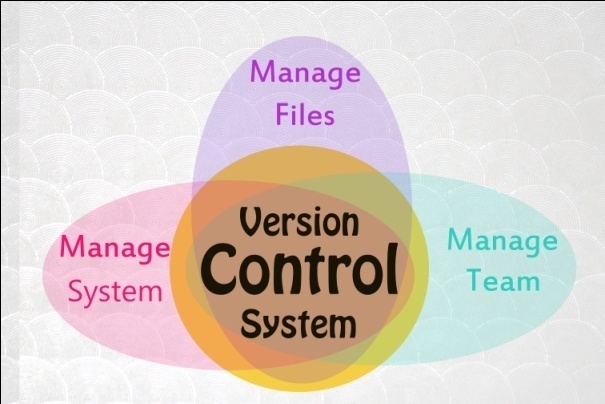
Git is a distributed Version control System. What is a distributed in here. Git has a feature that is not available in other Version Control systems and that is it allows developers to work on a project without requiring them to share a common network. In this page we will see more article on Git and its working,

Understanding Version Control System



Managing code is always hard. In the Early days when we used to write code, we usually save them to a disk location for future references. One developer working on a project knows where the code is saved, what changed and how it works.

But what if the project is written by multiple people. Saving the code to a same location on the disk can be conflicting. One user makes the changes and other gets confused with them. Moving the code to production can be very confusing.

What is

* We made a change to the code, realised it was a mistake and wanted to revert back?
* We lost the code?
* What is we lost the code and we had a backup which is very old then new code?
* What if i need to maintain multiple versions of the same code for different projects?
* How can we prove that a particular change has broken the code or fixed the code?
* What is we need to submit a change to some others code?
* What if we want to see how much work is being done, and where and by whom?
* What if we need to experiment with the new feature without interfering with the working copy of the code?

Source Code Management System is an answer for the above problems?

Source Code Management System ( SCM ) or Version Control System ( VCS ) or Revision Control System ( RVCS ) is a system that records changes to a file or a set of files over time.

VCS allows you to track the history of a collection of files by creating different version of the collection of file ( or files). Each version captures a snapshot of the files at a certain point of time. Vcs allows to switch between these versions. These versions are stored place  specific place called repository.

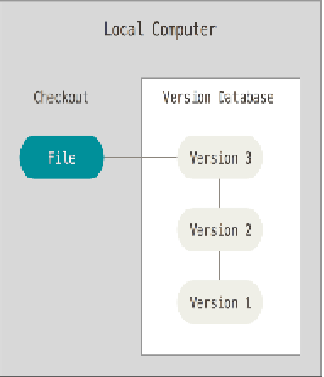
Vcs also allows us to revert files back to a previous state, revert a project back to its previous good state, compare changes over time, can show who made changes lastly, whose code introduced a bug etc.

Lastly if we screw things up or loose files we can easily recover using Vcs

Now that we understood what Vcs is , we will see the types of Vcs , their advantages and disadvantages.

**Local Version Control System** - In this system, developers store multiple version of the files in separate directories and used them when needed. This is a pretty easy and simple to use with smaller projects but very error prone.

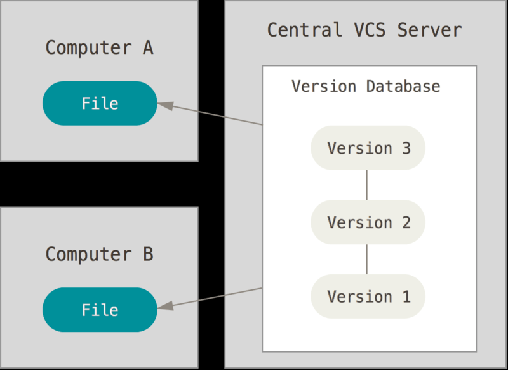
Most of the times , it is easy to forget which directory you are in and accidently write to a wrong file or copy over files that we don't mean to. In order to deal with this situation we came up with the Local Version Control System.



Lvcs has a simple database that kept all the changed to files under revision control. A revision control system is capable of reverting a modification done to a file to its earlier state. It allows users to identify and correct errors and provide security to the data and information.

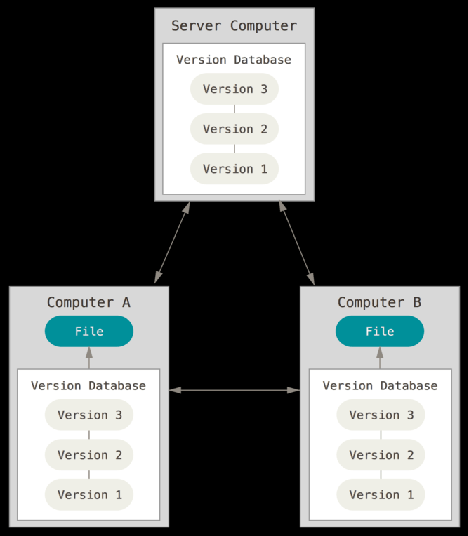
Though this used to work well, the problem with this is that if developers want to collaborate with other developers the Lvcs are not suitable. Since we make all the code changes or code saving to a local version control system, other developers working on different systems can’t access this database. In order to solve thisproblem , people came up with the Centralized Version Control System ( Cvcs ).

**Centralized Version Control System** - In order to solve multiple users accessing the Vcs , Cvcs was created. In this a Single independent server is maintained which will have all users versioned files, and multiple users or clients can connect to this machine to download the code or make changes to the code. The problem with this type of system is a single point of failure. If this single server containing all users code is crashed , then there is no way we can get back the code. Some of these type of tools include CVS, Subversion, perforce etc



In order to solve this single point of failure server, people came up with the Distributed Version Control System ( Dvcs )

**Distributed Version Control System** - In this type of system, there will be a single server where all the source code will be available. New users or existing users who want the code will download the code. In this case they don’t just download the code but mirror the full repository. If the machine where the source code exists crashes or dies , users don’t need to worry about the source code sinceevery one will have the full mirror of the source code. Repositories mirrored on the client machine can be used to restore the repository on the server machine.



Finally what does the Version Control provides?

**Backup/Restore**:  Files saved can be backed up or restored to a specific moment in time. Need a File to be changed to a version last year , we have that

**Synchronization:**Lets people share files and stay up-to-date with the latest version

**Track Changes –**As details about the files updated, merged, deleted will be available in the history maintained.

**Branching and merging**. A larger sandbox. You can **branch** a copy of your code into a separate area and modify it in isolation (tracking changes separately). Later, you can **merge** your work back into the common area.

## Introducing GIT

Git is a distributed Version control System. What is a distributed in here. Git has a feature that is not available in other Version Control systems and that is it allows developers to work on a project without requiring them to share a common network.

Much like in other Systems, Git maintains a Repository locally and developer will make all changes to the local. Once the developer thinks that changes needs to be pushed, then he commits changes from the local repository to the remote (main) repository.

The available version control tools are much like peer-to-peer approach. Git gives us the client-server approach. Rather than a single, central repository on which clients synchronize, each peer's working copy of the codebase is a complete repository

So every Git working directory in a machine is a full-fledged repository with complete history and full version tracking capabilities independent to the network access or a central server. Git when configured contains 2 data structures. A Stage location (or cache) that caches information regarding the working directory and next version to be committed. The other one is a object database

The files when pushed to the GIT repository are stored in the Object Database. It follows a process when storing the files,

1) Blob (Binary large Object) is stored with the contents of the file.

2) A Tree object which holds the structure of the directory being stored. This describes a snapshot of the source tree. This contains a list of file names with the blob information that has the file contents.

3) There exists another object like container which contains information regarding the commit object corresponding to a particular release of the data being tracked by Git.

The index serves as connection point between the object database and the working tree.

The above objects are identified by a SHA-A hash of its contents. The computation is done by GIT and uses the value for the object name. The object is put into a directory matching the first two characters of its hash. The rest of the hash is used as the file name for that object.

The blob objects are compressed using the Zlib compression. GIT also uses other compression tools to compress this Zlib blob files. Git servers typically listen on[TCP port](https://en.wikipedia.org/wiki/TCP_and_UDP_port) 9418

Git also provides ways to clean objects. Every object in the Git database which is not referred to may be cleaned up by using a garbage collection command, or automatically. This is due to the way blobs and objects are linked and references.

**Why do we need GIT – Svn vs Git**

As we do have many version control tools available in market? Why do we need to go to Git?

Git as said is distributed. This is the main difference.

So consider a case, where you want to go back to 3 years for some code. In othertools , this can be complex. The repository may be in a different location that we cannot reach or we cannot commit. Now If you want to make a copy of your code, you have to literally copy/paste it.

With Git, you do not have this problem. Your local copy is a repository, and you can commit to it and get all benefits of source control. When you regain connectivity to the main repository, you can commit against it.

Some other differences include,

1) Git has a Clean command. Every Source control tool dumps extra files , git provides us the facility to clean these with commands which still need to be available in SVN

2) SVN creates .svn directories in every single folder (Git only creates *one* .gitdirectory). Every script you write, and every grep you do, will need to be written to ignore these .svn directories.

3) You have to tell SVN whenever you move or delete something. Git will just figure it out.

4) Ignore semantics – If you want to ignore a pattern to coming (such as \*.pyc), it will be ignored for *all*subdirectories. But in SVN it is not possible.

5) GIT allows us to track content of the files rather than just files

6) Branches in GIT are light weight and easy to maintain

7) It's distributed, basically every repository is a branch. It's much easier to develop concurrently and collaboratively than with Subversion, in my opinion. It also makes **offline** development possible.

8) The **staging** area is awesome, it allows you to see the changes you will commit, commit partial changes and do various other stuff.

9) Git repositories are much **smaller in file size** than Subversion repositories. There's only one ".git" directory, as opposed to dozens of ".svn" repositories

10) When we are working with a subversion , we create working copes on the machine by checking-out version. This represents a snapshot in time of what the repository looks like. You update your working copy via updates, and you update the repository via commits.

But with GIT ,we don’t have a snapshot but a full codebase.

11) Want to check out code from last 3 months, we don’t need to connect to the remote repository as in SVN since in git it is available in local only

12) SVN is a single point of failure. That is when the repository on the remote machine fails all fails including the code base too but in the case GIT, every developer has his own repository and there is no single point of failure.

13) SSH with Git – It allows other developers to ssh to a GIT server on a developer machines and access the repository. This does not work in this case of SVN

**Installation** - Installing Git is very easy. Since git is developed Linus who developed Linux OS, git comes by default with the new versions of Linux.

[root@vx111a Downloads]# **yum install git\***

Loaded plugins: langpacks, product-id, subscription-manager

This system is not registered to Red Hat Subscription Management. You can use subscription-manager to register.

Resolving Dependencies

--> Running transaction check

---> Package git.x86\_64 0:1.8.3.1-4.el7 will be installed

--> Processing Dependency: perl-Git = 1.8.3.1-4.el7 for package: git-1.8.3.1-4.el7.x86\_64

--> Processing Dependency: perl(Git) for package: git-1.8.3.1-4.el7.x86\_64

--> Running transaction check

---> Package perl-Git.noarch 0:1.8.3.1-4.el7 will be installed

--> Finished Dependency Resolution

Dependencies Resolved

Once the installation is done ,we can test the git using,

[root@vx11a] git –version

Git version 1.8.3.1

If we see the git version printed, we can confirm the installation is good. We can also use,

[root@vx11a] whereis git

/usr/bin/git

## Git Configuration

While working with code and pushing changes to repository, metadata regarding the changes done, who did the changes are necessary. Some of these configuration details are set by git itself. “Git config” command lets you to configure the git.  
  
There are 3 types of these configurations available in git  
  
**System-wide Configuration** - A system wide git configuration is not very common. Most of the times, the git is configured for a user or a local repository. A system wide configuration can be done by using the same “git config” by passing the “--system” option as “git config --system”   
  
The configuration is saved in the /etc/gitconfig file.  
  
**User level Configuration** - This is the most basic type of configuration. Information like committer or author of a code change can be set using this type of configuration. The configuration can be set using “git config --global” command. The settings are saved in the .gitconfig file in the user home directory.  
  
**Repository level Configuration** - This configuration is specific to the repository. The setting are set using the “git config --system” command and these settings are saved in the location .git/config file. **Basic level Configuration**  
Color highlighting                : git config --global color.ui auto  
Default Editor for git            : git config --global core.editor vim  
User name                          : git config --global user.name "Firstname Lastname"  
Email ID                             : git config --global user.email [your.email@example.org](mailto:your.email@example.org)

## Git - Understanding Repositories

A repository contains the history of a collection of files starting from a certain directory. The process of copying a git repository using git tool is called cloning. Once the cloning is done and repository is available on our local machine we can have the complete history of the repository on our local machine.   
  
Git has 2 types of repositories - bare and non-bare repository  
  
**Non - bare repositories**: this is by default type of repository that we will use most of the times. When there is a repository available and a user clone the repository , the repo that comes to our local machine is called the non-bare repository. The .git directory will be available inside this non-bare repositories.  
  
**Bare repositories** - these are the types of repositories that are created on the server for sharing changing from the developers.

**How things work?**Most times a server admin will create a bare repository on the server. He then configures the ssh. On the other hand , user who want to add changes to this repository first clone this bare repository using the ssh communication. Once they clone the repo, a local copy will be available to this which is a non-bare repo. They then commit changes to the local repo. They then configure their local git with the remote location of the bare repo ( server ). Finally they push the changes from their local to the remote repo.  
  
**Create a bare repository** - Creating a bare repo can be done by using the “git init --bare” command as below,

jagadishAvailable$Wed Feb 20@ mkdir project.git

jagadishAvailable$Wed Feb 20@ cd project.git/

jagadishproject.git$Wed Feb 20@ git init --bare

Initialized empty Git repository in /Volumes/Available/project.git/

jagadishproject.git$Wed Feb 20@ ls -alrt

total 24

drwxrwxr-x  14 root admin  544 Feb 20 06:29 ..

drwxr-xr-x   4 jagadish admin  136 Feb 20 06:29 refs

drwxr-xr-x   4 jagadish admin  136 Feb 20 06:29 objects

drwxr-xr-x   3 jagadish admin  102 Feb 20 06:29 info

drwxr-xr-x  13 jagadish admin  442 Feb 20 06:29 hooks

-rw-r--r--   1 jagadish admin   73 Feb 20 06:29 description

-rw-r--r--   1 jagadish admin  111 Feb 20 06:29 config

drwxr-xr-x   2 jagadish admin   68 Feb 20 06:29 branches

-rw-r--r--   1 jagadish admin   23 Feb 20 06:29 HEAD

drwxr-xr-x  10 jagadish admin  340 Feb 20 06:29 .

In the above, i have created a directory “project.git” and inside the project.git, ran the “git init --bare” command which will create a bare repository. I created the bare repository as user jagadish. I have the credentials for the user.  
  
**Generate public/Private RSA Key Pairs** - in order for the communication to happen between the server machine ( remote repo ) and the clients, we need to configure the ssh communicate between them. Let’s say that we have a user root from a different machine who wants to clone the remote repo and make some changes to the code. Once done he wants to push them back to the remote repo.  
   
In order to configure the remote repo, use the “ssh-keygen” command available. I Am running the ssh-keygen command as root user on the client machine  
  
[root@vx111a docker]# ssh-keygen  
Generating public/private rsa key pair.  
Enter file in which to save the key (/root/.ssh/id\_rsa):   
Enter passphrase (empty for no passphrase):   
Enter same passphrase again:   
Your identification has been saved in /root/.ssh/id\_rsa.  
Your public key has been saved in /root/.ssh/id\_rsa.pub.  
The key fingerprint is:  
8e:d3:a5:5d:f6:a4:7f:5b:b0:e1:1e:5e:f3:3c:16:63 root@vx111a.jas.com  
  
The key's randomart image is:  
+--[ RSA 2048]----+  
| |  
| |  
| |  
| |  
| S . o + |  
| + + o = E |  
| o + . . \*.=|  
| . + \*=|  
| =o=|  
+-----------------+  
Once this is done , we will now have the Public key and private keys available. The public key is available in the ~/.ssh/ id\_rsa.pub file. Never share the private key file.  
**Copy the public key file to the server machine** - Once we have the public key, we need to add that to the server machine ( remote repo ) for further all communications. We can use the “ssh-copy-id” command to copy our public key to the remote machine.

[root@test-machine ~]# ssh-copy-id -i ~/.ssh/id\_rsa.pub jagadish@192.168.31.177

/bin/ssh-copy-id: INFO: Source of key(s) to be installed: "/root/.ssh/id\_rsa.pub"

The authenticity of host '192.168.31.177 (192.168.31.177)' can't be established.

ECDSA key fingerprint is SHA256:lMv3LvIWIsG9MI7ipWHXXP9PZfIrPsSN6KpWrQrDWPI.

ECDSA key fingerprint is MD5:46:92:90:50:0f:7f:ee:e6:2d:07:a6:96:0e:95:e1:90.

Are you sure you want to continue connecting (yes/no)? yes

/bin/ssh-copy-id: INFO: attempting to log in with the new key(s), to filter out any that are already installed

/bin/ssh-copy-id: INFO: 1 key(s) remain to be installed -- if you are prompted now it is to install the new keys

Password:

Number of key(s) added: 1

Now try logging into the machine, with:   "ssh 'jagadish@192.168.31.177'"

and check to make sure that only the key(s) you wanted were added.

I got the Ip address of the remote machine and used the command “ssh-copy-id -i ~/.ssh/id\_rsa.pub jagadish@192.168.31.177”. This will ask for a password and once we enter , it will add the public key of this machine to the remote machine. Once successfully added, we can login to the remote machine using the ssh command.  
  
Now that the communication is success, we now need to clone the repo from the server to the local machine with user root as below,

[root@test-machine ~]# git clone jagadish@192.168.31.177:/Volumes/Available/project.git

Cloning into 'project'...

warning: You appear to have cloned an empty repository.

Once i run the “git clone” command by passing the repo location from the server machine, i was able to clone the repo to my local machine. We then see a directory with the name project. Once we move inside the project we can .git directory available. This is one of the difference between bare and non-bare repositories. In a non-bare repository, the .git directory is created in the project. For the bare repositories, the .git directory is not created but multiple files are created. These files are the same files that exist in .git location of a non-bare repo.  
  
Configuring the Git - Now that we were able to clone the code from the remote machine we now need to configure the git with certain parameters. The include username, email etc. the most important of the configuration is the remote repo configuration.  
  
**Configure user name and email** - To configure user name and email ID, run the below

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

git config --global user.name "rootUser"

**Configure the Remote Repo location** - Once we have done the basic configuration , we need to configure the remote repo location. Once we make any changes to the code, we need to push them to the remote repo location so that the changes are available on the server for other users and clients. To configure the remote repo location run the below,

[root@test-machine project]# git remote add origin jagadish@192.168.31.177:/Volumes/Available/project.git

This will add the origin ( remote repo location ) details to the current git. In order to check that use the command “git remote -v” as,

[root@test-machine project]# git remote -v

origin    jagadish@192.168.31.177:/Volumes/Available/project.git (fetch)

origin    jagadish@192.168.31.177:/Volumes/Available/project.git (push)

Now that we have configured both user configuration and the repo details, let's push some code to the local repo and push the changes to the remote repo

[root@test-machine project]# echo “hello world” >> one

[root@test-machine project]# git add one

[root@test-machine project]# git commit -m "first file: one commit by root user"

[master (root-commit) 424fbce] first file: one commit by root user

1 file changed, 1 insertion(+)

create mode 100644 one

**Review the Changes** - Once our changes has been committed, lets review them before they were pushed to the remote repo. This can be done by using the “git log” command as below,

[root@test-machine project]# git log

commit 424fbce5183e1fe14b42a4c16e23b1fe1a7bc0f2

Author: rootUser

Date:   Wed Feb 20 02:58:25 2019 +0000

   first file: one commit by root user

This command will show the details of the commit history. It will show you all the commits that happened until now on the repository.  
  
Similar to the “git log” command, “git show” command also gives details about the latest commit with few additional details as,

[root@test-machine project]# git show

commit 424fbce5183e1fe14b42a4c16e23b1fe1a7bc0f2

Author: rootUser

Date:   Wed Feb 20 02:58:25 2019 +0000

   first file: one commit by root user

diff --git a/one b/one

new file mode 100644

index 0000000..6d8f933

--- /dev/null

+++ b/one

@@ -0,0 +1 @@

+this is the first one file

**Change the Commit Comments** - If we want to change the comment added during the last commit, we can do that by using the “amend” option with “git commit” command as,

[root@test-machine project]# git commit --amend -m "changed comments by amend command"

[master a1bfd95] changed comments by amend command

1 file changed, 1 insertion(+)

create mode 100644 one

[root@test-machine project]# git log

commit a1bfd951ee1916637aeddf4ee824678bf5a4a8d4

Author: rootUser

Date:   Wed Feb 20 02:58:25 2019 +0000

   changed comments by amend command

**Push the changes to the master**- Now let’s push the code to the master repo using,

[root@test-machine project]# git push origin master

Counting objects: 3, done.

Writing objects: 100% (3/3), 240 bytes | 0 bytes/s, done.

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

To jagadish@10.135.114.53:/Volumes/Available/project.git

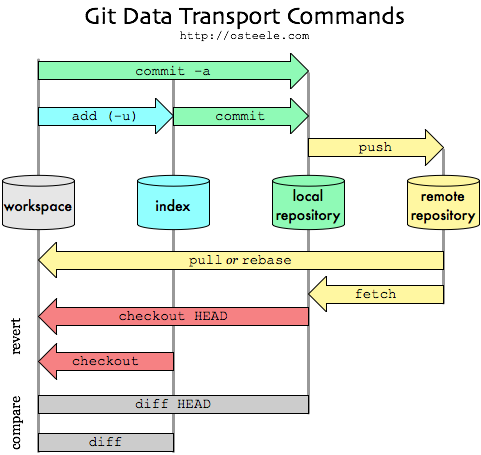
\* [new branch]      master -> master

**Convert a Git repo to Bare repo** - 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 --bare option.

## GIt - Working Tree, Staged ,Committed

Our original repository exists in a separate machine. A mirror of that repository exists in our local machine we call it as local repository. When ever we have completed our code we generally push the code from our local repository to the main repository. There are multiple things that happen before pushing our code from local to main repository.  
  
**Working Tree** - Whenever we want to contribute to a project, we download the source code which is checking out the source code from the main repository. The source code once checked out to our local machine is called working tree. We modify the code that is available in this local repository.   
  
**Staging** - Staging is an intermediate layer between working tree and local repository. Staging is a step before the commit process in git. We make changes to the source code and add them to this staging area. As long as the modified files are in staging area  
  
git allows you to edit it as you like (replace staged files with other versions of staged files, remove changes from staging, etc.).  
  
**Why do we need stage?**  
The practical purpose of the staging is a logical separation of file commits. Let's say i have to make changes to 2 files. Now by the end of day, i was able to make the code changes to the first file and did a half change to the second file. If i did not have stage location and i commit changes directly to the local repository and push to main repo, i'm breaking the code because the second change is only half made.   
  
In this case , i can push the changes to the stage location and then to the local repo. Once i push the local repo to main repo, it is not breaked as we pushed completed changes only.  
  
**Where does the staging area located?**  
The staging area is located inside the .git location. The index file in this location is a single, binary file which lists all files in the current branch including their checksums, time stamps and file names.  
  
**Commit** - this is final step in our local repository. Whatever the code changes that we did we push it to the local repository and this operation is called as “commit”.   
  
**Where is the code pushed to local repository located?**  
The local repository is a hidden directory under the .git location which includes a objects directory. This contains all versions of every single file in the repo ( local branches and copies of the remote branches ) and are compressed as blob files.

The below image represents the process,



Local repository or working tree is where we make the source code changes. A file in the working tree of a git repository can have 4 states. These states are  
  
Untracked - this file is not tracked by the git repository. This means the file is never staged or committed  
Tracked - committed to the local repository  
Staged - code is available in stage location   
dirty/modified - the file is modified but the changes are not staged.

jagadishsample$Mon Feb 18@ git init

Initialized empty Git repository in /Volumes/Available/sample/.git/

jagadishsample$Mon Feb 18@ touch one two three

jagadishsample$Mon Feb 18@ git status

On branch master

No commits yet

Untracked files:

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

    one

    three

    two

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

jagadishsample$Mon Feb 18@

We can see that all files one, two , three are in untracked states. Now do a “git add one” command and we can see the file one moves to the tracked status showed under the status “changes to be committed”

jagadishsample$Mon Feb 18@ git status

On branch master

No commits yet

Changes to be committed:

 (use "git rm --cached ..." to unstage)

    new file:   one

Untracked files:

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

    three

    two

Now lets edit the file one and see the output. The file one will be in both stage mode and in dirty mode as we made the changes.

jagadishsample$Mon Feb 18@ git status

On branch master

No commits yet

Changes to be committed:

 (use "git rm --cached ..." to unstage)

    new file:   one

Changes not staged for commit:

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

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

    modified:   one

Untracked files:

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

    three

    two

Now once we do a git commit, we can see the file one that was staged before making changes is committed and changes that we did is not yet staged.

jagadishsample$Mon Feb 18@ git status

On branch master

Changes not staged for commit:

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

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

    modified:   one

Untracked files:

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

    three

    two

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

Similarly as we use the git status to we have a short status available using “git status -s”. The out put of the git short status is different  
  
File that are not tracked as shown using “??”. New files that have been added to the staging area have an “A”. modified file have an “M”. There are some times 2 column output - the left hand column indicates the status of the staging area and right column indicate the status of the working tree.

jagadishsample$Mon Feb 18@ git status -s

?? one

?? three

?? two

jagadishsample$Mon Feb 18@ git status -s

A  one

?? three

?? two

jagadishsample$Mon Feb 18@ git status -s

AM one

?? three

?? two

jagadishsample$Mon Feb 18@ git status -s

M one

MM three

?? two

If you see the one file is given with “AM” which means that file is in staging mode and also edited ( dirty ). In the last image, we see only “M” for one file which means that file is in working tree but not staged. The “MM” means that file is modified and staged.

## Git - Understanding the .git directory structure

Every git local repository contains a .git directory which is hidden. This .git is the core directory for the local repository and provides many details. Lets understand the contents of the .git location. This directory contains information that lets git to work.

jagadishsample$Mon Feb 18@ tree .git/

.git/

├── HEAD

├── branches

├── config

├── description

├── hooks

│   ├── applypatch-msg.sample

│   ├── commit-msg.sample

│   ├── fsmonitor-watchman.sample

│   ├── post-update.sample

│   ├── pre-applypatch.sample

│   ├── pre-commit.sample

│   ├── pre-push.sample

│   ├── pre-rebase.sample

│   ├── pre-receive.sample

│   ├── prepare-commit-msg.sample

│   └── update.sample

├── info

│   └── exclude

├── objects

│   ├── info

│   └── pack

└── refs

   ├── heads

   └── tags

9 directories, 15 files

**Config** - The file contains the settings for your repository. This will contain url of the remote repo, your mail,user name etc. Every time we use the “git config” command it comes over here.  
  
**Description** - Contains the description of the repository and details. **Info : exclude** - if we want to tell git to ignore certain files for pushing them to the local repo we can use 2 ways,  
  
Adding the files types to the .gitignore file is one way  
Adding the file types that need to be ignored by the git can be added to the exclude file in this info location. This file will not be shared like .gitignore file  
  
**Hooks** - this directory contains certain scripts that can be run automatically at git stages. If you want to execute an action when a commit is done, you can write a post hook script and code your action in here.   
  
Before understanding other directories, let understand commit structure:  
  
**Objects** - A commit is basically a snapshot of the working directory. Every time we create a file and commit it, git compress it and stores in its own data structure. This compressed object will have unique name, hash and will be stored under the objects directory.   
  
Now i have created a file named “one” and committed it to the local repository. We can use the “git show” command to view the commit details as,

jagadishsample$Mon Feb 18@ git show

commit 0bfba39f1e6ddc56a5a55183f8af206e2da9a18b (HEAD -> master)

Author: jagadish

Date:   Mon Feb 18 19:26:22 2019 +0530

   first commit

diff --git a/one b/one

new file mode 100644

index 0000000..e69de29

A commit basically made up of 4 things,  
    Name of the working directory snapshot or hash  
   A Comment  
   Information about the commit details like user etc  
   Hash of the parent commit

Once the commit is done, we can see certain directories created under the objects directory as below,

jagadishsample$Mon Feb 18@ tree .git/objects/

.git/objects/

├── 0b

│   └── fba39f1e6ddc56a5a55183f8af206e2da9a18b

├── 5f

│   └── cffbd6e4c5c5b8d81f5e9314b20e338e3ffff5

├── e6

│   └── 9de29bb2d1d6434b8b29ae775ad8c2e48c5391

├── info

└── pack

5 directories, 3 files

If we can see the commit id starts with the 0b, and the first directory in the object directory also has the name 0b.  
  
The hash of the file created started with 0b and hence the file which is compressed will be stored under the 0b directory in objects. We can see 3 hashes   
  First would be for our files  
  Second hash is for the snapshot created when we committed  
  The third one is for the commit. A Commit is an object in itself and third hash contain            details about that

Now if we uncompress the commit file using

jagadishsample$Mon Feb 18@ git cat-file -p 0bfba39f1e6ddc56a5a55183f8af206e2da9a18b

tree 5fcffbd6e4c5c5b8d81f5e9314b20e338e3ffff5

author jagadish 1550498182 +0530

committer jagadish 1550498182 +0530

first commit

I passed the commit hash which i got by “git show”. It gives me all the details like who is the committer, commit message, author and snapshot hash. If we compare the snapshot hash with the directory structure in objects, we can see both start with same. 

Now if we uncompress the snapshot commit hash,

jagadishsample$Mon Feb 18@ git cat-file -p 5fcffbd6e4c5c5b8d81f5e9314b20e338e3ffff5

100644 blob e69de29bb2d1d6434b8b29ae775ad8c2e48c5391    one

This will give you all the details about the files that we committed.  
  
**HEAD** - the HEAD file always represents a pointer to the branch that we are working on. If we do a “cat HEAD” , we see

jagadish.git$Tue Feb 19@ cat HEAD

ref: refs/heads/master

And if we do a “cat refs/heads/master”, this will refer the branch that we are working currently,

jagadish.git$Tue Feb 19@ cat refs/heads/master

0bfba39f1e6ddc56a5a55183f8af206e2da9a18b

If we observe , we are actually pointing to the commit hash that we did earlier,

jagadish.git$Tue Feb 19@ git show

commit 0bfba39f1e6ddc56a5a55183f8af206e2da9a18b (HEAD -> master)

Author: jagadish

Date:   Mon Feb 18 19:26:22 2019 +0530

   first commit

diff --git a/one b/one

new file mode 100644

index 0000000..e69de29

## Git - Understanding Branching

Branching is most important concept in the version control system. Branching means we can diverge from the main line of development and continue to work without messing the main line of code.  
  
Lets say that we have developed some code and it is available in our master repository. Now when ever we want to make code changes we make the changes on our local repository and push them to remote or master.   
  
We have your master branch and we don't want to mess anything up on that branch. However, we need to implement a new feature that works with the codebase on the master branch. A branch will allow to make a "copy" while not affecting the original branch  
  
Lets understand more about the branches and how they work. Now with the same example as above, if we run

jagadishsample$Tue Feb 19@ git branch

\* master

The “git branch” is the command that lets to create, list, rename and delete branches. To create a branch we can use the “git branch ” command as below,

jagadishsample$Tue Feb 19@ git branch sample

jagadishsample$Tue Feb 19@ git branch --list

\* master

 sample

We can see that an asterisk is set before the master indicating that we are currently working with the master branch. The only thing that happens now is that a new pointer is set to the branch with the asterisk. Until we change the pointer to the new branch we cannot work with the new branch that we created.

jagadishsample$Tue Feb 19@ git checkout sample

Switched to branch 'sample'

jagadishsample$Tue Feb 19@ git branch

  master

\* sample

Now that we changed our branch from master to sample, lets add some content to the sample  
branch.

jagadishsample$Tue Feb 19@ vi two

When we check the status,

jagadishsample$Tue Feb 19@ git status

On branch sample

Untracked files:

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

    two

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

I have a file already available in the same location but i don't see that. This is because the file is already committed. Once we create a branch we will get a copy of the master branch that has file already committed.

Now let's add and commit the file two to the sample branch as,

jagadishsample$Tue Feb 19@ git add .

jagadishsample$Tue Feb 19@ git commit -m "sample branch"

[sample ea4ccb9] sample branch

1 file changed, 2 insertions(+)

create mode 100644 two

We can use the same “git show ” to show the files committed to the sample branch as,

jagadishsample$Tue Feb 19@ git show sample

commit ea4ccb9f6cbf5cb5f4c5c99b4053735aa4d36b9d (HEAD -> sample)

Author: jagadish

Date:   Tue Feb 19 11:19:25 2019 +0530

   sample branch

diff --git a/two b/two

new file mode 100644

index 0000000..da67877

--- /dev/null

+++ b/two

@@ -0,0 +1,2 @@

+this is jagadish in two file

+

We can now confirm that the branch sample has some new code committed and this code is not available with the master branch. We can use the “git show master” to see the files in the master branch.  
  
Now since the new code in our sample branch is not available , lets merge the new branch code to our master branch as,

jagadishsample$Tue Feb 19@ git checkout master

Switched to branch 'master'

Change to the master branch and confirm as,

jagadishsample$Tue Feb 19@ git branch

\* master

  sample

Once we see the asterisk before the master we can then merge our sample branch to our master branch as,

jagadishsample$Tue Feb 19@ git merge sample

Updating 0bfba39..ea4ccb9

Fast-forward

two | 2 ++

1 file changed, 2 insertions(+)

create mode 100644 two

Lets check the master branch now as,

jagadishsample$Tue Feb 19@ git show master

commit ea4ccb9f6cbf5cb5f4c5c99b4053735aa4d36b9d (HEAD -> master, sample)

Author: jagadish

Date:   Tue Feb 19 11:19:25 2019 +0530

   sample branch

diff --git a/two b/two

new file mode 100644

index 0000000..da67877

--- /dev/null

+++ b/two

@@ -0,0 +1,2 @@

+this is jagadish in two file

+

We can see both files are available now which means what ever separate code that we have written as branch is now merged. It is safe to delete a branch now using,

jagadishsample$Tue Feb 19@ git branch -D sample

Deleted branch sample (was ea4ccb9).

## Git - Stashing

Lets say that we are implementing a new feature for the product. We have written some code and suddenly we need to make some changes to master repo. At this moment with half written code, we cannot commit the changes or throw away the code. We need a place where we can save our half written code and restore when we want to work on that again.  
  
Stash operation in git helps us in doing so. The stash take your modified tracked files, stages changes and saves them on a stack of unfinished changes that we can re-apply any time.  
  
Lets create a few files and see how stash works.

[root@test-machine project]# touch three four

[root@test-machine project]# git status -s

?? four

?? three

[root@test-machine project]# vi three

[root@test-machine project]# git add three

[root@test-machine project]# git status -s

AM  three

?? four

Now that we see there are few files that are added and modified, we don't want these changes to be committed or deleted. So we stash them as below,

[root@test-machine project]# git stash

Saved working directory and index state WIP on master: a1bfd95 changed comments  
 by amend command

HEAD is now at a1bfd95 changed comments by amend command

The modified changes will not be moved to a stash location ( temp location ). Now if we see the “git status -s” again,

[root@test-machine project]# git status -s

?? four

We don't see the modified files. Lets see the list of stash available using,

[root@test-machine project]# git stash list

stash@{0}: WIP on master: a1bfd95 changed comments by amend command

In order to get back to the changes that are available on the stash, all we need to do is,

[root@test-machine project]# git stash pop

# On branch master

# Changes to be committed:

#   (use "git reset HEAD ..." to unstage)

#

#    new file:   three

#

# Untracked files:

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

#

#    four

Dropped refs/stash@{0} (e49d4f30fb8ad652059017dea2d4126dadaf44fb)

This will get back the top stash on the stack which is our last one. Now if we check the changes we can see,

[root@test-machine project]# git status -s

A  three

?? four

This way we can save our temporary changes and keep them aside to do our work.

## Git - Understanding Tags

We already understood that branches are separate line of development. When ever we create a branch, we basically take a dump of the master branch code and then make the changes or work on changing the code in that branch. This way we will work on the branch code and will not touch anything in the master branch. Once we are sure that our branch code can be pushed to the master branch we can then do a merge of the 2 branches and once done , we have the updated code in our master.  
  
Let’s say while developing code we want to release the code to the production machine for our testing. So we will take the branch code that is developed and test it. At this moment we will tag the branch code with a version or name and then release that to a machine and then test it.   
  
A tag represents a version of a particular branch at a moment in time. When we create a branch and committed some code to that branch, we can then create a tag using “git tag sample-tag”

jagadishsample$Tue Feb 19@ git tag sample-tag

It is common pattern to use version numbers like v1.4 etc along with a tag. Git supports 2 types of tags, annotated and lightweight.  
  
In the above example we created a lightweight tag. The two types of tags differ with the amount of metadata they store with them. With the annotated tags we store some more meta data like tagger name,email etc. with light weight we only store the name and pointer to the commit. As a best practice annotated tags are for public release since they contain much data and light weight are for private.  
  
Create a annotated tag using,

jagadishsample$Tue Feb 19@ git tag -a simple-tag

This will open the default editor configured and will allow to enter data.  
Lightweight tags are created with no options passed as,

jagadishsample$Tue Feb 19@ git tag sample-tag

**Listing tags** - to list the existing tags use,

jagadishsample$Tue Feb 19@ git tag

sample-tag

simple-tag

Merge to master - Once that we have created the tag for a branch with specific code, we need to push that to the master branch as below using,

jagadishsample$Tue Feb 19@ git merge master sample-tag

Updating ea4ccb9..094b4ac

Fast-forward

tag-example-file | 1 +

1 file changed, 1 insertion(+)

create mode 100644 tag-example-file

**Deleting tags** - deleting a tag is similar to the branch. Pass a “-D” with the tag and it will delete

jagadishsample$Tue Feb 19@ git tag -d sample-tag

Deleted tag 'sample-tag' (was 094b4ac)

## Git – All you need to know about hooks

Unlike other source code management systems, Git provides Hooks. Hooks are scripts that are called when certain events in the work flow of the source control system occur. These Hooks can be used in the work flow like before and after commit etc

Most of these Hooks are something like Pre and Post. So these hooks can be executed either before or after a certain event occurs. Most of these Hooks are executable Scripts. We need to make sure the scripts do the necessary executable permissions. The shebang (#!) in the starting should indicate which interpreter should run the script.

Hooks in Git are pretty simple. Git will execute the Hooks configured for an action. The Hook needs to be a executable for the hook to execute. In this article we will see how to configure these Hooks.

There are 2 types of Hooks,

**Client Side Hooks** – These hooks run the developer system. Developer can edit the local hooks to execute a action.

**Server Side Hooks** – These run the system where we are holding the repository. These are similar to the local hooks (Client Hooks) but they execute on a central repository, or a developer’s public repository.

The available Hooks in Git are located at .git/hooks location. We can see the below hooks available,

​applypatch-msg.sample  
commit-msg.sample  
post-update.sample  
pre-applypatch.sample  
pre-commit.sample  
prepare-commit-msg.sample  
pre-push.sample  
pre-rebase.sample  
update.sample

**Client Hooks**

The Client side Hooks are

**pre-commit** - The pre-commit script is executed every time you run git commit

**prepare-commit-msg** - The prepare-commit-msg hook is called after the pre-commit hook to populate the text editor with a commit message. We can use this to alert the commit message.

**commit-msg** - The commit-msg hook is much like the prepare-commit-msg hook, but it’s called after the user enters a commit message.

**post-commit** - The post-commit hook is called immediately after the commit-msg hook. Mostly used for the notication purpose.

**post-checkout** - The post-checkout hook works a lot like the post-commit hook, but it’s called whenever you successfully check out a reference with git checkout.

**pre-rebase** - The pre-rebase hook is called before git rebase changes anything, making it a good place to make sure something terrible isn’t about to happen.

Now lets see how to configure a basic example hook using the Shell Script. We will use the prepare-commit-msg hook for this example.

1) Change the name of the prepare-commit-msg.sample to prepare-commit-msg

2) Change the permissions on the file

chmod +x prepare-commit-msg

3) Now write a Sample Shell Script in the prepare-commit-msg as

#!/bin/sh

echo "# Please include a useful commit message!" > $1

Now the above message will be displayed when we go for the commit. The message will be displayed before the commit happens.

4) Lets commit the code

[root@vx111a SampleTest]# git commit -m "hello Git Hook"

**[master 68d357a] # Please include a useful commit message!**

1 file changed, 1 insertion(+), 1 deletion(-)

Now from the above output we can see the Commit message that we have added in the Shell Script.

The important thing to remember is to add the Correct Shebang for the script we are writing. For Python we can add the “#!/usr/bin/python2.7” based on your environment.

**Server Hooks**

The Server Hooks are,

**Pre -receive** - The pre-receive hook is executed every time somebody uses git push to push commits to the repository

**Update** - The update hook is called after pre-receive, and it works much the same way

**Post - receive** - The post-receive hook gets called after a successful push operation, making it a good place to perform notifications.

## Git with Git-hub

In this article we will see how we can configure our local repository to connect to the Git hub remote repository and push our code to the remote repository.

1) Initialize a Git repository using "git init"

2) Now create a couple of files. add and commit to the local repository

[root@vx111a Mytest]# git add ReadMe

[root@vx111a Mytest]# git add Hai.java

[root@vx111a Mytest]# cat hai.java

public class hai {

 public static void main(String st[]) {

    System.out.println("This Is Git Sample");

}

}

[root@vx111a Mytest]# git commit -m "testing GIT"

[master (root-commit) bad4374] testing GIT

 2 files changed, 6 insertions(+)

 create mode 100644 Hai.java

 create mode 100644 ReadMe

3) Once the files are added to the local repository we need to add these files to the remote repository but before that we need to add the remote repository details to the Git configuration.

check the remote repository configured for the local repository using,

[root@vx111a Mytest]# git remote show origin

\* remote origin

  Fetch URL: https://github.com/jagadish12/MyTest.git

  Push  URL: https://github.com/jagadish12/MyTest.git

  HEAD branch: master

  Remote branch:

    master tracked

  Local ref configured for 'git push':

    master pushes to master (up to date)

or we can also use,

[root@vx111a Mytest]# git config --get remote.origin.url

https://github.com/jagadish12/MyTest.git

The remote origin is already configured over here , but if that is not configured we dont see the above details.

For adding the remote configuration details, we can use

git remote add origin https://github.com/jagadish12/SampleTest.git

This will add the remote configuration information. Once the remote configuration is added, we can then push our local code committed to the remote repository using,

[root@vx111a Mytest]# git push origin master

Username for 'https://github.com': jagadish12

Password for 'https://jagadish12@github.com':

Counting objects: 3, done.

Delta compression using up to 8 threads.

Compressing objects: 100% (2/2), done.

Writing objects: 100% (2/2), 274 bytes | 0 bytes/s, done.

Total 2 (delta 0), reused 0 (delta 0)

To https://github.com/jagadish12/MyTest.git

   bad4374..089a50e  master -> master

The git Push asks for the Github User name and password for pushing the data to the remote location ( aka remote repository )

Now we can login to the Git Hub and check the updated file changes in the repository that we created.

Now if we want to remote the remote repository configured for a Git, we can use

[root@vx111a test]# git remote remove origin

[root@vx111a test]# git remote add origin https://github.com/jagadish12/SampleTest.git

## Maven with GIT

In this article we will see how we can configure git with Maven. For this we need to create a Git Repository and then create a maven based application on this. The same thing can be further used in Jenkins with Git Configuration.

The important thing to remember here is when you run the maven command make sure you create a directory with the application name and in side initialize the git and run the maven command in there. Create a directory sampleTest and inside that directory follow the steps,

1) run the maven sample web application command as

mvn archetype:generate -DgroupId=com.git.testing -DartifactId=SampleTest -DarchetypeArtifactId=maven-archetype-webapp -DinteractiveMode=false

2) Check the Status and we will see that we have a CounterWebApp application created.

[root@vx111a testing]# git status

# On branch master

#

# Initial commit

#

# Untracked files:

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

#

#       CounterWebApp/

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

[root@vx111a testing]# ll

total 0

drwxr-xr-x. 3 root root 30 Dec  3 14:47 CounterWebApp

Add and Commit

[root@vx111a testing]# git add .

[root@vx111a testing]# git status

# On branch master

#

# Initial commit

#

# Changes to be committed:

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

#

#       new file:   CounterWebApp/pom.xml

#       new file:   CounterWebApp/src/main/webapp/WEB-INF/web.xml

#       new file:   CounterWebApp/src/main/webapp/index.jsp

#

[root@vx111a testing]# git commit -m "Jenkins Testing with Local Repo"

[master (root-commit) 551c28c] Jenkins Testing with Local Repo

 3 files changed, 33 insertions(+)

 create mode 100644 CounterWebApp/pom.xml

 create mode 100644 CounterWebApp/src/main/webapp/WEB-INF/web.xml

 create mode 100644 CounterWebApp/src/main/webapp/index.jsp

Check the Status again which is clean

[root@vx111a testing]# git status

# On branch master

nothing to commit, working directory clean

Note – Make sure you add the <plugins> information in the pom.xml file if you want the web app to be packaged.

## Jenkins with GIT

Jenkins provides us with various options in building the code. On the other hand, git provides us with the Source code management system. Jenkins provides us with various plug-ins in dealing with various repository systems. In this article we will see how we can configure Jenkins with the local Git Repository System.

This article will explain you on how to create a maven web application and use the git repository. The final step is to use the Jenkins with Git in building and packing into a war file.

1) Create a directory SampleTest

2) Inside the sampleTest , run the maven web application command as,

mvn archetype:generate -DgroupId=com.git.testing -DartifactId= sampleTest -DarchetypeArtifactId=maven-archetype-webapp -DinteractiveMode=false

[root@vx111a testing]# git status

# On branch master

#

# Initial commit

#

# Untracked files:

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

#

#       sampleTest /

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

[root@vx111a testing]# ll

total 0

drwxr-xr-x. 3 root root 30 Dec  3 14:47 sampleTest

3) Git add and commit

[root@vx111a testing]# git add .

[root@vx111a testing]# git status

# On branch master

#

# Initial commit

#

# Changes to be committed:

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

#

#       new file:   sampleTest /pom.xml

#       new file:   sampleTest /src/main/webapp/WEB-INF/web.xml

#       new file:   sampleTest /src/main/webapp/index.jsp

#

[root@vx111a testing]# git commit -m "Jenkins Testing with Local Repo"

[master (root-commit) 551c28c] Jenkins Testing with Local Repo

 3 files changed, 33 insertions(+)

 create mode 100644 sampleTest /pom.xml

 create mode 100644 sampleTest /src/main/webapp/WEB-INF/web.xml

 create mode 100644 sampleTest /src/main/webapp/index.jsp

Now if we check the git status, we can see the no untracked files.

[root@vx111a testing]# git status

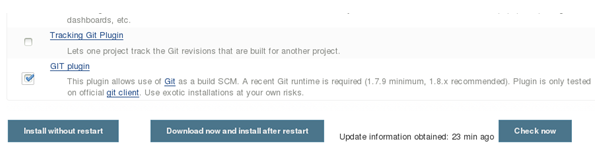
# On branch master

nothing to commit, working directory clean

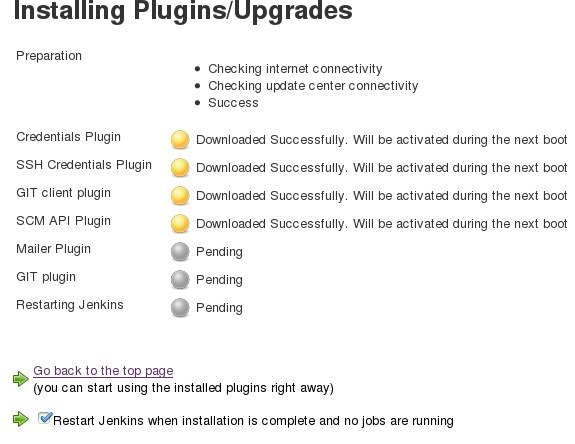
**Configuring Jenkins**

Once the maven application is done and git is configured, the next step is configure local git to jenkins in such a way that the jenkins build the locally developed code.

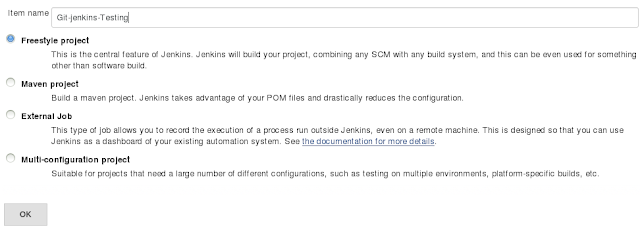
1) Download the Git-plugin using the manage Jenkins in the jenkins Web. Select the Manage Plug-in tab, go to the Available section and search for the GIT plugin. Once Obtained , check the plugin and click "Download now and install after restart"

[](http://3.bp.blogspot.com/-Ks3X7INNrx4/Vn_367vvWpI/AAAAAAAAFrg/Sfk6UrtcRRY/s1600/Jen7.PNG)

2) The Installation goes with the plug-in installation and restart of the Jenkins

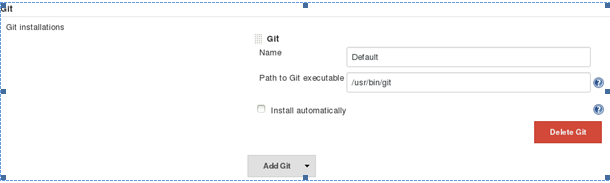
[](http://4.bp.blogspot.com/-avXWccC3W0Y/Vn_5xBh9ilI/AAAAAAAAFr0/Fym-ViNIRQc/s1600/Samp.PNG)

3) Now we will see how we use the installed plug-in along with local git repository to build the maven web application. For this choose the new Item and check for the "Maven project" . Give a Name to the item and click ok.

[](http://2.bp.blogspot.com/-uIlW_dtOlnU/Vn_59lrhIwI/AAAAAAAAFr8/RwwmwVYzkt8/s1600/jenkins9.png)

4) In the next page of the build item,  there are certain configurations that needs to be done.

Provide the Git Executable location

[](http://3.bp.blogspot.com/-QhoYpOrNmeE/Vn_6kPj0nmI/AAAAAAAAFsU/c4k9qDgSS7o/s1600/Snap.PNG)

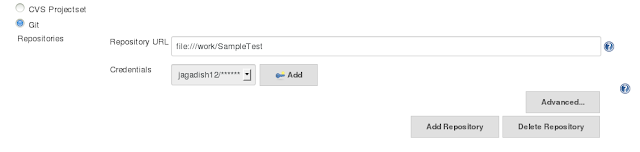
Provide the Git repository user name and Email Details

[](http://1.bp.blogspot.com/-T0fdwBBaihE/Vn_8GlnFEII/AAAAAAAAFsg/rAKKYZpZJL8/s1600/jenkins-GitPath1.png)

Now this is the important part, For building the code from our local git repository and build the code we need to specify the path to the local Git repo as file:// protocol like this:

file:///home/ay/dev/projects/my\_new\_project

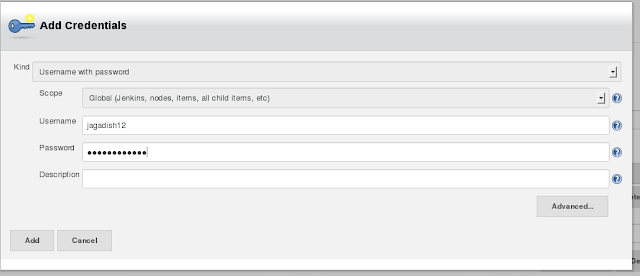
Now for our project we need to configure the Git repository details as,

[](http://3.bp.blogspot.com/-9SEcZ8bdVF8/Vn_8WJTGO8I/AAAAAAAAFso/8kxo3pUFe_g/s1600/jenkins-GitPath4.png)

 If you see the Path of the local Git repository is "file:///work/SampleTest".

Provide the Local Git Credentials. This is one of the important Step in getting the source code from the local Git repository.

Click on the Add credentials under the Repositories in the above screen and provide the credentials that we are using for the local repository

[](http://2.bp.blogspot.com/-44ixR9CHl3U/Vn_6MTAEASI/AAAAAAAAFsI/QHjXY1DnhnM/s1600/jenkins10.png)

Now once the changes are done run the Build now command and see the Console output to get the war file generation

## Git branching Model

While working with the git and Github  it is very important to have a consistent branching model. Developers create many branches so they can work with the code. But having a consistent branch model for our development saves a lot of time. This article talks about the branch model that fits most organisations.

Every team will have 2 main branches develop and master. Parallel to the master branch , another branch exists called develop. So most of the times develop is the one that code gets pushed always.

Master - The origin/master is the main branch where Source Code always reflects to the production ready state.

Develop - Also origin/develop is the main branch where the source code which reflects to the latest delivered development changes for the next releases. Some people call this as the “integration branch”. This is the branch where all the nightly builds use to build and make sure the code is set.

When the source code in the develop branch reach to a stable point and is ready to be released then all changes are merged from develop branch to the master branch. Then master will be tagged with a release number and deployed.

So any changes done to the master branch is a production release automatically. We can use a Git hook or a Jenkins build to identify changes to the master branch, build it and release to the production automatically.

Other helping branches

Other than the master and develop branches there will many other branches that will be created and deleted based on the requirements. These branches are used to aid parallel development between team, feature development, production release , bug fixing, live production problem solving etc.

These branches after some time will eventually be removed. These branches include

Feature Branches

Release Branches

Hotfix Branches

Each of these branches are defined keeping specific purpose in mind and a are bound to strict rules like which branch is the parent and which branch is the target.

Feature branches - Feature branches or sometime called as topic branches are used to develop new features for the upcoming releases. These release can be any time but based on the priority of the feature , a feature branch is created from the develop branch. The parent branch for a feature branch is the develop branch.

The target branch will not be known during the time of the feature branch. It may either merge back to the develop branch or to another feature branch or it can discarded completely ( in case if the feature is not required ).

Release Branch - Release branches are created so that they can support a new production release. The release branch will be created from the develop branch. The develop branch in turn should contain all the features, changes that are agreed for that release. All these changes and features are developed and then merged back to the develop branch. The changes once merged to the develop branch are then merged back to the master branch. Every changes to the master is a new production release by default.

Basically forking a branch from the develop branch for a release starts a release cycle. At this moment no new changes or features can be added to the release branch at this point. Only bug fixes, documentation, testing, security checks and other release oriented tasks should go in this branch. Once this is ready, the release gets merged into the master and tagged with a version number. It should also be merged back to the develop branch which may be progressed since the release was initiated.

Using a separate dedicated branch to prepare releases makes it possible for one team to work only on this branch and polish it for the current releases and other teams work on new features for the next releases.

HotFix Branch - These branch are very special as they are not planned before. These branches are created from the master branch in order to solve a production issues. When a release is done , and if the team identifies a bug or problem they create a branch from the corresponding tag on the master branch that marks the production version.

Once the bug fixing is done for the hotfix branch, it is then merged back to the master branch with a tag and released to production. The changes are again merged back to the develop branch to fix the same bug in the develop branch.

# Git Tutorial: 10 Common Git Problems and How to Fix Them

Learning Git? This Git tutorial covers the 10 most common Git tricks you should know about: how to undo commits, revert commits, edit commit messages, discard local files, resolve merge conflicts, and more.

**1. Discard local file modifications**

Sometimes the best way to get a feel for a problem is diving in and playing around with the code. Unfortunately, the changes made in the process sometimes turn out to be less than optimal, in which case reverting the file to its original state can be the fastest and easiest solution:

git checkout -- Gemfile # reset specified path

git checkout -- lib bin # also works with multiple arguments

In case you’re wondering, the double dash (--) is a common way for command line utilities to signify the end of command options.

**2. Undo local commits**

Alas, sometimes it takes us a bit longer to realize that we are on the wrong track, and by that time one or more changes may already have been committed locally. This is when git reset comes in handy:

git reset HEAD~2 # undo last two commits, keep changes

git reset --hard HEAD~2 # undo last two commits, discard changes

Be careful with the --hard option! It resets your working tree as well as the index, so all your modifications will be lost for good.

**3. Remove a file from git without removing it from your file system**

If you are not careful during a git add, you may end up adding files that you didn’t want to commit. However, git rm will remove it from both your staging area, as well as your file system, which may not be what you want. In that case make sure you only remove the staged version, and add the file to your .gitignore to avoid making the same mistake a second time:

git reset filename # or git remove --cached filename

echo filename >> .gitignore # add it to .gitignore to avoid re-adding it

**4. Edit a commit message**

Typos happen, but luckily in the case of commit messages, it is very easy to fix them:

git commit --amend # start $EDITOR to edit the message

git commit --amend -m "New message" # set the new message directly

But that’s not all git-amend can do for you. Did you forget to add a file? Just add it and amend the previous commit!

git add forgotten\_file

git commit --amend

Please keep in mind that --amend actually will create a new commit which replaces the previous one, so don’t use it for modifying commits which already have been pushed to a central repository. An exception to this rule can be made if you are absolutely sure that no other developer has already checked out the previous version and based their own work on it, in which case a forced push (git push --force) may still be ok. The --force option is necessary here since the tree’s history was locally modified which means the push will be rejected by the remote server since no fast-forward merge is possible.

**5. Clean up local commits before pushing**

While --amend is very useful, it doesn’t help if the commit you want to reword is not the last one. In that case an interactive rebase comes in handy:

git rebase --interactive

# if you didn't specify any tracking information for this branch

# you will have to add upstream and remote branch information:

git rebase --interactive origin branch

This will open your configured editor and present you with the following menu:

pick 8a20121 Upgrade Ruby version to 2.1.3

pick 22dcc45 Add some fancy library

# Rebase fcb7d7c..22dcc45 onto fcb7d7c

#

# 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.

#

# Note that empty commits are commented out

On top you’ll see a list of local commits, followed by an explanation of the available commands. Just pick the commit(s) you want to update, change pick to reword (or r for short), and you will be taken to a new view where you can edit the message.

However, as can be seen from the above listing, interactive rebases offer a lot more than simple commit message editing: you can completely remove commits by deleting them from the list, as well as edit, reorder, and squash them. Squashing allows you to merge several commits into one, which is something I like to do on feature branches before pushing them to the remote. No more “Add forgotten file” and “Fix typo” commits recorded for eternity!

**6. Reverting pushed commits**

Despite the fixes demonstrated in the previous tips, faulty commits do occasionally make it into the central repository. Still this is no reason to despair, since git offers an easy way to revert single or multiple commits:

git revert c761f5c # reverts the commit with the specified id

git revert HEAD^ # reverts the second to last commit

git revert develop~4..develop~2 # reverts a whole range of commits

In case you don’t want to create additional revert commits but only apply the necessary changes to your working tree, you can use the --no-commit/-n option.

# undo the last commit, but don't create a revert commit

git revert -n HEAD

The manual page at man 1 git-revert list further options and provides some additional examples.

**7. Avoid repeated merge conflicts**

As every developer knows, fixing merge conflicts can be tedious, but solving the exact same conflict repeatedly (e.g. in long running feature branches) is outright annoying. If you’ve suffered from this in the past, you’ll be happy to learn about the underused reuse recorded resolution feature. Add it to your global config to enable it for all projects:

git config --global rerere.enabled true

Alternatively you can enable it on a per-project basis by manually creating the directory .git/rr-cache.

This sure isn’t a feature for everyone, but for people who need it, it can be real time saver. Imagine your team is working on various feature branches at the same time. Now you want to merge all of them together into one testable pre-release branch. As expected, there are several merge conflicts, which you resolve. Unfortunately it turns out that one of the branches isn’t quite there yet, so you decide to un-merge it again. Several days (or weeks) later when the branch is finally ready you merge it again, but thanks to the recorded resolutions, you won’t have to resolve the same merge conflicts again.

The man page (man git-rerere) has more information on further use cases and commands (git rerere status, git rerere diff, etc).

**8. Find the commit that broke something after a merge**

Tracking down the commit that introduced a bug after a big merge can be quite time consuming. Luckily git offers a great binary search facility in the form of git-bisect. First you have to perform the initial setup:

git bisect start # starts the bisecting session

git bisect bad # marks the current revision as bad

git bisect good revision # marks the last known good revision

After this git will automatically checkout a revision halfway between the known “good” and “bad” versions. You can now run your specs again and mark the commit as “good” or “bad” accordingly.

git bisect good # or git bisec bad

This process continues until you get to the commit that introduced the bug.

**9. Avoid common mistakes with git hooks**

Some mistakes happen repeatedly, but would be easy to avoid by running certain checks or cleanup tasks at a defined stage of the git workflow. This is exactly the scenario that hooks were designed for. To create a new hook, add an executable file to .git/hooks. The name of the script has to correspond to one of the available hooks, a full list of which is available in the manual page (man githooks). You can also define global hooks to use in all your projects by creating a template directory that git will use when initializing a new repository (see man git-init for further information). Here’s how the relevant entry in ~/.gitconfig and an example template directory look like:

[init]

templatedir = ~/.git\_template

→ tree .git\_template

.git\_template

└── hooks

└── pre-commit

When you initialize a new repository, files in the template directory will be copied to the corresponding location in your project’s .git directory.

What follows is a slightly contrived example commit-msg hook, which will ensure that every commit message references a ticket number like “#123“.

ruby

#!/usr/bin/env ruby

message = File.read(ARGV[0])

unless message =~ /\s\*#\d+/

puts "[POLICY] Your message did not reference a ticket."

exit 1

end

**10. When all else fails**

So far we covered quite a lot of ground on how to fix common errors when working with git. Most of them have easy enough solutions, however there are times when one has to get out the big guns and rewrite the history of an entire branch. One common use case for this is removing sensitive data (e.g. login credentials for production systems) that were committed to a public repository:

git filter-branch --force --index-filter \

'git rm --cached --ignore-unmatch secrets.txt' \

--prune-empty --tag-name-filter cat -- --all

This will remove the file secrets.txt from every branch and tag. It will also remove any commits that would be empty as a result of the above operation. Keep in mind that this will rewrite your project’s entire history, which can be very disruptive in a distributed workflow. Also while the file in question has now been removed, the credentials it contained should still be considered compromised!