**GitHub**

Github is a global repository system which is used for version control.

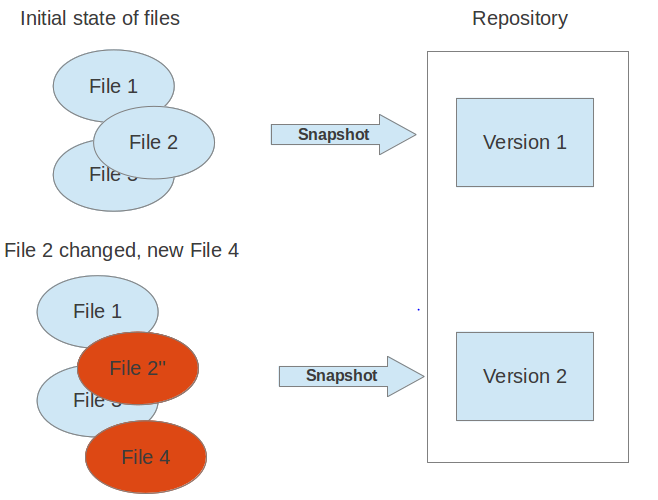
**Version Control System :**  is a software that helps software developers to work together and maintain a complete history of their work.

Listed below are the functions of a VCS:

* Allows developers to work simultaneously.
* Does not allow overwriting each other’s changes.
* Maintains a history of every version.

Following are the types of VCS:

* Centralized version control system (CVCS).
* Distributed/Decentralized version control system (DVCS).



**Centralized version control system (CVCS) :**

Centralized version control system (CVCS) uses a central server to store all files and enables team collaboration. But the major drawback of CVCS is its single point of failure, i.e., failure of the central server.

**Distributed/Decentralized version control system (DVCS) :**

DVCS clients not only check out the latest snapshot of the directory but they also fully mirror the repository. If the server goes down, then the repository from any client can be copied back to the server to restore it. Every checkout is a full backup of the repository. Git does not rely on the central server and that is why you can perform many operations when you are offline. You can commit changes, create branches, view logs, and perform other operations when you are offline. You require network connection only to publish your changes and take the latest changes.

### Local Repository:

### Every VCS tool provides a private workplace as a working copy. Developers make changes in their private workplace and after commit, these changes become a part of the repository. Git takes it one step further by providing them a private copy of the whole repository. Users can perform many operations with this repository such as add file, remove file, rename file, move file, commit changes, and many more.

### Working Directory and Staging Area or Index

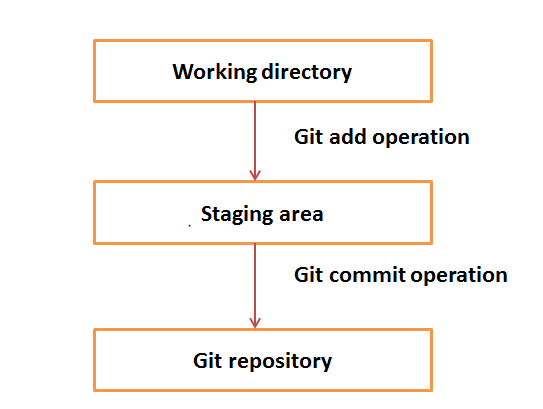
The working directory is the place where files are checked out. In other CVCS, developers generally make modifications and commit their changes directly to the repository. But Git uses a different strategy. Git doesn’t track each and every modified file. Whenever you do commit an operation, Git looks for the files present in the staging area. Only those files present in the staging area are considered for commit and not all the modified files.

Let us see the basic workflow of Git.

**Step 1** : You modify a file from the working directory.

**Step 2** : You add these files to the staging area.

**Step 3** : You perform commit operation that moves the files from the staging area. After push operation, it stores the changes permanently to the Git repository.



**DVCS Terminologies :**

**Blobs (Binary large object) :**

* Each version of a file is represented by blob.
* It doesn’t contain any metadata of the file.
* It is a binary file, and in Git database, it is named as SHA1 hash of that file

**Trees :**

Tree is an object, which represents a directory. It holds blobs as well as other sub-directories. A tree is a binary file that stores references to blobs.

**Commits :**

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.

You can consider a commit object as a node of the linked list. Every commit object has a pointer to the parent commit object.

**Branches :**

Branches are used to create another line of development. By default, Git has a master branch, which is same as trunk in Subversion. Usually, a branch is created to work on a new feature. Once the feature is completed, it is merged back with the master branch and we delete the branch.

Git supports *branching* which means that you can work on different versions of your collection of files. A

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

**Tags :**  
 Assign a meaning full name with specific version, Tages are immutable.

**Head :** HEAD is a pointer, which always points to the latest commit in the branch. Whenever you make a commit, HEAD is updated with the latest commit.

**Note :** <http://www.vogella.com/tutorials/Git/article.html#workingtree>

Gothrough : Chapter 11

[**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.

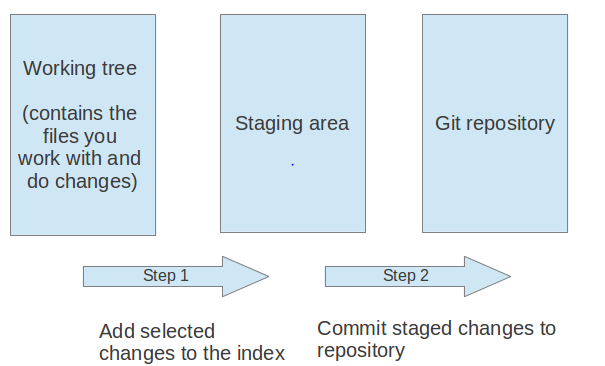
The user can change the files in the working tree by modifying existing files and by creating and removing files. A file in the working tree of a Git repository can have different states.

* untracked: the file is not tracked by the Git repository. This means that the file is 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

[**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 add command
* 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.

After adding the selected files to the staging area, you can commit these files to add them permanently to the Git repository.

$ git add sort.c (adding selected file to staging are)

$ git commit –m “Added sort operation” (committing the files to repository)

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

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

Git uses the /etc/gitconfig file for this system-wide configuration.

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

Git uses the  .gitconfig file for this user configuration.

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

Git uses the .git/config file for this repository configuration.

[**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](mailto:your.email@example.org)

### [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

### [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.

git config --global rebase.autoStash true

### [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 *EDITOR*environment 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

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

File conflict may occur during various operation . In this case user can resolve the issue. And also git allows to use 3rd party visual merge tools like tortoisemerge, p4merge, kdiff3 etc.

# setup kdiff3 as default merge tool (Linux)

git config --global merge.tool kdiff3

# to install it under Ubuntu use

sudo apt-get install kdiff3

**Note :**  [**http://www.vogella.com/tutorials/Git/article.html#workingtree**](http://www.vogella.com/tutorials/Git/article.html#workingtree)

**Gothrough : chapter 16**

**[Exercise(18): Performing a Exercise 17 and 18](http://www.vogella.com/tutorials/Git/article.html" \l "firstgit)**

**[local Git workflow](http://www.vogella.com/tutorials/Git/article.html" \l "firstgit)**

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.

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

Create directory with structure : ~/repo01/datafiles

[**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.

cd ~/repo01

# initialize the Git repository

git init

**[Create new content](http://www.vogella.com/tutorials/Git/article.html" \l "firstgit_content)**

Use the following commands to create several new files.

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

[**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 (notice output)

[**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 git status (notice output)

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

It Make changes to staged files as below and run git status

# append a string to the test03 file

echo "foo2" >> test03

# see the result

git status

Add files to staged state and notice status.

[**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.

# commit your file to the local repository

git commit -m "Initial commit"

-m => --message (allows you to specify the commint message)

[**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

[**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

|  |  |
| --- | --- |
|  |  |

### [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.

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"

### Merging vs. Rebasing :

|  |  |
| --- | --- |
| **Merge** | **Rebase** |
| Merging is nice because it’s a *non-destructive* operation. The existing branches are not changed in any way.  Feature wil have a extrenic merge commit every time need to have upstream changes.So this can be pollute history of feature branch.  git checkout feature  git merge master | This moves the entire feature branch to begin on the tip of the master branch, effectively incorporating all of the new commits in master  Rebasing *re-writes*the project history by creating brand new commits for each commit in the original branch.  git checkout feature  git merge master |

**Chapeter 19 & 20 : office**

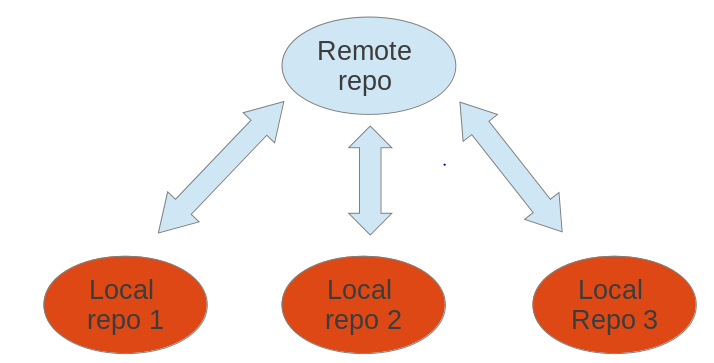
## http://www.vogella.com/tutorials/Git/article.html

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

### [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.



### [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.

# create a bare repository

git init --bare

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

### [Cloning a repository](http://www.vogella.com/tutorials/Git/article.html#remotes_cloneoperation) This is used to target an existing repository and create an copy of the target repository.

### If you clone a repository, Git implicitly creates a *remote* named *origin*.

git clone git@github.com:whatever folder-name

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

### By default remote name will be the orgin.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

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

### [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

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

# show the details of the remote repo called origin

git remote show origin

# show the existing defined remotes

git remote

# show details about the remotes

git remote -v

### [Push changes to another repository](http://www.vogella.com/tutorials/Git/article.html" \l "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.

### [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.

**20-25 in Home**

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

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

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.

### [Removing untracked files](http://www.vogella.com/tutorials/Git/article.html#gitclean_command) :

Used to remove untracked files in working tree.

Git clean –dry-run : it shows what happens if we clean a git (shows how many files will be delete)

Git clean –n : it shows what happens if we clean a git (shows how many files will be delete)

Git clean –f : used to delete untracked file.

Git clean –fdx : used to delete untracked directory.

-x : remove the hidden file.

-d : delete new directory.

### [Remove staged changes from the staging area](http://www.vogella.com/tutorials/Git/article.html#undochanges_reset) :

This is exactly opposite to the git add , this will remove the files from staging area to avoid changes to be included in next commit.

Git reset [filename] : remove file from stage. [this used to revert new file from staging area]

Git reset Head [filename] : this will used to revert changed file from staging area based on last commit.

Git reset –hard : command makes the working tree exactly match HEAD.

**Start from 40**

**Using Branches :** Branches are named pointer, we can work with independentlvy with different brancges.

**List of branches :**

* Git branch :list the local repositry branches. The currently active branch is marked with \*.
* Git branch –a : list of branches including remote branches.
* Git branch –r : list of remote repositry branches..
* Git branch –v : Provide more information about branches.

**Create new branch :**

Git branch [branch\_name] : used to create new branch.

Ex : git branch testing.

This command allows to specify the commit (c ommit 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.

**Checkout branches :**

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.

# switch to your new branch

git checkout testing

# do some changes to files

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 not modified

cat test01

**Rename Branch :**

Git branch –m [old\_name][new\_name]

**Delete Branch :**

Git branch –d [branch\_name] **:** this will delete branch.

Git branch –D [branch\_name] **:** Force delet.

**21.7 doubt**

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

Types of tags :

* **Lightweight tag :** Its pointer to commit without any additional information.

Git tag 1.7.1: create tag

Git show 1.7.1 : show the tag

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

* **annotated tag :** this conttain additional information about tags

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.

Git tag 1.7.1 –m ‘Release 1.7.1’: create tag

Git show 1.7.1 : show the tag

tag git checkout <tag\_name> : checkout tags

git push origin <tagname> :push tag or branch called tagname

git push origin tag<tagname> : to explicitly push a tag and not a branch

git push :push all tags.

**Gothrough 26(Reg logs)**