**Installation (for Windows)**

Git version 2.26.2 latest -20th April 2020

Git installation installs GUI and command line interface

The command line interface can use Linux commands.

Installation from <https://git-scm.com>

**Documentation**

Documentation at <https://git-scm.com/doc>

**Reference for this document**

Useful book on Git <https://git-scm.com/book/en/v2>

**Git benefits:**

Allows to create many branches without copying the entire source making it extremely fast to work in collaboration

It is fast as it is doesn’t use centralized server as operations are performed locally

It is distributed and every clone has complete information with the history (Perforce, CVS, SVN were centralized systems) so we don’t risk losing all the data. Also, many workflows can be created by collaborating with the non-central users.

Git stores snapshots and not differences like other VCS

It has staging area (index) not present in other versioning systems along with a working directory or working tree (after checkout) and the compressed repository to which commits are done. The three states of a file are committed, modified or staged.

**Configuration files in increasing priority order**

1. Location of gitconfig file for storing username and email for Git activities throughout the system for all users

C:\Program Files\Git\etc\gitconfig

1. Location of .gitconfig file for storing username and email for the user logged in

C:\Users\$USER\.gitconfig

1. Location of git config for the particular repo

.git\config

**Gitignore file**

.gitignore file keeps a record of those files which should not be tracked by Git in the working directory. Useful for binaries, libraries, etc. generated during builds which should not be committed to the repo. Please note that it is possible to have multiple .gitignore files for a project

A list of .gitignore files for various requirements is listed below at  
<https://github.com/github/gitignore>

**Some definitions**

*Author*: one who makes a patch

*Committer*: one who applies a patch

*Repository:* The files along with their history

*Remote repository*: version of your local repo which is held somewhere else over the internet (or on your PC itself)

*Local repository*: The folder with .git name is the local repo. This is different from working tree.

*Working directory:* Working tree. The actual folder which contains the files that you edit.

*Staging area*: A virtual area where the changes to be committed are held.

Tags: Tags are used to mark milestones in project development and are usually used for versioning of software. They are of two types: annotated and lightweight tags. The former one is recommended for tagging.

*Commit checksum:* Unique commit identifier

*Commit object, blobs, and tree:* Each commit is stored as a commit object containing information about the committer, author etc. And each file snapshot in the commit is kept as blobs. A tree contains the information on which file belongs to which blob. And the commit object holds pointer to this tree and also pointer to previous commit objects

*Branches*: A repository has many branches. E.g. ‘master’. A branch is a pointer to one of these commit objects. The branch pointer points to the latest commit object in that branch.

*HEAD*: pointer to the local branch and the commit object in that branch you are currently on. If the commit object is NOT the topmost object in the list of commit objects then it is termed as detached HEAD state. It happens when one checkouts a commit (or a tag) instead of the branch

*Topic branches:* Branches created temporarily for specific tasks which get deleted later after the small task is closed

*Remote references:* References to state of remote branches, remote tags and so on in the local repo. It is updated when the local repo is connected with the remote repo. They are of the form <remote repo>/<remote branch name>

*Remote-tracking branch*: they are of the form of origin/master and track the remote branches on a local repo.

*Tracking branches*: are branches that track a remote branch via a remote-tracking branch. The remote branch being tracked is known as the *upstream branch.* Short hand for upstream branch is @{u} or @{upstream} i.e. same as <remote repo>/<branch> e.g. ‘origin/master’

*Rebasing:* this technique is used instead of merging to apply new changes sometimes as it helps in generating a cleaner history. The end result of both commands is the same. In rebasing all the commits are applied again in the same order. However, in merging there is one final commit.

*Pull request*: Request to add changes/patches (in your repo) which is remote to another user and asking that user to pull your change to their repo. The command is ‘git request-pull’ and not ‘git pull’.

**Information**

Anything that is committed to the repo can always be recovered. Even if it was a commit that amended later or deleted later. Anything lost that was not committed will be lost forever.

If a file is modified, staged and modified again, the commit will be done only for the changes which have been staged and not the latest ones in the working directory

|  |  |
| --- | --- |
| **Configuration** | **Description** |
| git config –-system user.name “Ritesh Udhani”  git config –-global user.name “Ritesh Udhani”  git config –-local user.name “Ritesh Udhani” | Sets username used for all commits.  Updates the file #1, #2 and #3 above. |
| git config –-system user.email “\*\*\*\*@gmail.com”  git config –-global user.email “\*\*\*\*@gmail.com”  git config –-local user.email “\*\*\*\*@gmail.com” | To set email address for all commits.  Updates the file #1, #2 and #3 above. |
| git config --list --show-origin | To see where the configurations are stored. Listed in the order system, global and local with increasing priority |
| git config --global core.editor "'C:/Program Files/Notepad++/notepad++.exe' -multiInst -notabbar -nosession -noPlugin"  git config --global core.editor emacs | To set up your favorite text editor for git files |
| git help <command>  git <command> --help | To get help on a command |
| **Git basic commands** | **Description** |
| **Getting a git repo** |  |
| git init <project name>  git clone <project path> | To create a local repo afresh or from a remote repository. The .git folder is the repository and the files are a part of working directory (working tree). |
| **Recording changes in a git repo** |  |
| git add <filename> | It means “add precisely this content to the next commit”. This command is used to add a file to start tracking, staging and resolve merge conflicts |
| git restore –-staged <filename>  git reset HEAD <filename> | To unstage a file in the staging area (index) |
| git restore <filename>  git checkout -- <filename> | To discard modifications done in a file |
| git status | To check the status of each file whether it is modified, staged or non-modified. The output has two regions, staged area and non-staged area |
| git status -s | For a simplified output |
| git diff | To know the exact changes made but not staged. Compares staging area with working tree changes |
| git diff –-staged  git diff --cached | Also, to know the exact changes one is about to commit from the staging area of each file |
| git difftool | To use a different tool for comparison like vimdiff, emerge etc |
| git commit –m “message” | Commit changes to the local repo |
| git commit | The commit message is to be entered through the default text editor for git |
| git commit –a –m “message” | To commit directly from working tree to the repository by skipping the staging area. Use carefully as it can be risky |
| git rm <filename> | To remove and existing file from the repository. It also deletes file from the working tree and get the file deletion to the staging area for commiting. |
| git mv <filename1> <filename2> | To rename a file in git |
| **Viewing commit history** |  |
| git log | To see the history of commits in reverse chronological order. This lists commit id, author name, email, date of commit and commit messages |
| Git log –-pretty=oneline | Shows commit information (not file information) in one line. Shows a better readable formatting when many commits are present. |
| Git log –-pretty=short | Shows commit information (not file information) in short. Same as git log, but no commit date |
| Git log –-pretty=full | Shows committer and author information both in commit information (not file details) |
| Git log –-pretty=fuller | Committer and author information and the corresponding dates in commit information (not file details) |
| Git log –-pretty=format:”%h %ar” | To get the commit id, author, committer information, date, commit message in a specific format. Shows commit information (not file information) |
| Git log --relative-date | The time relative to now when the commit was made instead of the absolute date (shows commit information and not file information) |
| Git log --stat | In addition to commit information it displays file information like filename, no of insertions and deletions |
| Git log --shortstat | In addition to commit information it displays file information with stats summary line for file changes |
| Git log –-name-only | In addition to commit information it displays file information with only file name in stat |
| Git log –-name-status | In addition to commit information it displays file information with filename and the action done on that file, like additions, modification or deletion. |
| git log --graph | To view branches and merges information |
| git log –p -2 | Filter out commits. In addition to commit information and file information it displays modification information to show the patch details of the last two commits |
| git log –p -1 <commitid> | Filter out commits. In addition to commit information and file information it displays modification information to show patch details of a specific patch with commit id |
| git log –-author=”<part of author name>” | Filter out commits. To see commits by a specific author |
| git log –-committer=”<part of committer name>” | Filter out commits. To see commits by a specific comitter |
| Git log –since=”2 weeks” (“2 years, 2 minutes, 2 hours or specify a specific date "2008-01-15" etc.) | Filter out commits. By time boundation. |
| Git log –until=”2 weeks” (“2 years, 2 minutes, 2 hours or specify a specific date "2008-01-15" etc.) | Filter out commits. By time boundation. |
| Git log –grep “message” | Filter out commits. Greps message in commit messages and filters the commits |
| Git log <filename/directory name> | Filter out commits. Show only those commits which are related to a specific file or directory |
| Git log --no-merges | Filter out commits. Show only those commits which are not merge commits |
| **Undoing Things** |  |
| Git commit --amend | To amend previous commit by additionally committing staging area contents. It can also help to modify the commit message. The previous commit is deleted and replaced by a new commit. |
| **Working with remotes** |  |
| Git remote | To see the remote servers configured. If a repo is cloned then we should see atleast one remote repo. The default name of the repo server is ‘origin’ from which you cloned. |
| Git remote -v | To see the url of the remote servers |
| git remote add <shortname> <url> | To add a remote server. Shortname is usually ‘origin’. But it can be changed too. Shortname is only the localrepo’s name for the remote repo. The actual remote repo is a complete url. |
| Git remote rename <currentname> <newname> | To rename a remote repository in the local repo records. The server of remote repo has no changes. |
| Git clone –o <name for remote repo> | While cloning if we want a specific name of the remote repo, other than the default ‘origin’, it can be done like this |
| git remote remove <remote repo name>  git remote rm <remote repo name> | To remove a remote repo. E.g. remote repot name is ‘origin’. |
| git push <remote repo> --delete <remote branch name> | To delete a remote branch from a remote repo |
| Git fetch <remote repo name> | Merges data from the remote repository to the local repository. No change is made to the working directory. They have to be merged manually. Only the remote-tracking branch is updated in the local repository and not the tracking branch. |
| Git fetch -all | To fetch from all the remote repos instead of just one |
| Git pull <remote repo name> | It does the job of fetch and additionally tries to merge the remote repo contents to the working directory. It is recommended to do this only when all the changes are committed to the local branch. The remote-tracking and tracking branches, both are updated. |
| git push <remote> <branch> | To push your changes upstream to the remote repository branch. E.g. of remote is ‘origin’ and branch is ‘master’. It will work only if your local repo is up to date with the remote else the push request is rejected. Use fetch/pull to keep local repo updated. If the remote branch already doesn’t exist, it will be created. |
| Git push <remote> refs/heads/<local branch name>:refs/heads/<remote branch name>  Git push <remote> <local branch name>:<remote branch name> | It means push my local branch to remote repo under a different branch name |
| Git remote show <remote repo name> | To show the relationship between local and remote repositories, i.e., if one pushes the changes from a local branch, the command helps identify the remote repo and the corresponding branch the change will go to. Similarly, vice-versa for git pull. Example ‘remote repo name’ is ‘origin’ |
| **Tagging** |  |
| Git tag | Lists all the tags for a repository. |
| Git tag | grep “pattern\*” | List only those tags which have a pattern |
| Git tag –a <tagname> -m “<commit message>” | Creates annotated tag <tagname> (e.g. v1.0) with commit message and it also records taggers information |
| Git tag <tagname>-lw | Creates a tag of tagname-lw. This is a lightweight tag. It is better to use annotated tags |
| git tag -a <tagname> <commit checksum> | Create a tagname for a specific commit checksum. |
| Git show <tagname> | To show detailed information about a particular tag e.g. the tagger, time of tag etc. |
| git push <remote repo> <tagname> | Since tags are not pushed by default. They need to be pushed explicitly. Remote repo example is ‘origin’. This method pushes only one tag at a time. |
| git push <remote repo> --tags | Pushes all the tags (annotated and lightweight) to the remote server |
| git tag -d <tagname> | Delete a tag on a local repo |
| Git push <remote repo> -d <tagname> | Delete a tag on remote repo |
| git checkout <tagname> | If you want to view the source code at a particular tag. Not a good approach if you want to modify it also as the HEAD is in detached state |
| git checkout -b <new branchname> <tagname> | Create a new branch if you want to edit the files at a particular tagname |

|  |  |
| --- | --- |
| **Commands for branching** | **Description** |
| Git branch <new branch name> | To create a new branch. The branch is created pointing to the commit object the current branch was pointing to when the new branch was created. The current branch, however, is not switched |
| Git branch | Shows all the branches in a repo and also shows ur current branch marked with \* |
| Git branch –d <branch name> | To delete a branch. Works only when all changes are merged to another branch else it will fail |
| Git branch –D <branch name> | To forcefully delete a branch even if its contents aren’t merged yet. |
| Git branch –merged /--no-merged | Goto the branch which you want to retain e.g. master. If you want to find out the branches which are merged to it use –merged option and use –no-merged option for the ones which are not yet merged. This is useful to know for branch management, i.e. to delete them if their purpose is complete |
| Git branch –merged/--no-merged <ref branch name> | To see merged/no-merged branches for the branch ref branch name |
| Git branch -vv | To find out the local branches that are tracking branches, their status if they are up to date with remote-tracking branches or ahead or behind but not with the actual remote branches. |
| Git log --decorate | To see the current branch, i.e. to which branch the HEAD is pointing. Also, the current position in the branch is also shown (Not so useful as by default also HEAD is shown in latest git versions) |
| Git log | Log for the current branch |
| Git log <branch name> | To get commit history of the particular branch |
| Git log -all | To see the logs for all the branches |
| Git log –graph | To see graph for all branches |
| Git checkout < branch name> | To switch from an existing branch to the new branch. Moves to the top commit of the branch. Also modifies the working directory accordingly. Always commit your changes in the current branch before you invoke git checkout of a new branch, even if you are in the middle of something. But if you don’t want an incomplete commit, you can always amed the last commit when you come back to this branch after working on another. Branch switching won’t work if you have uncommitted changes |
| Git checkout –b <new branch name> | To create a new branch and switch to it |
| Git checkout –b <new local branch name> <remote repo>/<existing remote branch>  Is the same as  git checkout --track <remote repo>/<existing remote branch>  same as  git checkout <existing remote branch> | An existing branch can be local or remote branch reference. |
| Git checkout <commit id> | To checkout a specific commit id. Leaves the HEAD in detached state. So, don’t make changes here. If changes are necessary, create a branch instead. |
| Git merge <branch to be merged to another> | Merges the content of the branch to be merged to the current branch. If “fast-forward” phrase is observed during merge then it implies the two branches weren’t divergent and one was part of another’s history. For merge conflict file identification, run ‘git status’. The files with conflicts will contain symbols like “<<<” “>>>” and “===” delete these and resolve the conflict by editing this section and then do ‘git add’ to stage the changes for commit. |
| Git mergetool | This can be used for a visual merging tool when a merge conflict is detected. |
| git checkout experiment  git rebase master  git checkout master  git merge experiment | Steps for rebasing a new development on branch ‘experiment’ onto master branch which has diverged. |
| Git diff --check | Check for white spaces while submitting a commit |

Guideline:

User master branch for stable code. Always work on a separate branch for a new development or fixing a bug.

After integration to master branch, ensure to verify all functionalities. Better to have a test case ready.

User Topic branches for testing anything in particular.

Make tags for a functionality achieved

Use rebase for a cleaner history

Rebase commits only for your local repo. Don’t do it for outside repo.

Avoid creating topic branch of a topic branch and then do a rebase. It is doable but adds unnecessary level of complexity.

Centralized workflow to be used.

Before push do a fetch and merge

More than one person contribution needs verification

Follow Coding guidelines

No whitespaces in documents

Commit each feature separately as a logical set not as one massive code in a day. Use staging area to split code done together split into different logical parts

Put a useful message with each commit which is self-explanatory in under 50 words. Further details explanation if required can be added after a blank line

Benefits:

The same functionality if it has to be redone can be checked by referring to a commit id. Allows faster work

Easy to track work

Bugs if introduced can be easily tracked quickly by verification.

Github Flavored markdown can be used in Issue and Pull request comments,

Use Tasks list in pull requests.

Quoting by using > to quote what you are responding to.

Use code snippets as ````

Emojis can be used with :

Use commandline to leverage raw power of git. For basic usage GUI will be fine. Also, as they are same commands over any platform mac, linux, windows.