**GIT BASICS**

**Version Control System**

* Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later.
* It allows you to revert selected files back to a previous state, revert the entire project back to a previous state, compare changes over time, see who last modified something that might be causing a problem, who introduced an issue and when, and more
* **Centralized Version Control Systems [CVS, Subversion, and Perforce]**
* **Distributed Version Control Systems**

**Centralized Version Control Systems [CVS, Subversion, and Perforce]**



These systems (such as CVS, Subversion, and Perforce) have a single server that contains all the versioned files, and a number of clients that check out files from that central place.

Centralized collaboration, so everyone knows to a certain degree what everyone else on the project is doing.

Administrators have fine-grained control over who can do what, and it’s far easier to administer a CVCS than it is to deal with local databases on every client.

**Downside**

* Single point of failure that the centralized server represents. If that server goes down for an hour, then during that hour nobody can collaborate at all or save versioned changes to anything they’re working on.
* If the hard disk the central database is on becomes corrupted, and proper backups haven’t been kept, you lose absolutely everything — the entire history of the project except whatever single snapshots people happen to have on their local machines.

**Distributed Version Control Systems [Git, Mercurial,Bazaar or Darcs]**



* This is where Distributed Version Control Systems (DVCSs) step in. In a DVCS (such as Git, Mercurial,Bazaar or Darcs), clients don’t just check out the latest snapshot of the files; rather, they fully mirror the repository, including its full history.
* So Every clone is really a full backup of all the data.
* This allows you to set up several types of workflows that aren’t possible in centralized systems, such as hierarchical models.

**What is Git?**

**Snapshots, Not Differences**

* The major difference between Git and any other VCS (Subversion and friends included) is the way Git thinks about its data. Conceptually, most other systems store information as a list of file-based changes.
* With Git, every time you commit, or save the state of your project, Git basically takes a picture of what all your files look like at that moment and stores a reference to that snapshot.
* To be efficient, if files have not changed, Git doesn’t store the file again,
* just a link to the previous identical file it has already stored.

**Nearly Every Operation Is Local**

* With git each user has entire history of the project stored locally. Hence making access to file history is extremely fast and allows full functionality even when n/w disconnected. It also means every user has a backup.
* Here most of the operations are performed locally in git, that makes git mush faster compare to version control systems.

**Git Has Integrity**

* Everything in Git is checksummed before it is stored and is then referred to by that checksum.
* This means it’s impossible to change the contents of any file or directory without Git knowing about it.
* Git uses SHA-1 hash, it’s a 40-character string composed of hexadecimal characters (0–9 and a–f) and calculated based on the contents of a file or directory structure in Git.

**The Three States**

* **Modified** means that you have changed the file but have not committed it to your database yet.
* **Staged** means that you have marked a modified file in its current version to go into your next commit snapshot.
* **Committed** means that the data is safely stored in your local database.

**The basic Git workflow goes something like this**:

* You modify files in your working tree.
* You selectively stage just those changes you want to be part of your next commit, which adds *only* those changes to the staging area.
* You do a commit, which takes the files as they are in the staging area and stores that snapshot permanently to your Git directory.

**Git Installation**

* $ sudo dnf install git-all
* $ git --version

**First-Time Git Setup**

You should have to do these things only once on any given computer; they’ll stick

around between upgrades.

#git config --system /etc/gitconfig [System specific]

#git config --global ~/.gitconfig [user home directory specification]

#git config --local .gitconfig [current repo]

**Your Identity**

$ git config --global user.name "John Doe"

$ git config --global user.email johndoe@example.com

**Your Editor**

$ git config --global core.editor vim/nano/notepad++

**Checking Your Settings**

$ git config --list

$ git config user.name

**GIT BASICS**

**Getting a Git Repository**

You typically obtain a Git repository in one of two ways:

* You can take a local directory that is currently not under version control, and turn it into a Git repository, or
* You can clone an existing Git repository from elsewhere.

In either case, you end up with a Git repository on your local machine, ready for work.

**Initializing a Git Repository in an Existing Directory**

* $ cd /home/user/my\_project
* $ git init

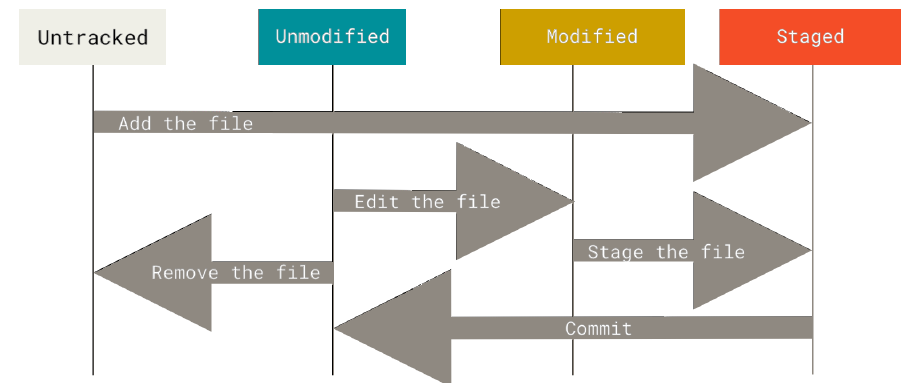
This creates a new subdirectory named .git that contains all of your necessary repository files.

**Cloning an Existing Repository**

* git clone <https://github.com/fvenkat/ReactjsPhotoGallery.git>
  + - * creates ReactjsPhotoGallery dir and it clones
* git clone <https://github.com/fvenkat/ReactjsPhotoGallery.git> <Reactrepo>
* creates Reactrepo dir and it clones

**Recording Changes to the Repository**

* Each file in your working directory can be in one of two states: tracked or untracked.
* **Tracked** files are files that were in the last snapshot; they can be unmodified, modified, or staged. In short, tracked files are files that Git knows about.
* **Untracked** files are everything else — any files in your working directory that were not in your last snapshot and are not in your staging area.
* When you first clone a repository, all of your files will be tracked and unmodified because Git just checked them out and you haven’t edited anything



**Git Initialization**

* $git init

**Staging [Tracking new files]**

* $git add . or -A
* $git add <filename>
* $git add -u updated only

**Commiting your changes**

* $git commit –m “<commit msg>” filename or .
* $git commit –a –m “<commit msg>”

[Skips staging for **all modified files**]

**Checking the Status of Your Files**

$ git status

$ git status –s or git status --short

* **M** README
* **MM** Rakefile
* **A** lib/git.rb [New /untracked file are staged]
* **M** lib/simplegit.rb [Modified /Tracked commited files]
* **??** LICENSE.txt [New files/ untracked]

There are two columns to the output — the lefthand column indicates the status of the staging area and the right-hand column indicates the status of the working tree.

**Ignoring Files**

* Generally automatically generated files such as log files or files produced by your build system can be ignore if needed. In such cases, you can create a file listing patterns to match them named **.gitignore**
* # ignore all .a files

**\*.a**

* # but do track lib.a, even though you're ignoring .a files above

**!lib.a**

* # only ignore the TODO file in the current directory, not subdir/TODO

**/TODO**

* # ignore all files in any directory named build

**build/**

* # ignore doc/notes.txt, but not doc/server/arch.txt

**doc/\*.txt**

* # ignore all .pdf files in the doc/ directory and any of its subdirectories

**doc/\*\*/\*.pdf**

**Viewing Your Staged and Unstaged Changes [Git diff]**

If the git status command is too vague for you — you want to know exactly what you changed, not just which files were changed — you can use the git diff command.

git diff shows you the exact lines added and removed — the patch, as it were.

**Working Directory VS Staging**

* $ git diff

**Staging vs Local Repo**

* $ git diff --staged or cached
* $git diff --check [identifies possible whitespace]

**Removing Files**

To remove a file from Git, you have to remove it from your tracked files (more accurately, remove it from your staging area) and then commit.

The git rm command does that, and also removes the file from your working directory so you don’t see it as an untracked file the next time around.

* $git rm -rvf <filename> [Deletes file from staged + working dir] [changes need to be commited]

If you simply remove the file from your working directory, it shows up under the “Changes not staged for commit” (that is, *unstaged*) area of your git status output.

* $git rm --cached <filename> [remove file from staging + keeps it in work dir] changes need to be commited]
* $rm –rvf <filename> [remove file from staging + keeps it in work dir] [Changes yet to be staged] [u has use git add . so That file is deleted frm staging as well]

**Moving Files [Renaming]**

* $ git mv file\_from file\_to

**Viewing the Commit History**

* lists the commits made in that repository [most recent commits show up first]

$ git log

$git log –p [which shows the differenceintroduced in each commit]

$git log –p -2 [shows onle last 2 entry]

$git log --stat [Prints each commit entry]

$git log –stat --pretty [changes log output format]

$git log --oneline

$git log --oneline –decorate --graph –all

$ git log featureA..origin/featureA

[it will print out the history of your commits, showing where your branch pointers are and how your history has diverged]

$ git log --pretty=oneline

$git log --grapgh

$git log --since=2.weeks [until]

**Undoing Things**

**Amend**

* $ git commit --amend [Use to rename the commit message & rename the hash id]

if you commit and then realize you forgot to stage the changes in a file you wanted

to add to this commit, you can do something like this:

* $ git commit -m 'initial commit'
* $ git add forgotten\_file
* $ git commit --amend

You end up with a single commit — the second commit replaces the results of the first

The obvious value to amending commits is to make minor improvements to your

last commit, without cluttering your repository history with commit messages of

the form, “Oops, forgot to add a file” or “Darn, fixing a typo in last commit”.

**Undo changes in Working tree [Unmodifying a Modified File]**

* $git restore <file> or $ git checkout -- <file>

**Unstaging a Staged File**

* $git restore --staged <file> or $git reset HEAD <filename>

**Working with Remotes**

**Showing Your Remotes**

$ git remote

$ git remote –v [with url]

**Inspecting a Remote**

$ git remote show origin

**Adding Remote Repositories**

git clone command implicitly adds the origin remote for you

$git remote add <remote name> <url>

**Renaming and Removing Remote names**

$ git remote rename pb paul

$ git remote remove paul

**Fetching and Pulling from Your Remotes**

$ git fetch <remote name>

git fetch command only downloads the data to your local repository — it doesn’t automatically merge it with any of your work or modify what you’re currently working on. You have to merge it manually into your work when you’re ready.

Once fetch you want to know wt part of fetched work has to be merge into his repo

$git log --no-merges <issue53>..<origin/master>

It tells which commit is not merged yet to your local & then u can merge it locallay.

$ git pull <remote>

git pull command to automatically fetch and then merge that remote branch into your current branch. This may be an easier or more comfortable workflow for 51 you;

By default, the git clone command automatically sets up your local master branch to track the remote master branch (or whatever the default branch is called) on the server you cloned from. Running git pull generally fetches data from the server you originally cloned from and automatically tries to merge it into the code you’re currently working on.

**Pushing to Your Remotes**

$ git push <remote name> <branch name> @@ $ git push origin master

$ git push –u <remote name> <branch name>

-u flag is short for --set-upstream

If you and someone else clone at the same time and they push upstream and then you push upstream, your push will rightly be rejected. You’ll have to fetch their work first and incorporate it into yours before you’ll be allowed to push

**Tagging**

Git has the ability to tag specific points in a repository’s history as being important. Typically, people use this functionality to mark release points (v1.0, v2.0 and so on).

**Listing Your Tags**

$ git tag

$ git tag –l

$ $ git tag -l "v1.8.5\*" [particular version]

**Creating Tags**

Git supports two types of tags: *lightweight* and *annotated*

A lightweight tag is very much like a branch that doesn’t change — it’s just a pointer to a specific commit.

Annotated tags, however, are stored as full objects in the Git database. They’re checksummed; contain the tagger name, email, and date; have a tagging message; and can be signed and verified with GNU Privacy Guard (GPG).

**Annotated Tags**

**$ git tag –a <tag name> -m <tag msg>**

**$** git tag -a v1.4 -m "my version 1.4"

**Lightweight Tags**

$ git tag <tagname>

$ git show <tag name>

**Tagging Later**

**$ git tag –a <tag name> <commit id>**

**Sharing Tags**

By default, the git push command doesn’t transfer tags to remote servers. You will have to explicitly push tags to a shared server after you have created them. This process is just like sharing remote branches — you can run git push origin <tagname>.

$ git push <remote name> <tag name>

$ git push origin v1.5

$ git push origin --tags [push all tags]

**Deleting Tags**

To delete a tag on your local repository, you can use git tag -d <tagname>.

$ git tag -d v1.4-lw

Remote tag delete

$ git push origin --delete <tagname>

**Checking out Tags**

If you want to view the versions of files a tag is pointing to, you can do a git checkout of that tag, although this puts your repository in “detached HEAD” state, which has some ill side effects.

In “detached HEAD” state, if you make changes and then create a commit, the tag will stay the same, but your new commit won’t belong to any branch and will be unreachable, except by the exact commit hash. Thus, if you need to make changes — say you’re fixing a bug on an older version, for instance — you will generally want to create a branch.

$ git checkout -b version2 v2.0.0

Switched to a new branch 'version2'

**Git Aliases**

$ git config --global alias.co checkout

$ git config --global alias.br branch

$ git config --global alias.ci commit

$ git config --global alias.st status

This means that, for example, instead of typing git commit, you just need to type git ci. As you go on using Git, you’ll probably use other commands frequently as well; don’t hesitate to create new aliases.