repository.

Git supports several transport protocols to connect to other Git repositories; the native protocol for Git is also called git.

The following command clones an existing repository using the Git protocol. The Git protocol uses the port 9148 which might be blocked by firewalls.

# switch to a new directory

mkdir ~/online

cd ~/online

# clone online repository

git clone git://github.com/vogella/gitbook.git

If you have SSH access to a Git repository, you can also use the ssh protocol. The name preceding @ is the user name used for the SSH connection.

# clone online repository

git clone ssh://git@github.com/vogella/gitbook.git

# older syntax

git clone git@github.com:vogella/gitbook.git

Alternatively you could clone the same repository via the http protocol.

# the following will clone via HTTP

git clone http://github.com/vogella/gitbook.git

### [19.5. Adding remote repositories](http://www.vogella.com/tutorials/Git/article.html#remote_add)

If you clone a repository, Git implicitly creates a remote named origin by default. The originremote links back to the cloned repository.

You can push changes to this repository via git push as Git uses origin as default. Of course, pushing to a remote repository requires write access to this repository.

You can add more remotes via the git remote add [name] [URL\_to\_Git\_repo] command. For example, if you cloned the repository from above via the Git protocol, you could add a new remote with the name github\_http for the http protocol via the following command.

# add the HTTPS protocol

git remote add github\_http https://vogella@github.com/vogella/gitbook.git

### [19.6. Rename remote repositories](http://www.vogella.com/tutorials/Git/article.html#remote_rename)

To rename an existing remote repository use the git remote rename command. This is demonstrated by the following listing.

# rename the existing remote repository from

# github\_http to github\_testing

git remote rename github\_http github\_testing

If you create a Git repository from scratch with the git init command, the origin remote is not created automatically.

### [19.7. Remote operations via HTTP](http://www.vogella.com/tutorials/Git/article.html#remote_httpprotocol)

HTTP as Git protocol proxy support in Git It is possible to use the HTTP protocol to clone Git repositories. This is especially helpful if your firewall blocks everything except HTTP or HTTPS.

git clone http://git.eclipse.org/gitroot/platform/eclipse.platform.ui.git

For secured SSL encrypted communication you should use the SSH or HTTPS protocol in order to guarantee security.

### [19.8. Using a proxy](http://www.vogella.com/tutorials/Git/article.html#remote_httpproxy)

Git also provides support for HTTP access via a proxy server. The following Git command could, for example, clone a repository via HTTP and a proxy. You can either set the proxy variable in general for all applications or set it only for Git.

The following listing configures the proxy via environment variables.

# Linux and Mac

export http\_proxy=http://proxy:8080

export https\_proxy=https://proxy:8443

# Windows

set http\_proxy http://proxy:8080

set https\_proxy http://proxy:8080

git clone http://git.eclipse.org/gitroot/platform/eclipse.platform.ui.git

The following listing configures the proxy via Git config settings.

# set proxy for git globally

git config --global http.proxy http://proxy:8080

# to check the proxy settings

git config --get http.proxy

# just in case you need to you can also revoke the proxy settings

git config --global --unset http.proxy

### [19.9. Adding a remote repository](http://www.vogella.com/tutorials/Git/article.html#adding-a-remote-repository)

You add as many remotes to your repository as desired. For this you use the git remote addcommand.

You created a new Git repository from scratch earlier. Use the following command to add a remote to your new bare repository using the origin name.

# add ../remote-repository.git with the name origin

git remote add origin ../remote-repository.git

### [19.10. Synchronizing with remote repositories](http://www.vogella.com/tutorials/Git/article.html#remotes_remote_synchronize)

You can synchronize your local Git repository with remote repositories. These commands are covered in detail in later sections but the following command demonstrates how you can send changes to your remote repository.

# do some changes

echo "I added a remote repo" > test02

# commit

git commit -a -m "This is a test for the new remote origin"

# to push use the command:

# git push [target]

# default for [target] is origin

git push origin

### [19.11. Show the existing remotes](http://www.vogella.com/tutorials/Git/article.html#remotes_showremote)

To see the existing definitions of the remote repositories, use the following command.

# show the details of the remote repo called origin

git remote show origin

To see the details of the remotes, e.g., the URL use the following command.

# show the existing defined remotes

git remote

# show details about the remotes

git remote -v

### [19.12. Push changes to another repository](http://www.vogella.com/tutorials/Git/article.html#cloneremotes_push)

The git push command allows you to send data to other repositories. By default it sends data from your current branch to the same branch of the remote repository.

By default you can only push to bare repositories (repositories without working tree). Also you can only push a change to a remote repository which results in a fast-forward merge. See [Fast-forward merge](http://www.vogella.com/tutorials/Git/article.html#gitmerge_fastforward) to learn about fast-forward merges.

See [Push changes of a branch to a remote repository](http://www.vogella.com/tutorials/Git/article.html#gitpushbranch) for details on pushing branches or the [Git push manpage](https://www.kernel.org/pub/software/scm/git/docs/git-push.html) for general information.

### [19.13. Pull changes from a remote repository](http://www.vogella.com/tutorials/Git/article.html#cloneremotes_pull)

The git pull command allows you to get the latest changes from another repository for the current branch.

The git pull command is actually a shortcut for git fetch followed by the git merge or the git rebase command depending on your configuration. In [Avoid merge commits for pulling](http://www.vogella.com/tutorials/Git/article.html#autosetuprebase)you configured your Git repository so that git pull is a fetch followed by a rebase. See [Fetch](http://www.vogella.com/tutorials/Git/article.html#gitfetch_intro)for more information about the fetch command.

## [20. Exercise: Working with a (local) remote repository](http://www.vogella.com/tutorials/Git/article.html#exercise_workingwithremotes)

This exercise is based on [Exercise: Performing a local Git workflow](http://www.vogella.com/tutorials/Git/article.html#firstgit). You now create a local bare repository based on your existing Git repository. In order to simplify the examples, the Git repository is hosted locally in the filesystem and not on a server in the Internet.

Afterwards you pull from and push to your bare repository to synchronize changes between your repositories.

### [20.1. Create a bare Git repository via the clone operation](http://www.vogella.com/tutorials/Git/article.html#remotes_setupexercise)

Execute the following commands to create a bare repository based on your existing Git repository.

# switch to the first repository

cd ~/repo01

# create a new bare repository by cloning the first one

git clone --bare . ../remote-repository.git

# check the content of the git repo, it is similar

# to the .git directory in repo01

# files might be packed in the bare repository

ls ~/remote-repository.git

### [20.2. Exercise: Clone your bare repository](http://www.vogella.com/tutorials/Git/article.html#remotes_remoteadd)

Clone your bare repository and checkout a working tree in a new directory via the following commands.

# switch to home

cd ~

# make new directory

mkdir repo02

# switch to new directory

cd ~/repo02

# clone

git clone ../remote-repository.git .

### [20.3. Exercise: Using the push command](http://www.vogella.com/tutorials/Git/article.html#exercise_cloneremotes_push)

Make some changes in one of your non-bare local repositories and push them to your bare repository via the following commands.

# make some changes in the first repository

cd ~/repo01

# make some changes in the file

echo "Hello, hello. Turn your radio on" > test01

echo "Bye, bye. Turn your radio off" > test02

# commit the changes, -a will commit changes for modified files

# but will not add automatically new files

git commit -a -m "Some changes"

# push the changes

git push ../remote-repository.git

### [20.4. Exercise: Using the pull command](http://www.vogella.com/tutorials/Git/article.html#exercise_cloneremotes_pull)

To test the git pull in your example Git repositories, switch to other non-bare local repository. Pull in the recent changes from the remote repository. Afterwards make some changes and push them again to your remote repository.

# switch to second directory

cd ~/repo02

# pull in the latest changes of your remote repository

git pull

# make changes

echo "A change" > test01

# commit the changes

git commit -a -m "A change"

# push changes to remote repository

# origin is automatically created as we cloned original from this repository

git push origin

You can pull in the changes in your first example repository with the following commands.

# switch to the first repository and pull in the changes

cd ~/repo01

git pull ../remote-repository.git/

# check the changes

git status

## [21. Using Branches](http://www.vogella.com/tutorials/Git/article.html#using-branches)

### [21.1. What are branches?](http://www.vogella.com/tutorials/Git/article.html#gitbranch_def)

Git allows you to create branches, i.e. named pointers to commits. You can work on different branches independently from each other. The default branch is most often called master.

A branch pointer in Git is 41 bytes large, 40 bytes of characters and an additional new line character. Therefore, the creating of branches in Git is very fast and cheap in terms of resource consumption. Git encourages the usage of branches on a regular basis.

If you decide to work on a branch, you checkout this branch. This means that Git populates the working tree with the version of the files from the commit to which the branch points and moves the HEAD pointer to the new branch.

As explained in [Summary of the core Git terminology](http://www.vogella.com/tutorials/Git/article.html#gitterminology) HEAD is a symbolic reference usually pointing to the branch which is currently checked out.

### [21.2. List available branches](http://www.vogella.com/tutorials/Git/article.html#gitbranch_listbranches)

The git branch command lists all local branches. The currently active branch is marked with \*.

# lists available branches

git branch

If you want to see all branches (including remote-tracking branches), use the -a for the git branch command. See [Remote tracking branches](http://www.vogella.com/tutorials/Git/article.html#gitremotebranch_overview) for information about remote-tracking branches.

# lists all branches including the remote branches

git branch -a

The -v option lists more information about the branches.

In order to list branches in a remote repository use the git branch -r command as demonstrated in the following example.

# lists branches in the remote repositories

git branch -r

### [21.3. Create new branch](http://www.vogella.com/tutorials/Git/article.html#gitbranch_createnewbranch)

You can create a new branch via the git branch [newname] command. This command allows to specify the commit (commit id, tag, remote or local branch) to which the branch pointer original points. If not specified, the commit to which the HEAD reference points is used to create the new branch.

# syntax: git branch <name> <hash>

# <hash> in the above is optional

git branch testing

### [21.4. Checkout branch](http://www.vogella.com/tutorials/Git/article.html#gitbranch_checkout)

To start working in a branch you have to checkout the branch. If you checkout a branch, the HEAD pointer moves to the last commit in this branch and the files in the working tree are set to the state of this commit.

The following commands demonstrate how you switch to the branch called testing, perform some changes in this branch and switch back to the branch called master.

# switch to your new branch

git checkout testing

# do some changes

echo "Cool new feature in this branch" > test01

git commit -a -m "new feature"

# switch to the master branch

git checkout master

# check that the content of

# the test01 file is the old one

cat test01

To create a branch and to switch to it at the same time you can use the git checkout command with the -b parameter.

# create branch and switch to it

git checkout -b bugreport12

# creates a new branch based on the master branch

# without the last commit

git checkout -b mybranch master~1

### [21.5. Rename a branch](http://www.vogella.com/tutorials/Git/article.html#git_rename_branch)

Renaming a branch can be done with the following command.

# rename branch

git branch -m [old\_name] [new\_name]

### [21.6. Delete a branch](http://www.vogella.com/tutorials/Git/article.html#gitdeletebrach)

To delete a branch which is not needed anymore, you can use the following command. You may get an error message that there are uncommited changes if you did the previous examples step by step. Use force delete (uppercase -D) to delete it anyway.

# delete branch testing

git branch -d testing

# force delete testing

git branch -D testing

# check if branch has been deleted

git branch

### [21.7. Push changes of a branch to a remote repository](http://www.vogella.com/tutorials/Git/article.html#gitpushbranch)

You can push the changes in a branch to a remote repository by specifying the target branch. This creates the target branch in the remote repository if it does not yet exist.

If you do not specify the remote repository, the origin is used as default

# push current branch to a branch called "testing" to remote repository

git push origin testing

# switch to the testing branch

git checkout testing

# some changes

echo "News for you" > test01

git commit -a -m "new feature in branch"

# push current HEAD to origin

git push

# make new branch

git branch anewbranch

# some changes

echo "More news for you" >> test01

git commit -a -m "a new commit in a feature branch"

# push anewbranch to the master in the origin

git push origin anewbranch:master

# get the changes into your local master

git checkout master

git pull

This way you can decide which branches you want to push to other repositories and which should be local branches. You learn more about branches and remote repositories in [Remote tracking branches](http://www.vogella.com/tutorials/Git/article.html#gitremotebranch_overview).

### [21.8. Switching branches with untracked files](http://www.vogella.com/tutorials/Git/article.html#gitbranch_untrackedfiles)

Untracked files (never added to the staging area) are unrelated to any branch. They exist only in the working tree and are ignored by Git until they are committed to the Git repository. This allows you to create a branch for unstaged and uncommitted changes at any point in time.

### [21.9. Switching branches with uncommitted changes](http://www.vogella.com/tutorials/Git/article.html#gitbranch_dirtyfiles)

Similar to untracked files you can switch branches with unstaged or staged modifications which are not yet committed.

You can switch branches if the modifications do not conflict with the files from the branch.

If Git needs to modify a changed file during the checkout of a branch, the checkout fails with a "checkout conflict" error. This avoids that you lose changes in your files.

In this case the changes must be committed, reverted or stashed (see [The git stash command](http://www.vogella.com/tutorials/Git/article.html#stash_usage1)). You can also always create a new branch based on the current HEAD.

### [21.10. Differences between branches](http://www.vogella.com/tutorials/Git/article.html#gitdiffbranches)

To see the difference between two branches you can use the following command.

# shows the differences between

# current head of master and your\_branch

git diff master your\_branch

You can use commit ranges as described in [Commit ranges with the double dot operator](http://www.vogella.com/tutorials/Git/article.html#commitreference_ranges_doubledot) and [Commit ranges with the triple dot operator](http://www.vogella.com/tutorials/Git/article.html#commitreference_ranges_tripledot). For example, if you compare a branch called your\_branch with the master branch the following command shows the changes in your\_branchand master since these branches diverged.

# shows the differences in your

# branch based on the common

# ancestor for both branches

git diff master...your\_branch

See [Viewing changes with git diff and git show](http://www.vogella.com/tutorials/Git/article.html#gitdiffchapter) for more examples of the git diff command.

## [22. Using tags in Git](http://www.vogella.com/tutorials/Git/article.html#using-tags-in-git)

## [23. Using Tags](http://www.vogella.com/tutorials/Git/article.html#using-tags)

### [23.1. What are tags?](http://www.vogella.com/tutorials/Git/article.html#tagging)

Git has the option to tag a commit in the repository history so that you find it easier at a later point in time. Most commonly, this is used to tag a certain version which has been released.

If you tag a commit, you create an annotated or lightweight tag.

#### [23.1.1. Lightweight and annotated tags](http://www.vogella.com/tutorials/Git/article.html#tagging_lightweight_annotated)

Git supports two different types of tags, lightweight and annotated tags.

A lightweight tag is a pointer to a commit, without any additional information about the tag. An annotated tag contains additional information about the tag, e.g., the name and email of the person who created the tag, a tagging message and the date of the tagging. Annotated tags can also be signed and verified with GNU Privacy Guard (GPG).

#### [23.1.2. Naming conventions for tags](http://www.vogella.com/tutorials/Git/article.html#tagging_namingconventions)

Tags are frequently used to tag the state of a release of the Git repository. In this case they are typically called release tags.

Convention is that release tags are labeled based on the [major].[minor].[patch] naming scheme, for example "1.0.0". Several projects also use the "v" prefix.

The idea is that the patch version is incremented if (only) backwards compatible bug fixes are introduced, the minor version is incremented if new, backwards compatible functionality is introduced to the public API and the major version is incremented if any backwards incompatible changes are introduced to the public API.

For the detailed discussion on naming conventions please see the following URL: [Semantic versioning](http://semver.org/).

### [23.2. List tags](http://www.vogella.com/tutorials/Git/article.html#tagging_list)

You can list the available tags via the following command:

git tag

### [23.3. Search by pattern for a tag](http://www.vogella.com/tutorials/Git/article.html#tag_searchbypattern)

You can use the -l parameter in the git tag command to search for a pattern in the tag.

git tag -l <pattern>

### [23.4. Creating lightweight tags](http://www.vogella.com/tutorials/Git/article.html#tagging_createlightweight)

To create a lightweight tag don’t use the -m, -a or -s option.

The term build describes the conversion of your source code into another state, e.g., converting Java sources to an executable JAR file. Lightweight tags in Git are often used to identify the input for a build. Frequently this does not require additional information other than a build identifier or the timestamp.

# create lightweight tag

git tag 1.7.1

# see the tag

git show 1.7.1

### [23.5. Creating annotated tags](http://www.vogella.com/tutorials/Git/article.html#tagging_createannoted)

You can create a new annotated tag via the git tag -a command. An annotated tag can also be created using the -m parameter, which is used to specify the description of the tag. The following command tags the current active HEAD.

# create tag

git tag 1.6.1 -m 'Release 1.6.1'

# show the tag

git show 1.6.1

You can also create tags for a certain commit id.

git tag 1.5.1 -m 'version 1.5' [commit id]

### [23.6. Creating signed tags](http://www.vogella.com/tutorials/Git/article.html#tagging_signedtags)

You can use the option -s to create a signed tag. These tags are signed with GNU Privacy Guard (GPG) and can also be verified with GPG. For details on this please see the following URL: [Git tag manpage](https://www.kernel.org/pub/software/scm/git/docs/git-tag.html).

### [23.7. Checkout tags](http://www.vogella.com/tutorials/Git/article.html#tagging_checkout)

If you want to use the code associated with the tag, use:

git checkout <tag\_name>

### [23.8. Push tags](http://www.vogella.com/tutorials/Git/article.html#tagging_pushtags)

By default the git push command does not transfer tags to remote repositories. You explicitly have to push the tag with the following command.

# push a tag or branch called tagname

git push origin [tagname]

# to explicitly push a tag and not a branch

git push origin tag <tagname>

# push all tags

git push --tags

### [23.9. Delete tags](http://www.vogella.com/tutorials/Git/article.html#tagging_delete)

You can delete tags with the -d parameter. This deletes the tag from your local repository. By default Git does not push tag deletions to a remote repository, you have to trigger that explicitly.

The following commands demonstrate how to push a tag deletion.

# delete tag locally

git tag -d 1.7.0

# delete tag in remote repository

# called origin

git push origin :refs/tags/1.7.0

## [24. Comparing changes](http://www.vogella.com/tutorials/Git/article.html#analyzechanges)

### [24.1. Listing changed files](http://www.vogella.com/tutorials/Git/article.html#gitstatus)

The git status command shows the current status of your repository and possible actions which you can perform.

It shows which files have changed, which are staged and which are not part of the staging area. It also shows which files have merge conflicts and gives an indication what the user can do with these changes, e.g., add them to the staging area or remove them, etc.

git status -u shows all untracked files. Otherwise, if you have a new directory with severals files, only

the directory is shown

### [24.2. Example: Using git status](http://www.vogella.com/tutorials/Git/article.html#gitstatus_example)

The following commands create some changes in your Git repository.

*Make some changes in your working tree*

# assumes that the test01 and test02 files exist

# and have been committed in the past

echo "This is a new change to the file" > test01

echo "and this is another new change" > test02

# create a new file

ls > newfileanalyzis.txt

Now use the status command.

git status

The output of the command looks like the following listing.

# On branch master

# Your branch is ahead of 'origin/master' by 1 commit.

# (use "git push" to publish your local commits)

#

# Changes not staged for commit:

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

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

#

# modified: test01

# modified: test02

#

# Untracked files:

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

#

# newfileanalyzis.txt

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

### [24.3. Using git diff](http://www.vogella.com/tutorials/Git/article.html#gitdiff_using)

The git diff command allows you to compare changes between commits, the staging area and working tree, etc. Via an optional third parameter you can specify a path to filter the displayed changes path can be a file or directory git diff [path].

The following example code demonstrate the usage of the git diff command.

*Make some changes in your working tree*

echo "This is a change" > test01

echo "and this is another change" > test02

*Use the git diff command*

git diff

git diff --cached

git diff COMMMIT\_REF1 COMMMIT\_REF2

git diff -- [file\_reference]

|  |  |
| --- | --- |
|  | shows the changes introduced in the working tree compared with the staging area |
|  | shows the differences between the staging area and the last commit |
|  | shows the differences introduced between two commits references |
|  | shows the differences introduced in the working tree compared with the staging area for [file\_reference] |

## [25. Analyzing the commit history with git log](http://www.vogella.com/tutorials/Git/article.html#analyzechanges)

### [25.1. Using git log](http://www.vogella.com/tutorials/Git/article.html#analyzechanges_log)

The git log command shows the history of the Git repository. If no commit reference is specified it starts from the commit referred to by the HEAD pointer.

git log

git log HEAD~10

git log COMMIT\_REF

|  |  |
| --- | --- |
|  | shows the history of commits starting from the HEAD~10 commit |
|  | shows the history of commits starting from the COMMIT\_REF commit |

### [25.2. Helpful parameters for git log](http://www.vogella.com/tutorials/Git/article.html#git_log_options)

The following gives an overview of useful parameters for the git log command.

git log --oneline

git log --abbrev-commit

git log --graph --oneline

git log --decorate

|  |  |
| --- | --- |
|  | --oneline - fits the output of the git log command in one line. --online is a shorthand for "--pretty=oneline --abbrev-commit" |
|  | --abbrev-commit - the log command uses shorter versions of the SHA-1 identifier for a commit object but keeps the SHA-1 unique. This parameter uses 7 characters by default, but you can specify other numbers, e.g., --abbrev-commit --abbrev=4. |
|  | graph - draws a text-based graphical representation of the branches and the merge history of the Git repository. |
|  | decorate - adds symbolic pointers to the log output |

For more options on the git log command see the [Git log manpage](https://www.kernel.org/pub/software/scm/git/docs/git-log.html).

### [25.3. View the change history of a file](http://www.vogella.com/tutorials/Git/article.html#analyzechangesg_changesinafile)

To see changes in a file you can use the -p option in the git log command.

git log -- [file\_reference]

git log -p -- [file\_reference]

git log --follow -p -- [file\_reference]

|  |  |
| --- | --- |
|  | - shows the list of commits for this file |
|  | - the -p parameter triggers that the diffs of each commit is shown |
|  | - --follow allow include renames in the log output |

### [25.4. Configuring output format](http://www.vogella.com/tutorials/Git/article.html#analyzechanges_output)

You can use the --pretty parameter to configure the output.

# command must be issued in one line, do not enter the line break

git log --pretty=format:'%Cred%h%Creset %d%Creset %s %Cgreen(%cr)

%C(bold blue)<%an>%Creset' --abbrev-commit

This command creates the output.

### [25.5. Filtering based on the commit message via regular expressions](http://www.vogella.com/tutorials/Git/article.html#analyzechanges_grep)

You can filter the output of the git log command to commits whose commit message, or reflog entry, respectively, matches the specified regular expression pattern with the --grep=<pattern> and --grep-reflog=<pattern> option.

For example the following command instructs the log command to list all commits which contain the word "workspace" in their commit message.

git log --oneline --grep="workspace"

There is also the --invert-grep=<pattern> option. When this option is used, git log lists the commits that don’t match the specified pattern.

### [25.6. Filtering the log output based on author or committer](http://www.vogella.com/tutorials/Git/article.html#analyzechanges_user)

You can use the --author=<pattern> or --committer=<pattern> to filter the log output by author or committer. You do not need to use the full name, if a substring matches, the commit is included in the log output.

The following command lists all commits with an author name containing the word "Vogel".

git log --author="Vogel"

## [26. Viewing changes with git diff and git show](http://www.vogella.com/tutorials/Git/article.html#gitdiffchapter)

### [26.1. See the differences introduced by a commit](http://www.vogella.com/tutorials/Git/article.html#analyzechanges_showcommitchanges)

To see the changes introduced by a commit use the following command.

git show <commit\_id>

### [26.2. See the difference between two commits](http://www.vogella.com/tutorials/Git/article.html#gitdiff_comparetwocommits)

To see the differences introduced between two commits you use the git diff command specifying the commits. For example, the following command shows the differences introduced in the last commit.

# directly between two commits

git diff HEAD~1 HEAD

# using commit ranges

git diff HEAD~1..HEAD

### [26.3. See the files changed by a commit](http://www.vogella.com/tutorials/Git/article.html#analyzechanges_file)

To see the files which have been changed in a commit use the git diff-tree command. The name-only tells the command to show only the names of the files.

git diff-tree --name-only -r <commit\_id>

## [27. Using the Git blame command](http://www.vogella.com/tutorials/Git/article.html#using-the-git-blame-command)

### [27.1. Analyzing line changes with git blame](http://www.vogella.com/tutorials/Git/article.html#git_blame_def)

Using the Git log command and filtering the history is a useful tool for inspecting the project history. However, if you look at a particular file and find a bug in a particular line of code you would like to instantly know who was the last person who changed this line of code. Additionally, you would like to know why the developer did that i.e. locate the commit in which the change was done.

In Git, this feature is called git blame or git annotate. The git blame command allows you to see which commit and author modified a file on a per line base. That is very useful to identify the person or the commit which introduced a change.

### [27.2. Example: git blame](http://www.vogella.com/tutorials/Git/article.html#git_blame_example)

The following code snippet demonstrates the usage of the git blame command.

# git blame shows the author and commit per

# line of a file

git blame [filename]

# the -L option allows limiting the selection

# for example by line number

# only show line 1 and 2 in git blame

git blame -L 1,2 [filename]

The git blame command can also ignore whitespace changes with the -w parameter.

## [28. Commit history of a repository or certain files](http://www.vogella.com/tutorials/Git/article.html#gitk)

[Gitk](http://git-scm.com/docs/gitk) can be used to visualize the history of a repository of certain files.

In some cases simply using git blame is not sufficient in order to see all details of certain changes. You can navigate to the file location in the target git repository and use the gitk [filename] command to see all commits of a file in a clear UI.

In this screenshot we can see all commits of the ShowViewHandler.java by using the gitk ShowViewHandler.java command:

## [29. git shortlog for release announcements](http://www.vogella.com/tutorials/Git/article.html#gitshortlog)

git shortlog The git shortlog command summarizes the git log output. It groups all commits by author and includes the first line of the commit message.

The -s option suppresses the commit message and provides a commit count. The -n option sorts the output based on the number of commits by author.

# gives a summary of the changes by author

git shortlog

# compressed summary

# -s summary, provides a commit count summary only

# -n sorted by number instead of name of the author

git shortlog -sn

This command also allows you to see the commits done by a certain author or committer.

# see the commits by the author "Lars Vogel"

git shortlog --author="Lars Vogel"

# see the commits by the author "Lars Vogel"

# restricted by the last years

git shortlog --author="Lars Vogel" --since=2years

# see the number of commits by the author "Lars Vogel"

git shortlog -s --author="Lars Vogel" --since=2years

## [30. Stashing changes in Git](http://www.vogella.com/tutorials/Git/article.html#stash_usage)

### [30.1. The git stash command](http://www.vogella.com/tutorials/Git/article.html#stash_usage1)

Git provides the git stash command which allows you to record the current state of the working directory and the staging area and to revert to the last committed revision.

This allows you to pull in the latest changes or to develop an urgent fix. Afterwards you can restore the stashed changes, which will reapply the changes to the current version of the source code.

### [30.2. When to use git stash](http://www.vogella.com/tutorials/Git/article.html#stash_usage2)

In general using the stash command should be the exception in using Git. Typically, you would create new branches for new features and switch between branches. You can also commit frequently in your local Git repository and use interactive rebase to combine these commits later before pushing them to another Git repository.

Even if you prefer not to use branches, you can avoid using the git stash command. In this case you commit the changes you want to put aside and amend the commit with the next commit. If you use the approach of creating a commit, you typically put a marker in the commit message to mark it as a draft, e.g., "[DRAFT] implement feature x".

### [30.3. Example: Using the git stash command](http://www.vogella.com/tutorials/Git/article.html#stash_example)

The following commands will save a stash and reapply them after some changes.

# create a stash with uncommitted changes

git stash

# do changes to the source, e.g., by pulling

# new changes from a remote repo

# afterwards, re-apply the stashed changes

# and delete the stash from the list of stashes

git stash pop

It is also possible to keep a list of stashes.

# create a stash with uncommitted changes

git stash save

# see the list of available stashes

git stash list

# result might be something like:

stash@{0}: WIP on master: 273e4a0 Resize issue in Dialog

stash@{1}: WIP on master: 273e4b0 Silly typo in Classname

stash@{2}: WIP on master: 273e4c0 Silly typo in Javadoc

# you can use the ID to apply a stash

git stash apply stash@{0}

# or apply the latest stash and delete it afterwards

git stash pop

# you can also remove a stashed change

# without applying it

git stash drop stash@{0}

# or delete all stashes

git stash clear

### [30.4. Create a branch from a stash](http://www.vogella.com/tutorials/Git/article.html#stash_branchexample)

You can also create a branch for your stash if you want to continue to work on the stashed changes in a branch. This can be done with the following command.

# create a new branch from your stack and

# switch to it

git stash branch newbranchforstash

## [31. Remove untracked files with git clean](http://www.vogella.com/tutorials/Git/article.html#gitclean)

### [31.1. Removing untracked files](http://www.vogella.com/tutorials/Git/article.html#gitclean_command)

If you have untracked files in your working tree which you want to remove, you can use the git clean command.

### [31.2. Example: Using git clean](http://www.vogella.com/tutorials/Git/article.html#gitclean_example)

The following commands demonstrate the usage of the git clean command.

# create a new file with content

echo "this is trash to be deleted" > test04

# make a dry-run to see what would happen

# -n is the same as --dry-run

git clean -n

# delete, -f is required if

# variable clean.requireForce is not set to false

git clean -f

# use -d flag to delete new directories

# use -x to delete hidden files, e.g., ".example"

git clean -fdx

## [32. Revert uncommitted changes in tracked files](http://www.vogella.com/tutorials/Git/article.html#undochanges)

### [32.1. Use cases](http://www.vogella.com/tutorials/Git/article.html#undochanges_usecase)

If you have a tracked file in Git, you can always recreate the file content based on the staging area or based on a previous commit. You can also remove staged changes from the staging area to avoid that these changes are included in the next commit. This chapter explain you how you can do this.

### [32.2. Remove staged changes from the staging area](http://www.vogella.com/tutorials/Git/article.html#undochanges_reset)

Staging area, remove staged changes You can use the git reset [paths] command to remove staged changes from the staging area. This means that git reset [paths] is the opposite of git add [paths]. It avoids that the changes are included in the next commit. The changes are still available in the working tree, e.g., you will not lose your changes and can stage and commit them at a later point.

In the following example you create a new file and change an existing file. Both changes are staged.

# do changes

touch unwantedstaged.txt

echo "more.." >> test02

// add changes to staging area

git add unwantedstaged.txt

git add test02

# see the status

git status

The output of git status command should look similar to the following.

On branch master

Changes to be committed:

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

modified: test02

new file: unwantedstaged.txt

Remove the changes from the staging area with the following command.

# remove test02 from the staging area

git reset test02

# remove unwantedstaged.txt from the staging area

git reset unwantedstaged.txt

Use the git status command to see the result.

On branch master

Changes not staged for commit:

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

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

modified: test02

Untracked files:

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

unwantedstaged.txt

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

The git reset behaves differently depending on the options you provide. To learn more about the git reset command see [[gitreset]](http://www.vogella.com/tutorials/Git/article.html#gitreset).

### [32.3. Remove changes in the working tree](http://www.vogella.com/tutorials/Git/article.html#undochanges_checkout)

Changes in the working tree which are not staged can be undone with git checkoutcommand. This command resets the file in the working tree to the latest staged version. If there are no staged changes, the latest committed version is used for the restore operation.

# delete a file

rm test01

# revert the deletion

git checkout -- test01

# note git checkout test01 also works but using

# two - ensures that Git understands that test01

# is a path and not a parameter

# change a file

echo "override" > test01

# restore the file

git checkout -- test01

For example, you can restore the content of a directory called data with the following command.

git checkout -- data

### [32.4. Remove changes in the working tree and the staging area](http://www.vogella.com/tutorials/Git/article.html#undochanges_directorydeletion)

If you want to undo a staged but uncommitted change, you use the git checkout [commit-pointer] [paths] command. This version of the command resets the working tree and the staged area.

The following demonstrates the usage of this to restore a delete directory.

# create a demo directory

mkdir checkoutheaddemo

touch checkoutheaddemo/myfile

git add .

git commit -m "Adds new directory"

# now delete the directory and add the change to

# the staging area

rm -rf checkoutheaddemo

# Use git add . -A for Git version < 2.0

git add .

# restore the working tree and reset the staging area

git checkout HEAD -- your\_dir\_to\_restore

The additional commit pointer parameter instructs the git checkout command to reset the working tree and to also remove the staged changes.

### [32.5. Remove staging area based on last commit change](http://www.vogella.com/tutorials/Git/article.html#undochanges_indexexisting)

When you have added the changes of a file to the staging area, you can also revert the changes in the staging area base on the last commit.

# some nonsense change

echo "change which should be removed later" > test01

# add the file to the staging area

git add test01

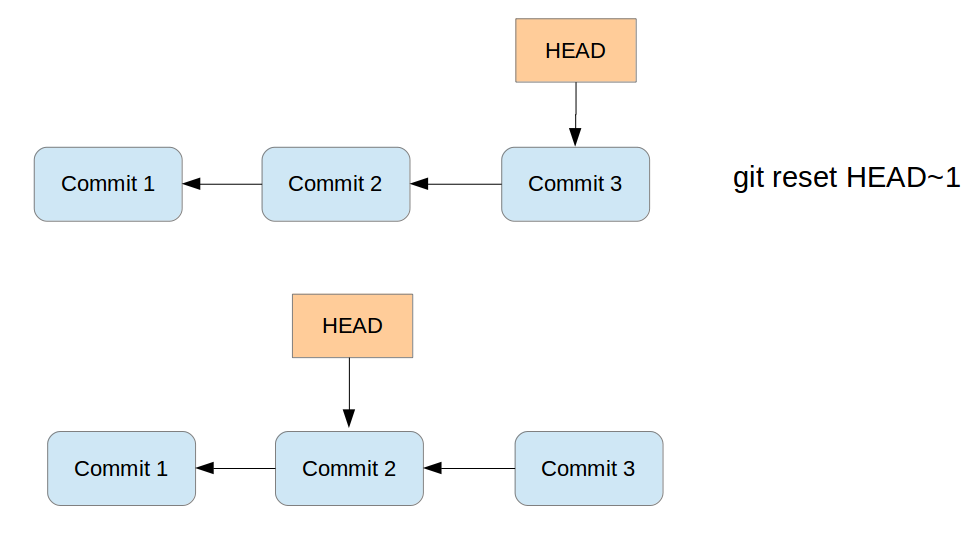
# restores the file based on HEAD in the staging area

git reset HEAD test01

## [33. Using Git reset](http://www.vogella.com/tutorials/Git/article.html#resetcommitsdef)

### [33.1. Moving the HEAD and branch pointer](http://www.vogella.com/tutorials/Git/article.html#movebranchpointer)

Sometimes you want to change the commmit your branch pointer is pointing to. The git resetcommand allows you to manually set the current HEAD pointer (and its associated branch) to a specified commit. This is for example useful to undo a particular change or to build up a different commit history.



All commits which were originally pointed to by the HEAD pointer and the commit pointed to by HEAD after the reset, are reseted, e.g., not directly visible anymore from the current HEAD and branch pointer.

Via parameters you can decide what you happen to the changes in the working tree and changes which were included in the commits between the original commit and the commit now referred to by the HEAD pointer. As a reminder, the working tree contains the files and the staging area contains the changes which are marked to be included in the next commit. Depending on the specified parameters the git reset command performs the following:

1. If you specify the --soft parameter, the git reset command moves the HEAD pointer. Changes in the working tree will be left unchanged and all changes which were commited included in commits which are reseted are staged.
2. If you specify the --mixed parameter (the default), the git reset command moves the HEAD pointer and resets the staging area to the new HEAD. Any file change between the original commit and the one you reset to shows up as modifications (or untracked files) in your working tree. Use this option to remove commits but keep all the work you have done. You can do additional changes, stage changes and commit again. This way you can build up a different commit history.
3. If you specify the --hard parameter, the git reset command moves the HEAD pointer and resets the staging area and the working tree to the new HEAD. This effectively removes the changes you have done between the original commit and the one you reset to.

Via parameters you can define if the staging area and the working tree is updated. These parameters are listed in the following table.

|  |  |  |  |
| --- | --- | --- | --- |
| *Table 2. git reset options* | | | |
| Reset | Branch pointer | Working tree | Staging area |
| soft | Yes | No | No |
| mixed (default) | Yes | No | Yes |
| hard | Yes | Yes | Yes |

The git reset command does not remove untracked files. See [Remove untracked files with git clean](http://www.vogella.com/tutorials/Git/article.html#gitclean) command for this purpose.

### [33.2. Not moving the HEAD pointer with git reset](http://www.vogella.com/tutorials/Git/article.html#resetcommit_path)

If you specify a path via the git reset [path] command, Git does not move the HEAD pointer. It updates the staging area or also the working tree depending on your specified option.

## [34. Resetting changes with git reset](http://www.vogella.com/tutorials/Git/article.html#resetcommits)

### [34.1. Finding commits that are no longer visible on a branch](http://www.vogella.com/tutorials/Git/article.html#findingresettedcommits)

If you reset the branch pointer of a branch to a certain commit, the git log commands does not show the commits which exist after this branch pointer. For example assume you have two commits A→ B, where B is the commit after A. You if you reset your branch pointer to A, the git log command does not include B anymore.

Commits like B can still be found via the git reflog command. See [Recovering lost commits](http://www.vogella.com/tutorials/Git/article.html#gitreflog).

## [35. Deleting changes in the working tree and staging area for tracked files](http://www.vogella.com/tutorials/Git/article.html#cleanworkingdir)

The git reset --hard command makes the working tree exactly match HEAD.

# removes staged and working tree changes

# of committed files

git reset --hard

### [35.1. Using git reset to squash commits](http://www.vogella.com/tutorials/Git/article.html#resettosquash)

git reset, squash commits As a soft reset does not remove your change to your files and index, you can use the git reset --soft command to squash several commits into one commit.

As the staging area is not changed with a soft reset, you keep it in the desired state for your new commit. This means that all the file changes from the commits which were reseted are still part of the staging area.

# squashes the last two commits

git reset --soft HEAD~1 && git commit -m "new commit message"

The interactive rebase adds more flexibility to squashing commits and allows to use the existing commit messages. See [Editing history with the interactive rebase](http://www.vogella.com/tutorials/Git/article.html#interactive_rebase) for details.

## [36. Retrieving files from the history](http://www.vogella.com/tutorials/Git/article.html#retrievefiles)

### [36.1. View file in different revision](http://www.vogella.com/tutorials/Git/article.html#retrievefiles_show)

The git show command allows to see and retrieve files from branches, commits and tags. It allows seeing the status of these files in the selected branch, commit or tag without checking them out into your working tree.

By default, this command addresses a file from the root of the repository, not the current directory. If you want the current directory then you have to use the ./ specifier. For example to address the pom.xml file the current directory use: ./pom.xml

The following commands demonstrate that. You can also make a copy of the file.

# [reference] can be a branch, tag, HEAD or commit ID

# [file\_path] is the file name including path

git show [reference]:[file\_path]

# to make a copy to copiedfile.txt

git show [reference]:[file\_path] > copiedfile.txt

# assume you have two pom.xml files. One in the root of the Git

# repository and one in the current working directory

# address the pom.xml in the git root folder

git show HEAD:pom.xml

# address the pom in the current directory

git show HEAD:./pom.xml

### [36.2. Restore a deleted file in a Git repo](http://www.vogella.com/tutorials/Git/article.html#retrievefiles_fromcommit)

You can checkout a file from the commit. To find the commit which deleted the file you can use the git log or the git ref-list command as demonstrated by the following command.

# see history of file

git log -- <file\_path>

# checkout file based on predecessors the last commit which affect it

# this was the commit which delete the file

git checkout [commit] ^ -- <file\_path>

# alternatively use git rev-list

git rev-list -n 1 HEAD -- <file\_path>

# afterwards, the same checkout based on the predecessors

git checkout [commit] ^ -- <file\_path>

## [37. See which commit deleted a file](http://www.vogella.com/tutorials/Git/article.html#retrievefiles_finddeletedfile)

The git log command allows you to determine which commit deleted a file. You can use the -- option in git log to see the commit history for a file, even if you have deleted the file.

# see the changes of a file, works even

# if the file was deleted

git log -- [file\_path]

# limit the output of Git log to the

# last commit, i.e. the commit which delete the file

# -1 to see only the last commit

# use 2 to see the last 2 commits etc

git log -1 -- [file\_path]

# include stat parameter to see

# some statics, e.g., how many files were

# deleted

git log -1 --stat -- [file\_path]

## [38. Revert commits](http://www.vogella.com/tutorials/Git/article.html#revertcommit)

### [38.1. Reverting a commit](http://www.vogella.com/tutorials/Git/article.html#undochanges_revertcommit1)

git revert You can revert commits via the git revert command. This command reverts the changes of a commit.

Such commits are useful to document that a change was withdrawn.

### [38.2. Example: Reverting a commit](http://www.vogella.com/tutorials/Git/article.html#undochanges_revertcommit2)

The following command demonstrates the usage of the git revert command.

# revert a commit

git revert commit\_id

## [39. Resetting the working tree based on a commit](http://www.vogella.com/tutorials/Git/article.html#undochanges_checkoutcommits)

### [39.1. Checkout based on commits and working tree](http://www.vogella.com/tutorials/Git/article.html#undochanges_checkoutcommits1)

git checkout, based on commit ID

You can check out arbitrary revisions of your file system via the git checkout command followed by the commit ID. This command will reset your complete working tree to the status described by this commit.

The commit ID is shown if you enter the git log command.

The following command shows the log.

# displays the commit history of the repository

# which contains the commit ID, author, message etc.

git log

The following listing shows an example output of a Git log command.

commit 046474a52e0ba1f1435ad285eae0d8ef19d529bf

Author: Lars Vogel <Lars.Vogel@gmail.com>

Date: Wed Jun 5 12:13:04 2013 +0200

Bug 409373 - Updates version number of e4 tools

Repairs the build

commit 2645d7eef0e24195fc407137200fe7e1795ecf49

Author: Lars Vogel <Lars.Vogel@gmail.com>

Date: Wed Jun 5 12:00:53 2013 +0200

Bug 409373 - Updates version number of e4 CSS spy features

### [39.2. Example: Checkout a commit](http://www.vogella.com/tutorials/Git/article.html#undochanges_checkoutcommits2)

To checkout a specific commit you can use the following command.

# checkout the older revision via

git checkout [commit\_id]

# based on the example output this could be

git checkout 046474a52e0ba1f1435ad285eae0d8ef19d529bf

# or you can use the abbreviated version

git checkout 046474a5

### [40.1. Detached HEAD](http://www.vogella.com/tutorials/Git/article.html#detachedheadmode)

If you checkout a commit or a tag, you are in the so-called detached HEAD mode. If you commit changes in this mode, you have no branch which points to this commit. After you checkout a branch you cannot see the commit you did in detached head mode in the git log command.

To find such commits you can use the git reflog command.

### [40.2. git reflog](http://www.vogella.com/tutorials/Git/article.html#gitreflog_definition)

Reflog is a mechanism to record the movements of the HEAD and the branches references.

The reflog command gives a history of the complete changes of the HEAD reference.

git reflog

# <output>

# ... snip ...

1f1a73a HEAD@{2}: commit: More chaanges - typo in the commit message

45ca204 HEAD@{3}: commit: These are new changes

cf616d4 HEAD@{4}: commit (initial): Initial commit

The git reflog command also list commits which you have removed.

### [40.3. Example](http://www.vogella.com/tutorials/Git/article.html#gitreflog_example)

The following example shows how you can use git reflog to reset the current local branch to a commit which isn’t reachable from the current branch anymore.

# assume the ID for the second commit is

# 45ca2045be3aeda054c5418ec3c4ce63b5f269f7

# resets the head for your tree to the second commit

git reset --hard 45ca2045be3aeda054c5418ec3c4ce63b5f269f7

# see the log

git log

# output shows the history until the 45ca2045be commit

# see all the history including the deletion

git reflog

# <output>

cf616d4 HEAD@{1}: reset: moving to 45ca2045be3aeda054c5418ec3c4ce63b5f269f7

# ... snip ...

1f1a73a HEAD@{2}: commit: More chaanges - typo in the commit message

45ca204 HEAD@{3}: commit: These are new changes

cf616d4 HEAD@{4}: commit (initial): Initial commit

git reset --hard 1f1a73a

## [41. Remote and local tracking branches](http://www.vogella.com/tutorials/Git/article.html#gitremotebranch)

### [41.1. Remote tracking branches](http://www.vogella.com/tutorials/Git/article.html#gitremotebranch_overview)

Your local Git repository contains references to the state of the branches on the remote repositories to which it is connected. These local references are called remote-tracking branches.

You can see your remote-tracking branches with the following command.

# list all remote branches

git branch -r

To update remote-tracking branches without changing local branches you use the git fetchcommand. See [Updating your remote-tracking branches with git fetch](http://www.vogella.com/tutorials/Git/article.html#gitfetch) for more information.

It is safe to delete a remote branch in your local Git repository, this does not affect a remote repository. The next time you run the git fetch command, the remote branch is recreated. You can use the following command for that.

# delete remote branch from origin

git branch -d -r origin/[remote\_branch]

### [41.2. Delete a remote branch](http://www.vogella.com/tutorials/Git/article.html#gitremotebranch_deleteremote)

To delete the branch in a remote repository use the following command.

# delete branch in a remote repository

git push [remote] --delete [branch]

### [41.3. Tracking branches](http://www.vogella.com/tutorials/Git/article.html#git_trackingbranches)

Branches can track another branch. This is called to have an upstream branch and such branches can be referred to as tracking branches.

Tracking branches\_ allow you to use the git pull and git push command directly without specifying the branch and repository.

If you clone a Git repository, your local master branch is created as a tracking branch for the master branch of the origin repository (short:\_origin/master\_) by Git.

You create new tracking branches by specifying the remote branch during the creation of a branch. The following example demonstrates that.

# setup a tracking branch called newbrach

# which tracks origin/newbranch

git checkout -b newbranch origin/newbranch

Instead of using the git checkout command you can also use the git branch command.

# origin/master used as example, but can be replaced

# create branch based on remote branch

git branch [new\_branch] origin/master

# use --track,

# default when the start point is a remote-tracking branch

git branch --track [new\_branch] origin/master

The --no-track allows you to specify that you do not want to track a branch. You can explicitly add a tracking branch with the git branch -u command later.

# instruct Git to create a branch which does

# not track another branch

git branch --no-track [new\_branch\_notrack] origin/master

# update this branch to track the origin/master branch

git branch -u origin/master [new\_branch\_notrack]

To see the tracking branches for a remote repository (short: remote) you can use the following command.

# show all remote and tracking branches for origin

git remote show origin

An example output of this might look as follows.

\* remote origin

Fetch URL: ssh://test@git.eclipse.org/gitroot/e4/org.eclipse.e4.tools.git

Push URL: ssh://test@git.eclipse.org/gitroot/e4/org.eclipse.e4.tools.git

HEAD branch: master

Remote branches:

integration tracked

interm\_rc2 tracked

master tracked

smcela/HandlerAddonUpdates tracked

Local branches configured for 'git pull':

integration rebases onto remote integration

master rebases onto remote master

testing rebases onto remote master

Local refs configured for 'git push':

integration pushes to integration (up to date)

master pushes to master (up to date)

## [42. Updating your remote-tracking branches with git fetch](http://www.vogella.com/tutorials/Git/article.html#gitfetch)

### [42.1. Fetch](http://www.vogella.com/tutorials/Git/article.html#gitfetch_intro)

The git fetch command updates your remote-tracking branches, i.e., it updates the local copy of branches stored in a remote repository. The following command updates the remote-tracking branches from the repository called origin.

git fetch origin

The fetch command only updates the remote-tracking branches and none of the local branches. It also does not change the working tree of the Git repository. Therefore, you can run the git fetch command at any point in time.

After reviewing the changes in the remote tracking branchm you can merge the changes into your local branches or rebase your local branches onto the remote-tracking branch.

Alternatively you can also use the git cherry-pick commit\_id command to take over only selected commits.

See [Applying a single commit with cherry-pick](http://www.vogella.com/tutorials/Git/article.html#cherrypick_definition) for information about cherry-pick. See [Merging](http://www.vogella.com/tutorials/Git/article.html#gitmerge_definition)for the merge operation and [Rebasing branches](http://www.vogella.com/tutorials/Git/article.html#rebase_branches). for the rebase command.

### [42.2. Fetch from all remote repositories](http://www.vogella.com/tutorials/Git/article.html#gitfetch_updateremotes)

The git fetch command updates only the remote-tracking branches for one remote repository. In case you want to update the remote-tracking branches of all your remote repositories you can use the following command.

# simplification of the fetch command

# this runs git fetch for every remote repository

git remote update

# the same but remove all stale branches which

# are not in the remote anymore

git remote update --prune

### [42.3. Compare remote-tracking branch with local branch](http://www.vogella.com/tutorials/Git/article.html#gitfetch_compare)

The following code shows a few options how you can compare your branches.

# show the log entries between the last local commit and the

# remote branch

git log HEAD..origin/master

# show the diff for each patch

git log -p HEAD..origin/master

# show a single diff

git diff HEAD...origin/master

# instead of using HEAD you can also

# specify the branches directly

git diff master origin/master

The above commands show the changes introduced in HEAD compared to origin. If you want to see the changes in origin compared to HEAD, you can switch the arguments or use the -Rparameter.

### [42.4. Rebase your local branch onto the remote-tracking branch](http://www.vogella.com/tutorials/Git/article.html#gitfetch_rebase)

You can rebase your current local branch onto a remote-tracking branch. The following commands demonstrate that.

# assume you want to rebase master based on the latest fetch

# therefore check it out

git checkout master

# update your remote-tracking branch

git fetch

# rebase your master onto origin/master

git rebase origin/master

### [42.5. Fetch compared with pull](http://www.vogella.com/tutorials/Git/article.html#gitremotebranch_fetchpull)

The git pull command performs a git fetch and git merge (or git rebase based on your Git settings). The git fetch does not perform any operations on your local branches. You can always run the fetch command and review the incoming changes.

## [43. Merging](http://www.vogella.com/tutorials/Git/article.html#gitmerge_definition)

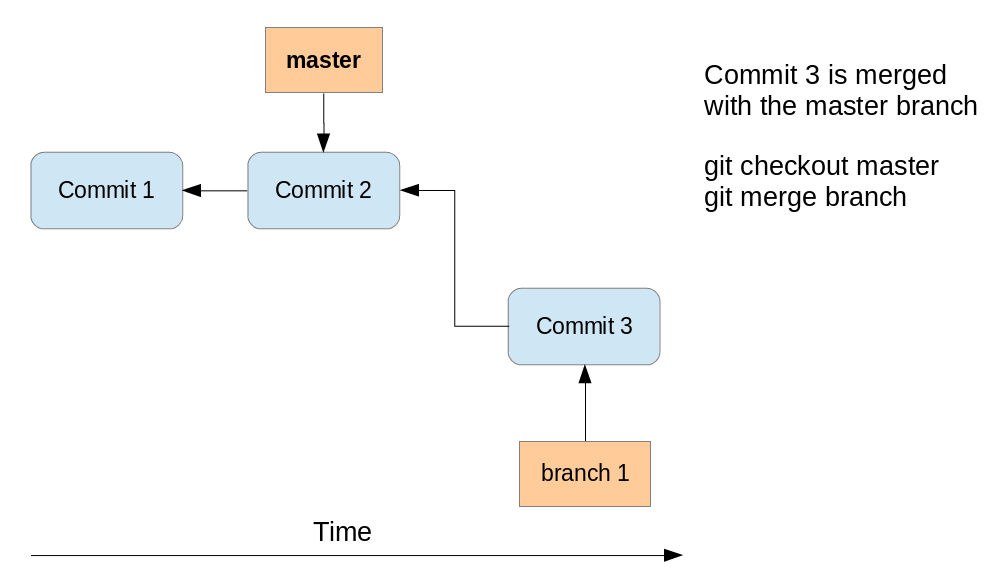
Git allows you to combine the changes which were created on two different branches. One way to achieve this is merging, which is described in this chapter. You can merge based on branches, tags or commits. Other ways are using rebase or cherry-pick.

This part explains how to merge changes between two different branches under the assumption that no merging conflicts happen. Solving conflicts is covered in [What is a conflict during a merge operation?](http://www.vogella.com/tutorials/Git/article.html#mergeconflict_definition).

### [43.1. Fast-forward merge](http://www.vogella.com/tutorials/Git/article.html#gitmerge_fastforward)

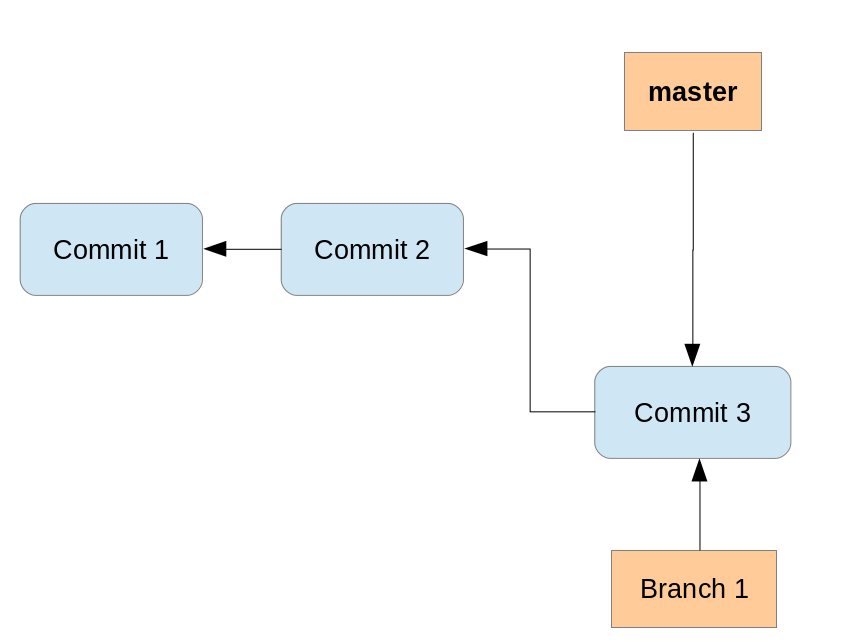
If the commits which are merged are direct successors of the HEAD pointer of the current branch, Git performs a so-called fast forward merge. This fast forward merge only moves the HEAD pointer of the current branch to the tip of the branch which is being merged.

This process is depicted in the following diagram. The first picture assumes that master is checked out and that you want to merge the changes of the branch labeled "branch 1" into your "master" branch. Each commit points to its predecessor (parent).



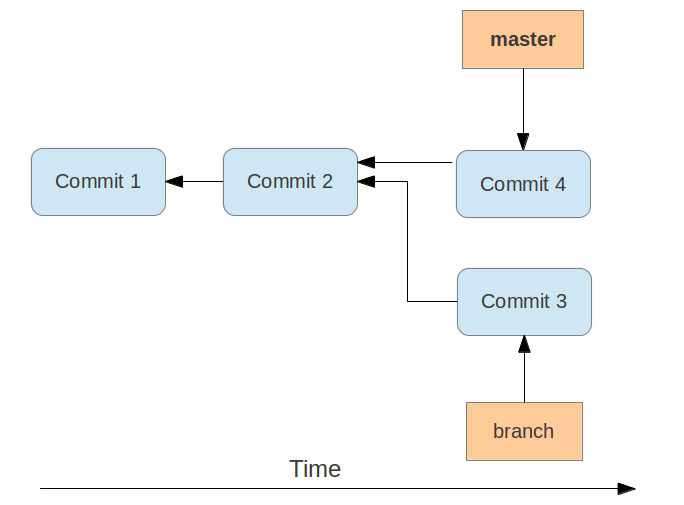
After the fast-forward merge the HEAD points to the master branch pointing to "Commit 3". The

"branch 1" branch points to the same commit.



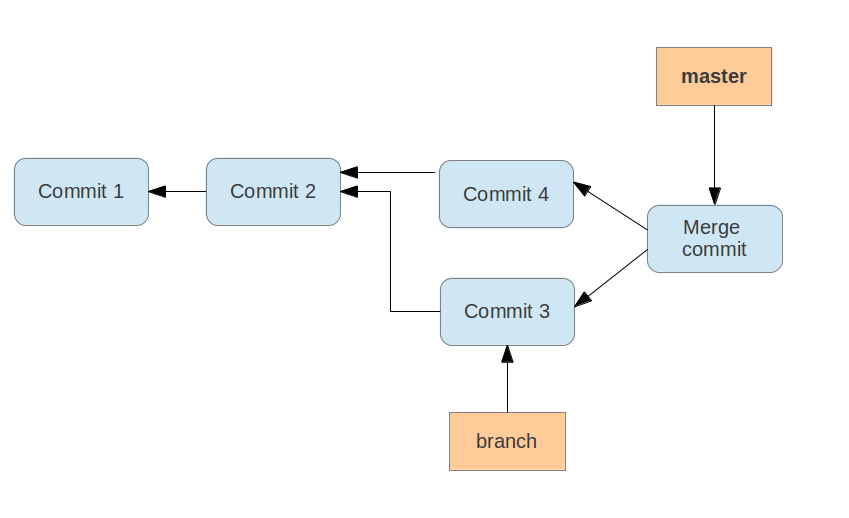
### [43.2. Merge commit](http://www.vogella.com/tutorials/Git/article.html#gitmerge_mergecommit)

If commits are merged which are not direct predecessors of the current branch, Git performs a so-called three-way-merge between the latest commits of the two branches, based on the most recent common predecessor of both.



As a result a so-called merge commit is created on the current branch. It combines the respective

changes from the two branches being merged. This commit points to both of its predecessors.



If multiple common predecessors exist, Git uses recursion to create a virtual common predecessor. For this Git creates a merged tree of the common ancestors and uses that as the reference for the 3-way merge. This is called the recursive merge strategy and is the default merge strategy.

### [43.3. Merge strategies - Octopus, Subtree, Ours](http://www.vogella.com/tutorials/Git/article.html#gitmerge_octopus)

If a fast-forward merge is not possible, Git uses a merge strategy. The default strategy called recursive merge strategy was described in [Merge commit](http://www.vogella.com/tutorials/Git/article.html#gitmerge_mergecommit).

The Git command line tooling also supports the octopus merge strategy for merges of multiple references. With this operation it can merge multiple branches at once.

The subtree option is useful when you want to merge in another project into a sub-directory of your current project. It is rarely used and you should prefer the usage of Git submodules. See [Git Submodules](http://www.vogella.com/tutorials/GitSubmodules/article.html) for more information.

The ours strategy merges a branch without looking at the changes introduced in this branch. This keeps the history of the merged branch but ignores the changes introduced in this branch.

You can use the ours merge strategy to document that you have integrated a branch and decided to ignore all changes from this branch.

### [43.4. Using the git merge command](http://www.vogella.com/tutorials/Git/article.html#gitmergecommand_branches)

The git merge command performs a merge. You can merge changes from one branch to the current active one via the following command.

# syntax: git merge <branch-name>

# merges into your currently checked out branch

git merge testing

### [43.5. Specifying merge strategies](http://www.vogella.com/tutorials/Git/article.html#gitmerge_mergeoptionsstrategies)

The -s parameter allows you to specify other merge strategies. This is demonstrated with the following command.

For example, you can specify the ours strategy in which the result of the merge is always that of the current branch head, effectively ignoring all changes from all other branches. This is demonstrated with the following command.

# merge branch "obsolete" ignoring all

# changes in the branch

git merge -s ours obsolete

The usage of the octopus merge strategy is triggered if you specify more than one reference to merge.

# merge the branch1 and the branch2 using

# changes in the branch

git merge branch1 branch2`

### [43.6. Specifying parameters for the default merge strategy](http://www.vogella.com/tutorials/Git/article.html#gitmerge_mergeoptions)

The recursive merge strategy (default) allows you to specify flags with the -X parameter. For example you can specify here the ours option. This option forces conflicting changes to be auto-resolved by favoring the local version. Changes from the other branch that do not conflict with our local version are reflected to the merge result. For a binary file, the entire contents are taken from the local version.

A similar option to ours is the theirs option. This option prefers the version from the branch which is merged.

Both options are demonstrated in the following example code.

# merge changes preferring our version

git merge -s recursive -X ours [branch\_to\_merge]

# merge changes preferring the version from

# the branch to merge

git merge -s recursive -X theirs [branch\_to\_merge]

Another useful option is the ignore-space-change parameter which ignores whitespace changes.

For more information about the merge strategies and options see [Git merge manpage](https://www.kernel.org/pub/software/scm/git/docs/git-merge.html).

### [43.7. Enforcing the creation of a merge commit](http://www.vogella.com/tutorials/Git/article.html#gitmerge_forcemergecommit)

If you prefer to have merge commits even for situations in which Git could perform a fast-forward merge you can use the git merge --no-ff command.

The --no-ff parameter can make sense if you want to record in the history at which time you merged from a maintenance branch to the master branch.

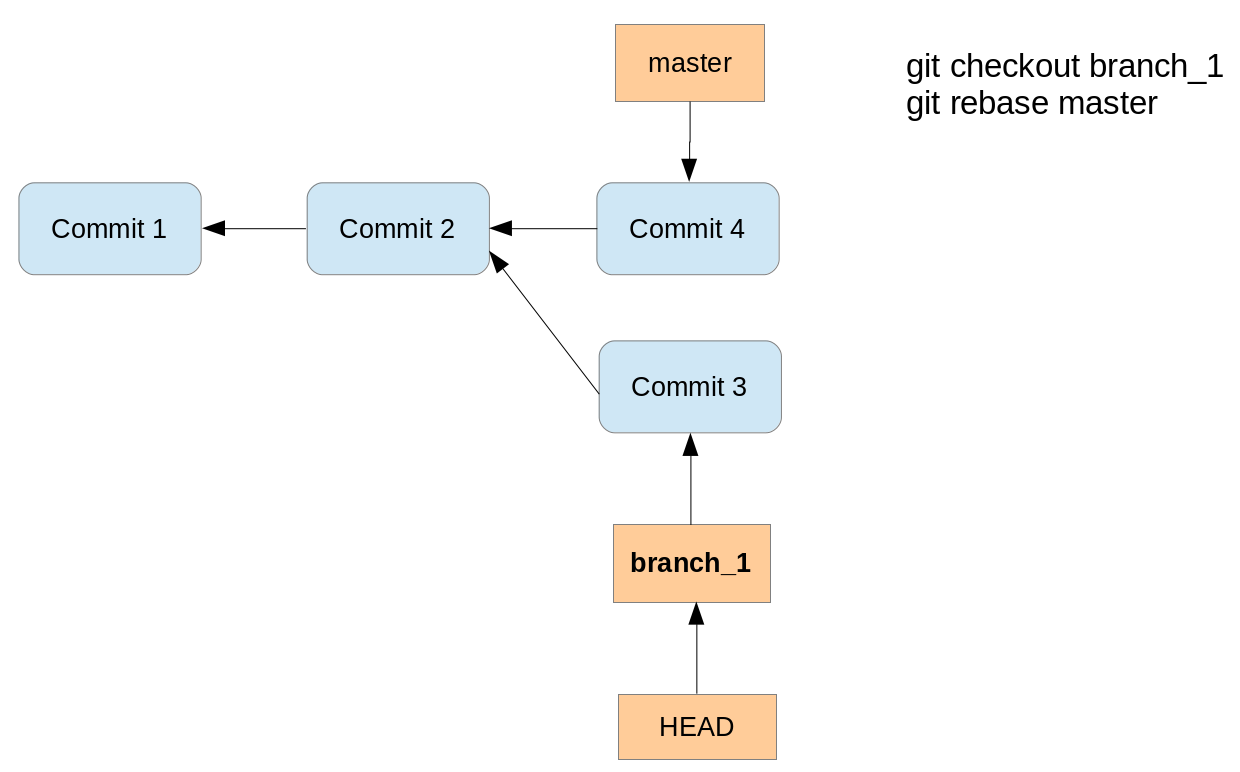
When pulling from a remote repository, prefer doing a rebase to a merge. This will help to keep the history easier to read. A merge commit can be helpful to document that functionality was developed in parallel.

## [44. Rebasing branches](http://www.vogella.com/tutorials/Git/article.html#rebase)

### [44.1. Rebasing branches](http://www.vogella.com/tutorials/Git/article.html#rebase_branches)

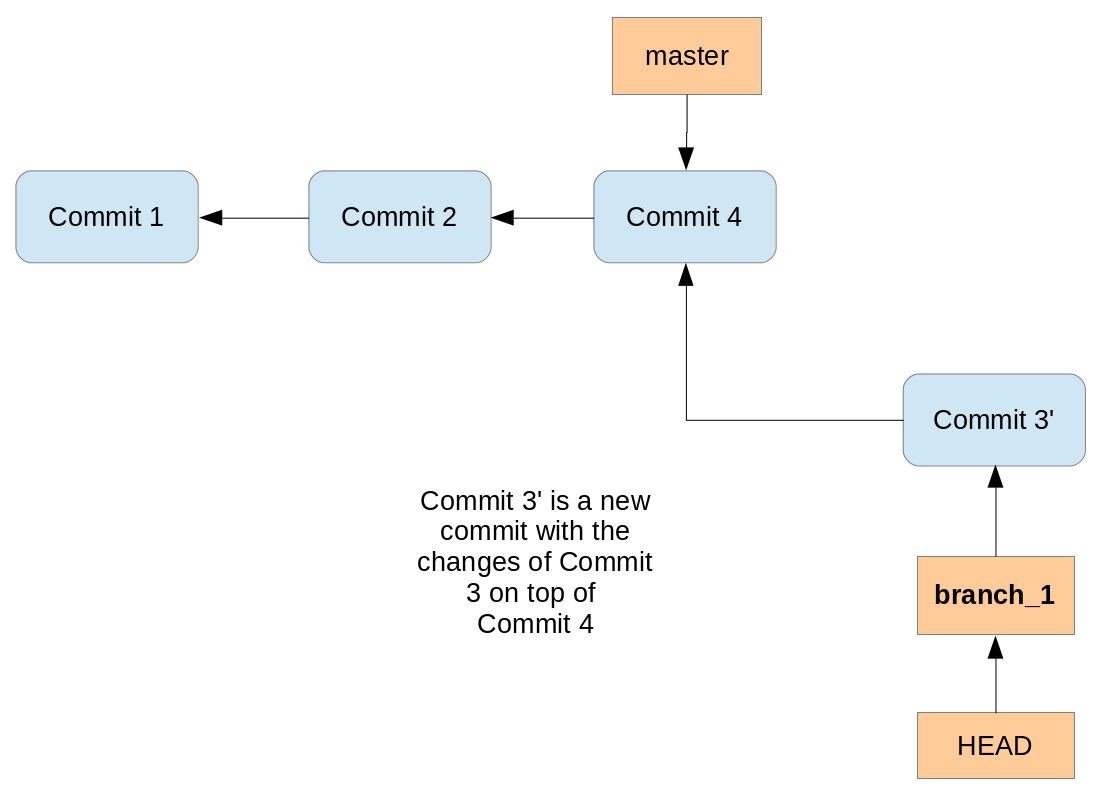
You can use Git to rebase one branch on another one. As described, the merge command combines the changes of two branches. If you rebase a branch called A onto another, the gitcommand takes the changes introduced by the commits of branch A and applies them based on the HEAD of the other branch. After this operation the changes in the other branch are also available in branch A.

The process is displayed in the following picture. We want to rebase the branch called branch\_1 onto master.



Running the rebase command creates a new commit with the changes of the branch on top of the

master branch.



Performing a rebase does not create a merge commit. The final result for the source code is the same as with merge but the commit history is cleaner; the history appears to be linear.

Rebase can be used to forward-port a feature branch in the local Git repository onto the changes of the master branch. This ensures that your feature is close to the tip of the upstream branch until it is finally published.

If you rewrite more than one commit by rebasing, you may have to solve conflicts per commit. In this case the merge operations might be simpler to be performed because you only have to solve merge conflicts once.

Also, if your policy requires that all commits result in correct software you have to test all the rewritten commits since they are "rewritten" by the rebase algorithm. Since merge/rebase/cherry-pick are purely text-based and do not understand the semantics of these texts they can end up with logically incorrect results. Hence, it might be more efficient to merge a long feature branch into upstream instead of rebasing it since you only have to review and test the merge commit.

### [44.2. Good practice for rebase](http://www.vogella.com/tutorials/Git/article.html#rebase_bestpractice)

You should avoid using the Git rebase operation for changes which have been published in other Git repositories. The Git rebase operation creates new commit objects, this may confuse other developers using the existing commit objects.

Assume that a user has a local feature branch and wants to push it to a branch on the remote repository. However, the branch has evolved and therefore pushing is not possible. Now it is good practice to fetch the latest state of the branch from the remote repository. Afterwards you rebase the local feature branch onto the remote tracking branch. This avoids an unnecessary merge commit. This rebasing of a local feature branch is also useful to incorporate the latest changes from remote into the local development, even if the user does not want to push right away.

### [44.3. Example for a rebase](http://www.vogella.com/tutorials/Git/article.html#rebase_example)

The following demonstrates how to perform a rebase operation.

# create new branch

git checkout -b rebasetest

# create a new file and put it under revision control

touch rebase1.txt

git add . && git commit -m "work in branch"

# do changes in master

git checkout master

# make some changes and commit into testing

echo "rebase this to rebasetest later" > rebasefile.txt

git add rebasefile.txt

git commit -m "create new file"

# rebase the rebasetest onto master

git checkout rebasetest

git rebase master

# now you can fast forward your branch onto master

git checkout master

git merge rebasetest

## [45. Editing history with the interactive rebase](http://www.vogella.com/tutorials/Git/article.html#interactive_rebase)

Git allows you to edit your commit history with a functionality called interactive rebase. For example, you can combine several commits into one commit, reorder or skip commits and edit the commit message.

This is useful as it allows the user to rewrite some commit history (cleaning it up) before pushing the changes to a remote repository.

Interactive rebase allows you to quickly edit a series of commits using the following actions:

| *Table 3. Interactive rebase actions* | |
| --- | --- |
| **Action** | **Description** |
| pick | includes the selected commit, moving pick entries enables reordering of commits |
| skip | removes a commit |
| edit | amends the commit |
| squash | combines the changes of the commit with the previous commit and combines their commit messages |
| fixup | squashes the changes of a commit into the previous commit discarding the squashed commit’s message |
| reword | similar to pick but allows modifying the commit message |

The setup for the rebase is called the rebase plan. Based on this plan, the actual interactive rebase can be executed.

### [45.1. Example: Interactive rebase](http://www.vogella.com/tutorials/Git/article.html#interactiverebase_example)

The following commands create several commits which will be used for the interactive rebase.

# create a new file

touch rebase.txt

# add it to git

git add . && git commit -m "add rebase.txt to staging area"

# do some silly changes and commit

echo "content" >> rebase.txt

git add . && git commit -m "add content"

echo " more content" >> rebase.txt

git add . && git commit -m "just testing"

echo " more content" >> rebase.txt

git add . && git commit -m "woops"

echo " more content" >> rebase.txt

git add . && git commit -m "yes"

echo " more content" >> rebase.txt

git add . && git commit -m "add more content"

echo " more content" >> rebase.txt

git add . && git commit -m "creation of important configuration file"

# check the git log message

git log

We want to combine the last seven commits. You can do this interactively via the following command.

git rebase -i HEAD~7

This command opens your editor of choice and lets you configure the rebase operation by defining which commits to pick, squash or fixup.

The following listing shows an example of the selection. We pick the last commit, squash 5 commits and fix the sixth commit. The listing uses the long format of the commands (for example fixup instead of the short form f ) for better readability.

pick 7c6472e rebase.txt added to index

fixup 4f73e68 added content

fixup bc9ec3f just testing

fixup 701cbb5 ups

fixup 910f38b yes

fixup 31d447d added more content

squash e08d5c3 creation of important configuration file

# Rebase 06e7464..e08d5c3 onto 06e7464

#

# Commands:

# p, pick = use commit

# r, reword = use commit, but edit the commit message

# e, edit = use commit, but stop for amending

# s, squash = use commit, but meld into previous commit

# f, fixup = like "squash", but discard this commit's log message

# x, exec = run command (the rest of the line) using shell

#

# These lines can be re-ordered; they are executed from top to bottom.

#

# If you remove a line here THAT COMMIT WILL BE LOST.

# However, if you remove everything, the rebase will be aborted.

## [46. Using the Git cherry-pick command](http://www.vogella.com/tutorials/Git/article.html#using-the-git-cherry-pick-command)

### [46.1. Applying a single commit with cherry-pick](http://www.vogella.com/tutorials/Git/article.html#cherrypick_definition)

The git cherry-pick command allows you to select the patch which was introduced with an individual commit and apply this patch on another branch. The patch is captured as a new commit on the other branch.

This way you can select individual changes from one branch and transfer them to another branch.

### [46.2. Example: Using cherry-pick](http://www.vogella.com/tutorials/Git/article.html#cherrypick_example)

In the following example you create a new branch and commit two changes.

# create new branch

git checkout -b picktest

# create some data and commit

touch pickfile.txt

git add pickfile.txt

git commit -m "adds new file"

# create second commit

echo "changes to file" > pickfile.txt

git commit -a -m "changes in file"

You can check the commit history, for example, with the git log --oneline command.

# see change commit history

git log --oneline

# results in the following output

2fc2e55 changes in file

ebb46b7 adds new file

[MORE COMMITS]

330b6a3 initial commit

The following command selects the first commit based on the commit ID and applies its changes to the master branch. This creates a new commit on the master branch.

git checkout master

git cherry-pick ebb46b7

The cherry-pick command can be used to change the order of commits. git cherry-pickalso accepts commit ranges for example in the following command.

git checkout master

# pick the last two commits

git cherry-pick picktest~1..picktest~2

See [Commit ranges with the double dot operator](http://www.vogella.com/tutorials/Git/article.html#commitreference_ranges_doubledot) for more information about commit ranges.

If things go wrong or you change your mind, you can always reset to the previous state using the following command.

git cherry-pick --abort

## [47. Solving merge conflicts](http://www.vogella.com/tutorials/Git/article.html#mergeconflict)

### [47.1. What is a conflict during a merge operation?](http://www.vogella.com/tutorials/Git/article.html#mergeconflict_definition)

Merge conflict A conflict during a merge operation occurs if two commits from different branches have modified the same content and Git cannot automatically determine how both changes should be combined when merging these branches.

This happens for example if the same line in a file has been replaced by two different commits.

If a conflict occurs, Git marks the conflict in the file and the programmer has to resolve the conflict manually.

After resolving it, he adds the file to the staging area and commits the change. These steps are required to finish the merge operation.

## [48. Keep a version of a file during a merge conflict](http://www.vogella.com/tutorials/Git/article.html#mergeconflict_theirs)

Merge conflict LARSSECONDARY"theirs parameter"LARSSECONDARY Merge conflict

LARSSECONDARY"ours parameter"LARSSECONDARY Sometimes if a conflict occurs the developer does not want to solve the conflict. He decides that he wants to keep the original version or the new version of the file.

For this, there is the --theirs and the --ours options on the git checkout command. The first option keeps the version of the file that you merged in, and the second option keeps the version before the merge operation was started.

git checkout --ours foo/bar.java

git add foo/bar.java

git checkout --theirs foo/bar.java

git add foo/bar.java

## [49. Exercise: Solving a conflict during a merge operation](http://www.vogella.com/tutorials/Git/article.html#mergeconflict)

### [49.1. Create a conflict](http://www.vogella.com/tutorials/Git/article.html#mergeconflict_examplecreate)

In the following example you create a conflict during a merge operation.

The following steps create a merge conflict. It assumes that repo1 and repo2 have the same origin repository defined.

# switch to the first directory

cd ~/repo01

# make changes

echo "Change in the first repository" > mergeconflict.txt

# stage and commit

git add . && git commit -a -m "Will create conflict 1"

# switch to the second directory

cd ~/repo02

# make changes

touch mergeconflict.txt

echo "Change in the second repository" > mergeconflict.txt

# stage and commit

git add . && git commit -a -m "Will create conflict 2"

# push to the master repository

git push

# switch to the first directory

cd ~/repo01

# now try to push from the first directory

# try to push --> assuming that the same remote repository is used,

# you get an error message

git push

As this push would not result in a non-fast-format merge, you receive an error message similar to the following listing.

! [rejected] master -> master (fetch first)

error: failed to push some refs to '../remote-repository.git/'

hint: Updates were rejected because the remote contains work that you do

hint: not have locally. This is usually caused by another repository pushing

hint: to the same ref. You may want to first integrate the remote changes

hint: (e.g., 'git pull ...') before pushing again.

hint: See the 'Note about fast-forwards' in 'git push --help' for details.

To solve this, you need to integrate the remote changes into your local repository. In the following listing the git fetch command gets the changes from the remote repository. The git merge command tries to integrate it into your local repository.

# get the changes via a fetch

git fetch origin

# now merge origin/master into the local master

# this creates a merge conflict in your

# local repository

git merge origin/master

This creates the conflict and a message similar to the following.

Auto-merging mergeconflict.txt

CONFLICT (add/add): Merge conflict in mergeconflict.txt

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

The resulting conflict is displayed in ? and solved in ?

|  |  |
| --- | --- |
|  | If you use the git pull command it performs the "fetch and merge" or the "fetch and rebase" command together in one step. Whether merge or rebase is used depends on your Git configuration for the branch. See ? for the global configuration. |

### [49.2. Review the conflict in the file](http://www.vogella.com/tutorials/Git/article.html#mergeconflict_examplesee)

Git marks the conflicts in the affected files. In the example from ? one file has a conflict and the file looks like the following listing.

<<<<<<< HEAD

Change in the first repository

=======

Change in the second repository

>>>>>>> b29196692f5ebfd10d8a9ca1911c8b08127c85f8

The text above the ======= signs is the conflicting change from your current branch and the text below is the conflicting change from the branch that you are merging in.

### [49.3. Solve a conflict in a file](http://www.vogella.com/tutorials/Git/article.html#mergeconflict_examplesolve)

In this example you resolve the conflict which was created in ? and apply the change to the Git repository.

To solve the merge conflict you edit the file manually. The following listing shows a possible result.

Change in the first and second repository

Afterwards add the affected file to the staging area and commit the result. This creates the merge commit. You can also push the integrated changes now to the remote repository.

# add the modified file

git add .

# creates the merge commit

git commit -m "Merge changes"

# push the changes to the remote repository

git push

Instead of using the -m option in the above example you can also use the git commitcommand without this option. In this case the command opens your default editor with the default commit message about the merged conflicts. It is good practice to use this message.

|  |  |
| --- | --- |
|  | Alternatively, you could use the git mergetool command. git mergetoolstarts a configurable merge tool that displays the changes in a split screen. Some operating systems may come with a suitable merge tool already installed or configured for Git. |

## [50. Solving rebase conflicts](http://www.vogella.com/tutorials/Git/article.html#rebaseconflict)

### [50.1. What is a conflict during a rebase operation?](http://www.vogella.com/tutorials/Git/article.html#rebaseconflict_definition)

Rebase conflict During a rebase operation, several commits are applied onto a certain commit. If you rebase a branch onto another branch, this commit is the last common ancestor of the two branches.

For each commit which is applied it is possible that a conflict occurs.

## [51. Handling a conflict during a rebase operation](http://www.vogella.com/tutorials/Git/article.html#rebaseconflict_solving)

If a conflict occurs during a rebase operation, the rebase operation stops and the developer needs to resolve the conflict. After he has solved the conflicts, the developer instructs Git to continue with the rebase operation.

A conflict during a rebase operation is solved similarly to the way a conflict during a merge operation is solved. The developer edits the conflicts and adds the files to the Git index. Afterwards he continues the rebase operation with the following command.

# rebase conflict is fixed, continue with the rebase operation

git rebase --continue

To see the files which have a rebase conflict use the following command.

# lists the files which have a conflict

git diff --name-only --diff-filter=U

You solve such a conflict the same way as you would solve a merge conflict.

You can also skip the commit which creates the conflict.

# skip commit which creates the conflict

git rebase --skip

## [52. Aborting a rebase operation](http://www.vogella.com/tutorials/Git/article.html#rebaseconflict_abort)

You can also abort a rebase operation with the following command.

# abort rebase and recreate the situation before the rebase

git rebase --abort

## [53. Picking theirs or ours for conflicting file](http://www.vogella.com/tutorials/Git/article.html#rebaseconflict_theirs)

If a file is in conflict, you can instruct Git to take the version from the new commit of the version of commit onto which the new changes are applied. This is sometimes easier than to solve all conflicts manually. For this you can use the git checkout with the --theirs or --ours flag. During the conflict --ours points to the file in the commit onto which the new commit is placed, i.g., using this skips the new changes for this file.

Therefore to ignore the changes in a commit for a file use the following command.

git checkout --ours foo/bar.java

git add foo/bar.java

To take the version of the new commit use the following command.

git checkout --theirs foo/bar.java

git add foo/bar.java

## [54. Define alias](http://www.vogella.com/tutorials/Git/article.html#alias)

### [54.1. Using an alias](http://www.vogella.com/tutorials/Git/article.html#alias_definition)

An alias in Git allows you to create a short form of one or several existing Git commands. For example, you can define an alias which is a short form of your own favorite commands or you can combine several commands with an alias.

### [54.2. Alias examples](http://www.vogella.com/tutorials/Git/article.html#alias_example)

The following defines an alias to see the staged changes with the new git staged command.

git config --global alias.staged 'diff --cached'

Or you can define an alias for a detailed git log command. The following command defines the git ll alias.

git config --global alias.ll 'log --graph --oneline --decorate --all'

You can also run external commands. In this case you start the alias definition with a !character. For example, the following defines the git ac command which combines git add . -A and git commit commands.

# define alias

git config --global alias.act '!git add . -A && git commit'

# to use it

git act -m "message"

## [55. Error search with git bisect](http://www.vogella.com/tutorials/Git/article.html#bisect)

### [55.1. Using git bisect](http://www.vogella.com/tutorials/Git/article.html#bisect_definition)

git bisect The git bisect command allows you to run a binary search through the commit history to identify the commit which introduced an issue. You specify a range of commits and a script that the bisect command uses to identify whether a commit is good or bad.

This script must return 0 if the condition is fulfilled and non-zero if the condition is not fulfilled.

### [55.2. git bisect example](http://www.vogella.com/tutorials/Git/article.html#bisect_cloning)

Create a new Git repository, create the text1.txt file and commit it to the repository. Do a few more changes, remove the file and again do a few more changes.

We use a simple shell script which checks the existence of a file. Ensure that this file is executable.

#!/bin/bash

FILE=$1

if [ -f $FILE ];

then

exit 0;

else

exit 1;

fi

Afterwards use the git bisect command to find the bad commit. First you use the git bisect start command to define a commit known to be bad (showing the problem) and a commit known to be good (not showing the problem).

# define that bisect should check

# the last 5 commits

git bisect start HEAD HEAD~5

Afterwards run the bisect command using the shell script.

# assumes that the check script

# is a directory above the current

git bisect run ../check.sh test1.txt

|  |  |
| --- | --- |
|  | The above commands serve as an example. The existence of a file can be easier verified with the git bisect command: git bisect run test -f test1.txt |

## [56. Rewriting commit history with git filter-branch](http://www.vogella.com/tutorials/Git/article.html#filterbranch)

### [56.1. Using the git filter branch command (filter-branch)](http://www.vogella.com/tutorials/Git/article.html#filterbranch_definition)

The git filter-branch command allows you to rewrite the Git commit history. This can be done for selected branches and you can apply custom filters on each revision. This creates different hashes for all modified commits. This implies that you get new IDs for all commits based on any rewritten commit.

The command allows you to filter for several values, e.g., the author, the message, etc. For details please see the [git-filter-branch manual page](http://www.kernel.org/pub/software/scm/git/docs/git-filter-branch.html)

|  |  |
| --- | --- |
|  | Using the filter-branch command is dangerous as it changes the Git repository. It changes the commit IDs and reacting on such a change requires explicit action from the developer, e.g., trying to rebase the stale local branch onto the corresponding rewritten remote-tracking branch. |

For example, you can use git filter-branch if you want to remove a file which contains a password from the Git history. Or you want to remove huge binary files from the history. To completely remove such files, you need to run the filter-branch command on all branches.

### [56.2. filter-branch examples](http://www.vogella.com/tutorials/Git/article.html#filterbranch_example)

The following command extracts a directory from a Git repository and retains all commits for this subfolder.

git filter-branch --prune-empty --subdirectory-filter FOLDER-NAME BRANCH-NAME

The following command replaces the email address of one author from all commits.

git filter-branch -f \

--env-filter 'if [ "$GIT\_AUTHOR\_NAME" = "Lars Vogel" ]; then \

GIT\_AUTHOR\_EMAIL="lars.vogel@gmail.com"; fi' HEAD)

## [57. Working with patch files](http://www.vogella.com/tutorials/Git/article.html#working-with-patch-files)

### [57.1. What is a patch file?](http://www.vogella.com/tutorials/Git/article.html#gitpatch)

A patch is a text file that contains changes to other text files in a standarized format. A patch created with the git format-patch command includes meta-information about the commit (committer, date, commit message, etc) and also contains the changes introduced in binary data in the commit.

This file can be sent to someone else and the receiver can use it to apply the changes to his local repository. The metadata is preserved.

Alternatively you could create a diff file with the git diff command, but this diff file does not contain the metadata information.

### [57.2. Create and apply patches](http://www.vogella.com/tutorials/Git/article.html#gitpatch_createandapply)

The following example creates a branch, changes several files and creates a commit recording these changes.

# create a new branch

git branch mybranch

# use this new branch

git checkout mybranch

# make some changes

touch test05

# change some content in an existing file

echo "new content for test01" >test01

# commit this to the branch

git add .

git commit -m "first commit in the branch"

The

next example creates a patch for these changes.

# creates a patch --> git format-patch master

git format-patch origin/master

# this creates the file:

# patch 0001-First-commit-in-the-branch.patch

To apply this patch to your master branch in a different clone of the repository, switch to it and use the git apply command.

# switch to the master branch

git checkout master

# apply the patch

git apply 0001-First-commit-in-the-branch.patch

Afterwards you can commit the changes introduced by the patches and delete the patch file.

# patch is applied to master

# change can be committed

git add .

git commit -m "apply patch"

# delete the patch file

rm 0001-First-commit-in-the-branch.patch

|  |  |
| --- | --- |
|  | Use the git am command to apply and commit the changes in a single step. To apply and commit all patch files in the directory use, for example, the git am \*.patch command. You specify the order in which the patches are applied by specifying them on the command line. |

### [57.3. Create a patch for a selected commit](http://www.vogella.com/tutorials/Git/article.html#gitpatch_createforcommit)

You can specify the commit ID and the number of patches which should be created. For example, to create a patch for selected commits based on the HEAD pointer you can use the following commands.

# create patch for the last commit based on HEAD

git format-patch -1 HEAD

# create a patch series for the last three commits

# based on head

git format-patch -3 HEAD

## [58. Git commit and other hooks](http://www.vogella.com/tutorials/Git/article.html#gitcommithooks)

### [58.1. Usage of Git hooks](http://www.vogella.com/tutorials/Git/article.html#gitcommithooks_intro)

Git provides commit hooks, e.g., programs which can be executed at a pre-defined point during the work with the repository. For example, you can ensure that the commit message has a certain format or trigger an action after a push to the server.

These programs are usually scripts and can be written in any language, e.g., as shell scripts or in Perl, Python etc. You can also implement a hook, for example, in C and use the resulting executables. Git calls the scripts based on a naming convention.

### [58.2. Client and server side commit hooks](http://www.vogella.com/tutorials/Git/article.html#gitcommithooks_clientserver)

Git provides hooks for the client and for the server side. On the server side you can use the pre-receive and post-receive script to check the input or to trigger actions after the commit. The usage of a server commit hook requires that you have access to the server. Hosting providers like GitHub or Bitbucket do not offer this access.

If you create a new Git repository, Git creates example scripts in the .git/hooks directory. The example scripts end with .sample. To activate them make them executable and remove the .sample from the filename.

The hooks are documented under the following URL: [Git hooks manual page](https://www.kernel.org/pub/software/scm/git/docs/githooks.html).

### [58.3. Restrictions](http://www.vogella.com/tutorials/Git/article.html#gitcommithooks_restrictions)

Not all Git server implementations support server side commit hooks. For example Gerrit (a Git server which also provides the ability to do code review) does not support hooks in this form. Also Github and Bitbucket do not support server hooks at the time of this writing.

Local hooks in the local repository can be removed by the developer.

## [59. Handling line endings on different platforms](http://www.vogella.com/tutorials/Git/article.html#line_feedendings)

### [59.1. Line endings of the different platforms](http://www.vogella.com/tutorials/Git/article.html#line_feedendings_platform)

Every time a developer presses return on the keyboard an invisible character called a line ending is inserted. Unfortunately, different operating systems handle line endings differently.

Linux and Mac use different line endings than Windows. Windows uses a carriage-return and a linefeed character (CRLF), while Linux and Mac only uses a linefeed character (LF). This becomes a problem if developers use different operating system to commit changes to a Git repository.

To avoid commits because of line ending differences in your Git repository you should configure all clients to write the same line ending to the Git repository.

### [59.2. Configuring line ending settings as developer](http://www.vogella.com/tutorials/Git/article.html#line_feedendings_convert)

On Windows systems you can tell Git to convert line endings during a checkout to CRLF and to convert them back to LF during commit. Use the following setting for this.

# configure Git on Windows to properly handle line endings

git config --global core.autocrlf true

On Linux and Mac you can tell Git to convert CRLF to LF with the following setting.

# configure Git on Linux and Mac to properly handle line endings

git config --global core.autocrlf input

### [59.3. Configuring line ending settings per repository](http://www.vogella.com/tutorials/Git/article.html#line_feedendings_gitattributes)

You can also configure the line ending handling per repository by adding a special .gitattributes file to the root folder of your Git repository. If this file is committed to the repository, it overrides the core.autocrlf setting of the individual developer.

In this file you can configure Git to auto detect the line endings.

|  |  |
| --- | --- |
|  | Not all graphical Git tools support the .gitattributes file, for example the Eclipse IDE does currently not support it. See [Eclipse Bug report](https://bugs.eclipse.org/bugs/show_bug.cgi?id=342372). |

## [60. Migrating from SVN](http://www.vogella.com/tutorials/Git/article.html#svn_migration)

To convert Subversion projects to Git you can use a RubyGem called svn2git. This tool relies on git svn internally and handles most of the trouble.

To install it (on Ubuntu) simply type:

sudo apt-get install git-svn ruby rubygems

sudo gem install svn2git

Let’s say you have a repository called

http://svn.example.com/repo

with the default layout (trunk, branches, tags) and already prepared a local git repository where you want to put everything. Then navigate to your git directory and use the following commands:

svn2git http://svn.example.com/repo --verbose

svn2git --rebase

The parameter --verbose adds detailed output to the commandline so you can see what is going on including potential errors. The second svn2git --rebase command aligns your new git repository with the svn import. You are now ready to push to the web and get forked! If your svn layout deviates from the standard or other problems occur, seek svn2git --help for documentation on additional parameters.

## [61. Frequently asked questions](http://www.vogella.com/tutorials/Git/article.html#gitfaq)

### [61.1. Can Git handle symlinks?](http://www.vogella.com/tutorials/Git/article.html#gitfaq_symlinks)

The usage of symlinks requires that the operating system used by the developers supports them.

Git as version control system can handle symlinks.

If the symlink points to a file, then Git stores the path information it is symlinking to, and the file type. This is similar to a symlink to a directory; Git does not store the contents under the symlinked directory.

===========================================================================