**Introduction to GIT and GITHUB**

Git is a distributed revision control and source code management system with an emphasis on speed. Git was initially designed and developed by Linus Torvalds for Linux kernel development. Git is a free software distributed under the terms of the GNU General Public License version 2. This tutorial explains how to use Git for project version control in a distributed environment while working on web-based and non web-based applications development.

**Audience**

This tutorial will help beginners learn the basic functionality of Git version control system. After completing this tutorial, you will find yourself at a moderate level of expertise in using Git version control system from where you can take yourself to the next levels.

**Prerequisites**

We assume that you are going to use Git to handle all levels of Java and Non-Java projects. So it will be good if you have some amount of exposure to software development life cycle and working knowledge of developing web-based and non web-based applications.

**GIT – Basics**

**Version Control System**

**Version Control System (VCS)** 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).

In this chapter, we will concentrate only on distributed version control system and especially on Git. Git falls under distributed version control system.

**Distributed Version Control System**

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. Unfortunately, if the central server goes down for an hour, then during that hour, no one can collaborate at all. And even in a worst case, if the disk of the central server gets corrupted and proper backup has not been taken, then you will lose the entire history of the project. Here, distributed version control system (DVCS) comes into picture.

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.

**Advantages of Git**

**Free and open source**

Git is released under GPL’s open source license. It is available freely over the internet. You can use Git to manage property projects without paying a single penny. As it is an open source, you can download its source code and also perform changes according to your requirements.

**Fast and small**

As most of the operations are performed locally, it gives a huge benefit in terms of speed. Git does not rely on the central server; that is why, there is no need to interact with the remote server for every operation. The core part of Git is written in C, which avoids runtime overheads associated with other high-level languages. Though Git mirrors entire repository, the size of the data on the client side is small. This illustrates the efficiency of Git at compressing and storing data on the client side.

**Implicit backup**

The chances of losing data are very rare when there are multiple copies of it. Data present on any client side mirrors the repository, hence it can be used in the event of a crash or disk corruption.

**Security**

Git uses a common cryptographic hash function called secure hash function (SHA1), to name and identify objects within its database. Every file and commit is check-summed and retrieved by its checksum at the time of checkout. It implies that, it is impossible to change file, date, and commit message and any other data from the Git database without knowing Git.

**No need of powerful hardware**

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.

**Easier branching**

CVCS uses cheap copy mechanism, If we create a new branch, it will copy all the codes to the new branch, so it is time-consuming and not efficient. Also, deletion and merging of branches in CVCS is complicated and time-consuming. But branch management with Git is very simple. It takes only a few seconds to create, delete, and merge branches.

**DVCS Terminologies**

**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.

Suppose you modified two files, namely “sort.c” and “search.c” and you want two different commits for each operation. You can add one file in the staging area and do commit. After the first commit, repeat the same procedure for another file.



# First commit

[bash]$ git add sort.c

# adds file to the staging area

[bash]$ git commit –m “Added sort operation”

# Second commit

[bash]$ git add search.c

# adds file to the staging area

[bash]$ git commit –m “Added search operation”

### Blobs

Blob stands for **B**inary **L**arge **Ob**ject. Each version of a file is represented by blob. A blob holds the file data but doesn’t contain any metadata about the file. It is a binary file, and in Git database, it is named as SHA1 hash of that file. In Git, files are not addressed by names. Everything is content-addressed.

### 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 and trees which are also named as **SHA1** hash of the tree object.

### Commits

Commit holds the current state of the repository. A commit is also named by **SHA1** hash. You can consider a commit object as a node of the linked list. Every commit object has a pointer to the parent commit object. From a given commit, you can traverse back by looking at the parent pointer to view the history of the commit. If a commit has multiple parent commits, then that particular commit has been created by merging two branches.

### 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. Every branch is referenced by HEAD, which points to the latest commit in the branch. Whenever you make a commit, HEAD is updated with the latest commit.

### Tags

Tag assigns a meaningful name with a specific version in the repository. Tags are very similar to branches, but the difference is that tags are immutable. It means, tag is a branch, which nobody intends to modify. Once a tag is created for a particular commit, even if you create a new commit, it will not be updated. Usually, developers create tags for product releases.

### Clone

Clone operation creates the instance of the repository. Clone operation not only checks out the working copy, but it also mirrors the complete repository. Users can perform many operations with this local repository. The only time networking gets involved is when the repository instances are being synchronized.

### Pull

Pull operation copies the changes from a remote repository instance to a local one. The pull operation is used for synchronization between two repository instances. This is same as the update operation in Subversion.

### Push

Push operation copies changes from a local repository instance to a remote one. This is used to store the changes permanently into the Git repository. This is same as the commit operation in Subversion.

### 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. The heads of the branches are stored in **.git/refs/heads/** directory.

[CentOS]$ ls -1 .git/refs/heads/

master

[CentOS]$ cat .git/refs/heads/master

570837e7d58fa4bccd86cb575d884502188b0c49

### Revision

Revision represents the version of the source code. Revisions in Git are represented by commits. These commits are identified by **SHA1** secure hashes.

### URL

URL represents the location of the Git repository. Git URL is stored in config file.

[tom@CentOS tom\_repo]$ pwd

/home/tom/tom\_repo

[tom@CentOS tom\_repo]$ cat .git/config

[core]

repositoryformatversion = 0

filemode = true

bare = false

logallrefupdates = true

[remote "origin"]

url = gituser@git.server.com:project.git

fetch = +refs/heads/\*:refs/remotes/origin/\*

## Installation of Git Client

If you are using Debian base GNU/Linux distribution, then **apt-get** command will do the needful.

[ubuntu ~]$ sudo apt-get install git-core

[sudo] password for ubuntu:

[ubuntu ~]$ git --version

git version 1.8.1.2

And if you are using RPM based GNU/Linux distribution, then use **yum** command as given.

[CentOS ~]$

su -

Password:

[CentOS ~]# yum -y install git-core

[CentOS ~]# git --version

git version 1.7.1

## Customize Git Environment

Git provides the git config tool, which allows you to set configuration variables. Git stores all global configurations in **.gitconfig** file, which is located in your home directory. To set these configuration values as global, add the **--global** option, and if you omit **--global** option, then your configurations are specific for the current Git repository.

You can also set up system wide configuration. Git stores these values in the **/etc/gitconfig** file, which contains the configuration for every user and repository on the system. To set these values, you must have the root rights and use the **--system** option.

When the above code is compiled and executed, it produces the following result −

### Setting username

This information is used by Git for each commit.

[jerry@CentOS project]$ git config --global user.name "Jerry Mouse"

### Setting email id

This information is used by Git for each commit.

[jerry@CentOS project]$ git config --global user.email "jerry@tutorialspoint.com"

### Avoid merge commits for pulling

You pull the latest changes from a remote repository, and if these changes are divergent, then by default Git creates merge commits. We can avoid this via following settings.

jerry@CentOS project]$ git config --global branch.autosetuprebase always

### Color highlighting

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

[jerry@CentOS project]$ git config --global color.ui true

[jerry@CentOS project]$ git config --global color.status auto

[jerry@CentOS project]$ git config --global color.branch auto

### Setting default editor

By default, Git uses the system default editor, which is taken from the VISUAL or EDITOR environment variable. We can configure a different one by using git config.

[jerry@CentOS project]$ git config --global core.editor vim

### Setting default merge tool

Git does not provide a default merge tool for integrating conflicting changes into your working tree. We can set default merge tool by enabling following settings.

[jerry@CentOS project]$ git config --global merge.tool vimdiff

### Listing Git settings

To verify your Git settings of the local repository, use **git config –list** command as given below.

[jerry@CentOS ~]$ git config --list

The above command will produce the following result.

user.name=Jerry Mouse

user.email=jerry@tutorialspoint.com

push.default=nothing

branch.autosetuprebase=always

color.ui=true

color.status=auto

color.branch=auto

core.editor=vim

merge.tool=vimdiff

**Life Cycle of GIT**

General workflow is as follows −

* You clone the Git repository as a working copy.
* You modify the working copy by adding/editing files.
* If necessary, you also update the working copy by taking other developer's changes.
* You review the changes before commit.
* You commit changes. If everything is fine, then you push the changes to the repository.
* After committing, if you realize something is wrong, then you correct the last commit and push the changes to the repository.

Shown below is the pictorial representation of the work-flow.



**How to create a REMOTE Repository:**

we will see how to create a remote Git repository; from now on, we will refer to it as Git Server. We need a Git server to allow team collaboration.

## Create New User

# add new group

[root@CentOS ~]# groupadd dev

# add new user

[root@CentOS ~]# useradd -G devs -d /home/gituser -m -s /bin/bash gituser

# change password

[root@CentOS ~]# passwd gituser

## The above command will produce the following result.

Changing password for user gituser.

New password:

Retype new password:

passwd: all authentication token updated successfully.

## Create a Bare Repository

Let us initialize a new repository by using **init** command followed by **--bare** option. It initializes the repository without a working directory. By convention, the bare repository must be named as **.git**.

[gituser@CentOS ~]$ pwd

/home/gituser

[gituser@CentOS ~]$ mkdir project.git

[gituser@CentOS ~]$ cd project.git/

[gituser@CentOS project.git]$ ls

[gituser@CentOS project.git]$ git --bare init

Initialized empty Git repository in /home/gituser-m/project.git/

[gituser@CentOS project.git]$ ls

branches config description HEAD hooks info objects refs

## Generate Public/Private RSA Key Pair

Let us walk through the process of configuring a Git server, **ssh-keygen** utility generates public/private RSA key pair, that we will use for user authentication.

Open a terminal and enter the following command and just press enter for each input. After successful completion, it will create a **.ssh** directory inside the home directory.

tom@CentOS ~]$ pwd

/home/tom

[tom@CentOS ~]$ ssh-keygen

The above command will produce the following result.

Generating public/private rsa key pair.

Enter file in which to save the key (/home/tom/.ssh/id\_rsa): **Press Enter Only**

Created directory '/home/tom/.ssh'.

Enter passphrase (empty for no passphrase): **---------------> Press Enter Only**

Enter same passphrase again: **------------------------------> Press Enter Only**

Your identification has been saved in /home/tom/.ssh/id\_rsa.

Your public key has been saved in /home/tom/.ssh/id\_rsa.pub.

The key fingerprint is:

df:93:8c:a1:b8:b7:67:69:3a:1f:65:e8:0e:e9:25:a1 tom@CentOS

The key's randomart image is:

+--[ RSA 2048]----+

| |

| |

| |

|

.

|

| Soo |

| o\*B. |

| E = \*.= |

| oo==. . |

| ..+Oo

|

+-----------------+

**ssh-keygen** has generated two keys, first one is private (i.e., id\_rsa) and the second one is public (i.e., id\_rsa.pub).

**Note:** Never share your PRIVATE KEY with others.

## Adding Keys to authorized\_keys

Suppose there are two developers working on a project, namely Tom and Jerry. Both users have generated public keys. Let us see how to use these keys for authentication.

Tom added his public key to the server by using **ssh-copy-id** command as given below −

[tom@CentOS ~]$ pwd

/home/tom

[tom@CentOS ~]$ ssh-copy-id -i ~/.ssh/id\_rsa.pub gituser@git.server.com

The above command will produce the following result.

gituser@git.server.com's password:

Now try logging into the machine, with "ssh 'gituser@git.server.com'", and check in:

.ssh/authorized\_keys

to make sure we haven't added extra keys that you weren't expecting.

Similarly, Jerry added his public key to the server by using ssh-copy-id command.

[jerry@CentOS ~]$ pwd

/home/jerry

[jerry@CentOS ~]$ ssh-copy-id -i ~/.ssh/id\_rsa gituser@git.server.com

The above command will produce the following result.

gituser@git.server.com's password:

Now try logging into the machine, with "ssh 'gituser@git.server.com'", and check in:

.ssh/authorized\_keys

to make sure we haven't added extra keys that you weren't expecting.

## Push Changes to the Repository

We have created a bare repository on the server and allowed access for two users. From now on, Tom and Jerry can push their changes to the repository by adding it as a remote.

Git init command creates **.git** directory to store metadata about the repository every time it reads the configuration from the **.git/config** file.

Tom creates a new directory, adds README file, and commits his change as initial commit. After commit, he verifies the commit message by running the **git log** command.

[tom@CentOS ~]$ pwd

/home/tom

[tom@CentOS ~]$ mkdir tom\_repo

[tom@CentOS ~]$ cd tom\_repo/

[tom@CentOS tom\_repo]$ git init

Initialized empty Git repository in /home/tom/tom\_repo/.git/

[tom@CentOS tom\_repo]$ echo 'TODO: Add contents for README' > README

[tom@CentOS tom\_repo]$ git status -s

?? README

[tom@CentOS tom\_repo]$ git add .

[tom@CentOS tom\_repo]$ git status -s

A README

[tom@CentOS tom\_repo]$ git commit -m 'Initial commit'

The above command will produce the following result.

[master (root-commit) 19ae206] Initial commit

1 files changed, 1 insertions(+), 0 deletions(-)

create mode 100644 README

Tom checks the log message by executing the git log command.

[tom@CentOS tom\_repo]$ git log

The above command will produce the following result.

commit 19ae20683fc460db7d127cf201a1429523b0e319

Author: Tom Cat <tom@tutorialspoint.com>

Date: Wed Sep 11 07:32:56 2013 +0530

Initial commit

Tom committed his changes to the local repository. Now, it’s time to push the changes to the remote repository. But before that, we have to add the repository as a remote, this is a one-time operation. After this, he can safely push the changes to the remote repository.

**Note** − By default, Git pushes only to matching branches: For every branch that exists on the local side, the remote side is updated if a branch with the same name already exists there. In our tutorials, every time we push changes to the **origin master** branch, use appropriate branch name according to your requirement.

[tom@CentOS tom\_repo]$ git remote add origin gituser@git.server.com:project.git

[tom@CentOS tom\_repo]$ git push origin master

The above command will produce the following result.

Counting objects: 3, done.

Writing objects: 100% (3/3), 242 bytes, done.

Total 3 (delta 0), reused 0 (delta 0)

To gituser@git.server.com:project.git

\* [new branch]

master −> master

Now, the changes are successfully committed to the remote repository.