# Git Version Control Explained!

Version control systems help programmers keep track of the code they write, and also help them code in collaboration with others. They are very helpful in organising code based on different purposes, versions, in merging different pieces of code, and have many other interesting uses. Git is a new-generation distributed version control system written by Linux Torvalds. This article is a starter guide to Git, and walks through its usage patterns, essential features, internals and how to collaborate with multiple developers using its  collaboration tools such as GitHub.

Most people are familiar with centralised version control systems such as Subversion, in which every action requires connectivity to the (Subversion) server. If the central server fails, all collaborators that depend on the server will be affected, and any data access using the central repository will stop. Distributed version control systems (DVCS) do not rely on a central repository server; rather, each user has a full copy of the repository. Whenever the main server fails, you can recover the server from the individual repository copies.

Git combines the best of the earlier DVCS used to maintain the Linux kernel. It is designed on the principles of speed, simplicity, scalability, integrity of data, non-linear development and being fully distributed. Before learning Git, please forget everything you know about version control systems—because Git breaks all the traditional practices and workflows about version control systems that you may be familiar with. Git is modelled like a UNIX filesystem rather than a version control system. Integrity of data is implied by its design. If some files are modified or corrupted, Git itself can identify it by default. Since understanding Git's workflow and internals helps a lot, this article walks you through basic Git terminology and the initialisation steps, then moves to a hands-on workflow, which illustrates how to use it while coding.

## Installation

You can install Git from source code, or from your distro's package management system. To install from the latest source, use the following commands:

```
$ wget http://git-core.googlecode.com/files/git-1.7.9.tar.gz
$ tar -xzvvf git-1.7.9.tar.gz
$ cd git-1.7.9
$ make install
```

On Ubuntu, a simple *sudo apt-get install git-core* will do the needful.

## Basic terminology and concepts

### Blobs, trees and commit

These are the three main data structures used in the Git version control system. A *blob* is used to hold a single file. A *tree* is an object used to model a directory that contains files, as well as other sub-directories (a tree holds blobs and sub-trees). A *commit* holds the current state of the repository. It points to a tree.

Let's get more details on the concepts. A blob is a binary file with the file named as the SHA1 hash of that file. A tree is a binary file that stores references to blobs and trees, also named as the SHA1 hash of the tree object. A commit is again named by the SHA1 hash. These hashes are used for naming, in order to ensure the integrity of data. In Git, files are not addressed by name. Everything is content-addressed. Every commit object has a pointer to the parent commit object. From a given commit, you can reverse-traverse by looking at the parent pointers (like a linked list) to check out the history of the commits. If a commit has multiple parent commits, that means the particular commit is created by merging two branches.

When a Git repository is created, the root directory of the repository contains a special directory named *.git*, which contains all the data required for Git's functioning. All the binary objects (commits, blobs and trees) are stored in the *.git/objects* sub-directory. When you add files into the repository, corresponding blobs are created. When you commit your change with a description, a commit object is created, and it references the current files in the working directory. When you modify a file, Git creates another copy of the file internally, and renames it as the SHA1 hash for that newly created file. Thus, Git handles content-addressable data. When some file is modified manually, Git will see the mismatch between the SHA1 of the file content and its name, and this makes it easier to preserve the integrity of the data.

## The local repository

In Git, you have a full copy of the repository locally. Whatever changes you make are to the local repository. You can work offline without any connectivity to a master server and make commits to the local repository. To sync changes to the master server, you can add the remote repository and push changes into it, or sync your local repository with new changes in the remote repository.

## The working directory and staging area

The working directory is the repository directory where files are checked out. Whenever you want to do a commit after making some modifications, unlike other version control systems, Git will not track the modifications made to all the files in the working directory. When you do a *Git commit*, it will look at the files added to the staging area. Only files in the staging area are considered for a commit. Basically, the staging area is a method by which you can selectively add files in the current working directory for a specific commit. Let's suppose there are 10 files in the directory. You made modifications to four files, but changes to two files belong to a particular feature addition, and other modifications belong to some other feature. You don't want to commit changes related to two features in a single commit, but changes related to one feature in one commit, and other changes in another commit. The staging area helps you to do that, as follows:

```
git add file1 file2 file3
```

…can be used to add files to the staging area, and then run git commit  to add those changes in the staging area to a new commit. After the first commit, add the other files, and do the next commit. The *git add* command basically moves the added files to the staging area; only added files are considered for a commit.

Every object in Git can be referenced by the SHA1 hash (41 characters). Usually, you do not need to type 41 characters—for any reference, only the first five characters of the hash are required.

## Branches and tags

By default, Git uses the branch named *master*, which is just like *trunk* in Subversion—but the concept of branches is different in Git. A branch can be a short-lived branch or long-term, according to Git concepts. Usually we use short-lived branches (called topic-branches) to develop new features; once the feature development is complete, you can merge it back to the main branch (master) and delete the topic-branch. Creating and deleting branches is never expensive in Git. It is part of the Git workflow to create and delete branches.

Every branch is referenced by a branch head (HEAD)—a pointer to the latest commit in that branch. When you make commits to a branch, the head is updated with the latest commit. The heads of branches are listed in the directory *.git/refs/heads/* :

```
$ cat .git/refs/heads/master
46bf1f4592a7f121438a16a9d5e20e16db239814
```

The contents of the master branch head file refer to the commit ID of the latest commit in the master branch.

Tags are very similar to branches—but the difference is that tags are immutable. Once you create a tag for a particular commit, even when you create a new commit, it will not be updated. Suppose you want to tag some particular commit as a version release, you create a tag and keep it.

You can refer to the latest commit in the current working branch using the shorthand 'HEAD'. You can also reference parent commits from the latest current commits using *HEAD~N*  to reference the *n*th older commit from a particular moment. You can run *git show HEAD* to display information about the latest commit.

## Fast-forward and conflicting merges

Merging is a common task when working with version-control systems. Many people may work on the same source code, each developing different portions of it. At some point, you will need to merge all the code into the main branch. Git has its own interesting ways of merging code. There are two terms associated with merging code: *fast-forward merge* and *conflicting merge*. Let us represent our Git repository as a series of commits, each having a parent commit, except for the first commit:

```
A-B-C-D
```

Here, A is the first commit and B, C and D are next commits. D is the HEAD commit (the latest). One of the developers (Dev-1) clones the repo to his local machine, and adds a few more commits. Let us represent his local repo as follows:

```
A-B-C-D-E-F
```

Dev-1 now needs to merge his changes back to the main repository. He can easily do so  because no other change has been committed to the main repository by other developers. The difference between his local repo and the main repo is two commits (E and F); D is still the head commit in the main repo, so by applying E and F, he can make the main repo exactly the same as his local repo. This type of merging, without any conflicts, applying newly added commits to the parent repo, is known as fast-forward merge.

Now, let's look at a conflicting merge version of the above scenario: by the time Dev-1 made changes, some other developer (Dev-2) added some other changes to the main repo. The main repo now looks like what follows:

```
A-B-C-D-G
```

Now, when Dev-1 tries to merge, Git responds that a fast-forward merge cannot be done. Then Dev-1 has to rebase his code on top of the main repo, and resubmit to apply the patches, i.e., the developer has to fetch the code from the main branch, and apply his local commits on top of the commit G. This is fairly simple. Git provides easy methods to resolve conflicts. When Dev-1 tries to merge, it will say that the merge failed and ask him to manually resolve conflicts to continue—and will list the files in which conflicts exist. The developer can open the files and see that Git has inserted markers like '<<<', which separate code from the local branch and code from the other repo. You should then choose the correct lines of code that you need, make the necessary modifications to resolve the issues, delete the '<<<' markers and then continue the merge operation. The operation will succeed and the developer can sync the code to the main repo. Merges and merge conflicts can occur between different repositories as well as branches.

## Basic usage
Let us learn the basic usage commands and flows for Git.

### Setting up the identity
Initially, after installing Git, you need to set up the identity for the committer. You can either set a global identity for user-wide configuration, or create a repository-specific identity. For example, to set a global user identity, use the following commands:

```
$ git config --global user.name "Sarath Lakshman"
$ git config --global user.email "sarathlakshman@slynux.org"
```

If you want to set a repository-specific private configuration, change the working directory to a Git repository and invoke the above commands without the —*global* option.

You can list all the Git configurations with (repository) *git config list*, or (global) *git config —global list*. You can check the current user name and email with *git config user.name* and *git config user.email*.

### Creating an empty repository
To create a repository in the current directory, use *git init*.

### Adding files to the staging area
Every time you make a change to a file in the repository, you have to add that file to the staging area before making a commit. Use *git add file1 file2* to add the file(s). It is also possible to select code chunks to be added from a file using the *-p* (prompt) option with *add: git add -p filename*. You can select the code chunk using the *y/n* prompt.

### Committing changes from the staging area
When you run *git commit*, it asks for a commit message. A nice commit message has a one-line subject with a length of 65 characters, and a multi-line description of 70 characters, separated by a blank line. The Git *commit* command takes the following options:

- *git commit -m "Message"*
- *git commit -a* (all modified files in the current directory are moved to the staging area and added to the commit)
- *git commit file1 file2* (If you want to move *file1* and *file2* to the staging area and create the commit in one shot, files can be specified with *git commit*.)

If you created a commit, and think that the message was not right, or some more code should be added to the commit, you can modify the commit using *–amend*. To interactively modify the commit, step by step, use *git commit –amend -i*.

### Looking at the Git log
Commit logs are very important for version control systems. To view the commit logs, use the following commands:

```
$ git log
$ git log -2 (show last two commits)
$ git log -p
```

The -p option stands for *pretty*. It also displays the differences between the last commits.

```
$ git log --since 'May 22 2012'
$ git log --before "2 days"
$ git log --after "3 days"
```

It is also possible to list the log related to a commit by giving the commit ID (hash) as the parameter to the Git log command (*git log commit_id*).

### Status of the repository
The *git status* command is very important. It is used to display the current status of the repository, and will show the contents

of the working directory, the unmodified files, modified files and staged files.

### Displaying differences

We can view the differences between different commits, or between the last commit and the modified files in the current directory, or staged files. To see the difference between the last commit and the changes made in the repo, use *git diff*. To see the difference between the last commit and the staging area (files added using *git add*, but not committed), use *git diff --staged*.

It is also possible to see the difference between a given commit ID and the current modified files with *git diff commit_id*. To see the difference between current files against a different branch, use *git diff branchname*. It is possible to see how many lines got added or removed using *git diff --stat*. To display the difference between the current repo and an *nth* old commit, use *git diff HEAD~N*.

### Resetting or reverting changes

After making a change, you may later decide to remove it. The Git *reset* command is used to reset some changes. There are three different types of resetting—soft, hard and mixed. Let us look at each of them.

When you add some changes to the staging area using *git add*, and those changes haven't been committed, you can use *git reset* to remove those files/changes from the staging area. The actual changes made to the working copy of the file are unaffected. The default Git reset is equivalent to *git reset –mixed*.

If you want to reset the HEAD of the current branch to some commit, *git reset –soft COMMIT_ID* can be used. It will not destroy anything, but it will just reset the HEAD.

The *hard* option will do three things; it will clear the staging area, reset the HEAD to a recent HEAD or specified COMMIT_ID and delete the local file modifications.

### Checking out

If you want to get files from a specified commit or branch, you can use *git checkout branchname*. You can also check out only a single file from a branch with *git checkout master filename*.

### Cloning a remote repository

Any public Git repository can be cloned to a local repository using the *git clone* command, as *git clone repo_path*. Here, *repo_path* can be an HTTP URL, local UNIX-style path or SSH location.

### Pushing local commits to a remote repository

You make changes to your local repository and also need to push the changes to the main repository frequently. You can push changes with *git push remote_repo branchname*. If you created your local repo by cloning the main repo, the *remote_repo* name will be *origin*. You can push the changes to the master branch with *git push origin master*. If it is a fast-forward merge, it will succeed. If there are conflicts, you have to rebase your code and push it again.

### Branches

In Git's workflow, branches are very important. You can check which branch you have currently checked out using the *git branch* command, which lists different branches and shows a * mark on the currently checked out branch.

You can create a new branch using *git branch new-topic-branch*. You can switch between branches using *git checkout branchname*. There is a shortcut to create and switch to the new branch:

*git checkout -b newbranch*.

A branch can be deleted using the *-D* option, such as *git branch -D new-topic-branch*.

### Managing a remote repository

Often, you work on a local repository on your machine, but may need to sync with multiple remote branches. GitHub is a social code-hosting website that enables you to create remote repositories. Create a *github.com* account to play around and share your code.

You can list the different remote repositories that are linked to your local repository with *git remote*; add the –v switch for more verbose output. A remote repository can be linked to the current local Git repository with *git remote add repo_name repo_path*. For example:

```
$ git remote add my_test_repo git@github.com:slynux/coderepo.git
$ git remote #List remote repos for this repository
my_test_repo
```

All branches, including remote repo branches, can be listed with *git branch -a*. You can delete a branch from a remote repository with *git push repo_name :branch_name*. If the current repo is cloned from the remote repo, use *git push origin :branch_name*.

In this article, we have covered the essential concepts related to Git and the basic commands to work with it. My next article will cover more advanced features, including merges, rebasing, tagging, cherry-picking and an example workflow by integrating with GitHub. Happy hacking! END

**By: Sarath Lakshman**

The author is a software engineer as well as a hacktivist of Free and Open Source Software and passionate about UNIX like operating systems and design. He has recently authored the Linux Shell Scripting Cookbook, which gives insights on shell scripting through 119 recipes. He can be reached via his website *http://www.sarathlakshman.com*.