GIT is a distributed Version Control System (VCS).

# Introduction

* GIT is a VCS used for tracking changes in a file and coordinating those changes among many people.

# Features

Distributed VCS (i.e., repository is centralized, but each user can clone this into his local repo)

Helps in Distributed workflow i.e., nonlinear work.

Snapshot based storage, concept of Branches.

Remote access to server.

Git bash – uses Linux commands for interface

Git GUI – to share, collaborate files.

Merge option, commit, push, pull, fetch easily done.

Uses UNIX like CLI to work with GIT.

# Previous VCS

CVS 🡪 concurrent version system

SVN 🡪 Sub-version

Mercurio (Hg)

# Download GIT

* Download and Install GIT (bash and/or GUI) on Windows (32/64 bit)/Mac/Linux.

<https://git-scm.com/downloads>

* After installation Git bash GUI will be added to start working with git on system.

Bare Repo – are server repos that is used to share changes coming from different users.

Working Repo – where you work, which has .git files etc for tracking changes.

# Configuring GIT

|  |  |
| --- | --- |
| $ git config --global user.name"name" | Set the name you want attached to your commit IDs.  Ex: $ git config --global user.name "John Doe" or  Ex: $ git config --global user.name John Doe |
| $ git config --global user.email "emailaddress" | Sets the email you want attached to your commit IDs.  Ex: $ git config --global user.email “john@example.com” |
| $ git config --global color.ui auto | Enables helpful colorization of command line output |
| $ git config - -global core.editor emacs | Configuring Emacs as default text editor whenever git needs user to type message. If not configured, git uses system’s default editor. |
| $ git config - -global core.editor “path of executable text editor” | Ex: $ git config --global core.editor “C:/ProgramFiles/Notepad++/notepad++.exe” |
| $ git config – - list | * Displays all the configuration of a user on git. |
| $ git - – version | * Displays version of git installed on the machine |

* Git works within a machine/system in which it is installed.
* Always remember, git commands works only with git folder.
* Many users can be created and named accordingly in git only by the admin, thus allowing multiple user access.

# Creating repository

* Repository 🡪 is where all the committed data is stored, it has IDs (Commit ID).
* Working directory 🡪 local directory where you work on, it has no IDs.
* . git 🡪 stores all the metadata (hidden directory, normally not used by the user)
* All the git commands can be used only on a git directory i.e., within git directory created by git init command.

|  |  |
| --- | --- |
| $ git init [name] | Git will create a new directory/repository with [directory name provided] and initialize .git inside this directory.  This (.git) is where all of the git internals are located.  Example: $ git init example1 🡪 creates a git repo  *This example1 folder contains many hidden folders (.git, ., .. etc)* |
| $ mkdir [folder\_name] | Creates a folder/directory.  Example: $ mkdir folder1 |
| $ touch [file name] | Creates an empty file of any format.  Example: $ touch hello.txt 🡪 creates a txt file named hello. |
| $ echo “data”  $ echo “data”>[file name] | Echoes/prints data on terminal.  Example: $ Echo “welcome” 🡪 displayed on terminal  Adds data on the file.  Example: $ Echo “welcome”>hi.txt 🡪 text within “ “ will be added to the file hi.txt |
| $ cat [filename] | Displays content of the file.  $ cat fil11.txt 🡪 displays content of fil11.txt file |
| $ git add [file] | Snapshots the file in preparation for versioning  (Staging/indexing/caching the file to prepare it for commiting).  $ git add file.txt 🡪 stages file.txt  $ git add \*.html 🡪 stage files only with .html extension.  $ git add . 🡪 All untracked files will be staged. |
| $ git status | Displays the status of file (staged, unstaged, committed etc):  Staged files yet to be committed. (staged = added file)  Unstaged/Untracked files yet to be staged/added. |
| $ rm [file name]  $ git rm – - cached [filename]  $ git reset HEAD [file] | Deletes a file.  Example: $ rm hello.txt 🡪 file will be deleted.  Example: $ rm –r folder 🡪 deletes complete folder.  Removes staged file from working tree and/or index.  Example: $ git rm README.md  Removes the file from staging area, working directory (so it cannot be seen under untracked files)  Unstages the file, but preserve its contents.  Ex: $ git reset HEAD file1.txt |
| $ git mv [from] [to]  $ git mv [from.md] [to.txt] | Moving a file  Example 1: $ git mv README.md READ 🡪  file name README.md is changed to READ  Changing format of the file  Example 2: $ git mv README.md README.pdf 🡪  file format also can be changed. |
| $ git diff ;  $ git diff --staged  $ git diff –hard | To check Difference between working directories, staged files and commits.  $ git diff <folder1> <folder2>;   * Shows difference between folder 1 and 2   $ git diff <commitID1> <commitID2>;   * Shows difference between commit IDs.   $ git diff <file1.txt> <commitID2>;   * Shows difference between file1 and commitID2.   Difference between staging area files and most recent committed file. Shows file differences between staging and the last file version.  Discards changes in working directory and staging area.  (Be careful, because the changes would not have been staged/ committed so those changes will be discarded & can never be recovered). |
| $ git commit [./filename] GIT VIM commit message $ git commit –m ”commit message”  $ git commit -a –m "commit msg"  UNDOING A COMMIT  (used to add missed files) | The git commit command takes all the files that have been staged with git add and records a new permanent snapshot in the database and then moves the branch pointer on the current branch up to it.  $ git commit [filename]🡪 if file is staged, enters into VIM screen.  Type commit message then follow below commands.  I 🡪 enters insert mode {type commit message here}  Esc 🡪 to escape from insert mode.  : wq 🡪 saves the message and exits from vim like screen.  Commits the staged files into repository with commit message.  Example: $ git commit –m "adding the file README.md"  Direct commit without staging (i.e., no need to use git add).  Only previously committed files can be committed directly without staging.  Any new file cannot be committed directly without staging.  $ 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. |
| $ git show  $ git show [commit ID] | Shows all the commits of the repository with their commit ID.  Shows the diff between commit mentioned and its parent commit. |
| $ git checkout  [commit ID] | To revert a commit. |
| git fetch | The git fetch command communicates with a remote repository and fetches down all the information that is in that repository that is not in your current one and stores it in your local database.  **$ git pull = $ git fetch+ $ git merge**  $ git fetch origin |
| git merge | The git merge tool is used to merge one or more branches into the branch you have checked out. It will then advance the current branch to the result of the merge.  $ git merge [branch-name]  $ git checkout master 🡪 Switched to branch 'master'  $ git merge branch-1 🡪 Merge made, branch-1 is merged to master. |
| git pull | The git pull command is basically a combination of the git fetch and git merge commands, where Git will fetch from the remote you specify and then immediately try to merge it into the branch you’re on.  $ git pull [url]  Ex: $ git pull <https://github.com/pullingafile/file1> |
| git push | The git push command is used to communicate with another repository, calculate what your local database has that the remote one does not, and then pushes the difference into the other repository. It requires write access to the other repository and so normally is authenticated somehow.  $ git push origin master  $ git push [url] |
| git clone [option]  $ git clone URL  $ git clone local\_path | This Command creates a new directory, goes into it and runs git init i.e creates an empty Git repository, adds a remote (git remote add) to the URL that you pass it (by default named origin), runs a git fetch from that remote repository and then checks out the latest commit into your working directory with git checkout.  $ git clone [url] 🡪 clone from URL  $ git clone <https://github.com/libgit2/libgit2>  $ git clone /libgit2/library\_File 🡪 clone locally.  Thus Clones file (library\_File) from the path mentioned into the current directory where we are working. |
| git archive | The git archive command is used to create an archive file of a specific snapshot of the project. |
| git submodule | The git submodule command is used to manage external repositories within a normal repositories. This could be for libraries or other types of shared resources. The submodule command has several sub-commands (add, update, sync, etc) for managing these resources. |
| git stash | Used to switch files between branches and use the unstaged files present in one branch to use it in another.  Example:   1. Create files (file1, file2, file3) in branch1. 2. $ git stash 🡪 will save the unstaged data of branch1. 3. Modify changes but don’t stage it.   4. Switch to master branch.  5. To work on the files created in b1, git stash apply. |
| . gitignore | Files to be ignored are stored here.  Example:  $ touch .gitignore 🡪 creates a .gitignore file   1. $ Echo “\*.log “>.gitignore 🡪 all .log files are ignored in staging and committing entries. 2. Inside .gitignore write \*.a 🡪 all .a files ignored 3. Inside .gitignore write doc/\*\*/\*.pdf 🡪 ignore all .pdf files in the doc directory (which is a main directory). 4. Inside .gitignore write doc/\*.txt🡪 ignore doc/notes.txt, but not doc/server/arch.txt 5. Inside .gitignore write build/ 🡪 ignore all files in the build/ directory. 6. Inside .gitignore write !lib.a 🡪 but do track lib.a, even though you're ignoring .a files above |
| git help [command] | To access or get data of any command.  Example: $ git help pull 🡪 shows related commands and pull usage. |
| $ git checkout  --<file name> | This discards the changes you’ve made.  It’s important to understand that git checkout -- <file> is a dangerous command. Any changes you made to that file are gone.  $ git checkout - - file; 🡪 changes made on file before committing will be discarded. |
| $ git repo init | Initialize a repository i.e latest repository source into current directory. |
| $ git repo sync | Update working tree with latest repository version. |

# Branches

* Branch is mainly to work on some particular part of project i.e., feature/ path/ bug fix individually than merging it to main branch to add the changes.
* Branch, log, diff option will be available only after commit.

|  |  |
| --- | --- |
| $ git branch [branch-name] | Creates a branch (no spaces in branch name.)  Example: git branch branch1 |
| $ git branch | Displays the name of branches and master.  \* indicates the branch user is currently on. |
| $ git checkout [branch-name] | To switch branch.  Ex: git checkout branch1  🡪 Checks whether you are on branch1, else it switches to branch1. |
| $ git branch -d [branch-name] | Deletes the specified branch name, but the commits of that branch will be present if it was merged with master before deletion. |
| $ git cat [file name] | Concatenate the file and display its contents on terminal.  Example: $ cat page.html: displays the data present in page.html onto the terminal. |
| $ git branch –v | To see last commit on each branch. (-v: verbose) |
| $ git branch –merged | To see which branches are already merged into the branch you’re on. |
| $ git branch --no-merged | To see which branches are not merged into master branch |
| $ git merge [branch1] [branch2] | Combining two commits of branches 1 and 2. |
| $ git merge [branch] | Combines the specified branch’s history into the current branch |

Delete branch only after merging else the data on that branch will be deleted if you delete a branch.

## MERGE CONFLICT

* git merge conflict ERROR is when both master and branch has some changes in one particular file and it cannot be merged.
* Solution 🡪 use mergetool in git, configure it and start using to solve merge conflicts.
* Example of Merge Tools:

opendiff kdiff3 tkdiff xxdiff meld tortoisemerge gvimdiff diffuse ecmerge p4merge araxis bc3 emerge vimdiff.

ERROR MESSAGE:

<<<HEAD ------ code on the branch 1

||||| ----- original code that differs in both branch1 and branch 2

>>>>master--- code on branch 2(or master)

now edit the changes in original file, save, stage it and commit => merge conflict solved.

# Log

Shows all the logs of Commit made by the users.

Best simplified log output command: $ git log --oneline --decorate --graph -–all

|  |  |
| --- | --- |
| Command | Description |
| $ git log | Shows up commit history in reverse chronological order (latest first).  Commit history includes commit ID, author, date, commit message. |
| $ git log -p  $ git log --stat  $ git log --shortstat  $ git log --name-only  $ git log --name-status  $ git log --abbrev-commit  $ git log --relative-date  $ git log --graph | -p shows the difference introduced in each commit.  Shows statistics for files modified in each commit.  Display only the changed/insertions/deletions line from the --stat command.  Shows list of files modified after the commit information.  Shows list of files affected with added/modified/deleted info as well.  Shows only first few characters of the SHA-1 checksum instead of all 40.  Display the date in a relative format (for example, “2 weeks ago”) instead of using the full date format.  Display an ASCII graph of branch and merge history beside the log output. |
| $ git log -n  $ git log -2  $ git log -p -4 | Shows only the last n commits  Example: -2 limits the output to only the last two entries.  Commands can be clubbed. |
| $ git log –S  $ git log –-since  $ git log --after  $ git log --until  $ git log --before  $ git log --author  $ git log --committer  $ git log --grep | Only show commits adding or removing code matching the string.  Limit the commits to those made since/after the specified date.  $ git log --since=2.weeks;  $ git log --after=2008-01-15;  $ git log --until=2 years 1 day 3 minutes ago.  Only show commits in which the author entry matches the specified string.  $ git log --author=Mary; (case sensitive string)  Only show commits in which the committer entry matches the specified  String.  Only show commits with a commit message containing the string. |
| $ git log --pretty=[option] | --pretty option changes the log output to formats other than the default.  It includes below options:   * “Online” 🡪 prints each commit on a single line. * “short” 🡪 Short commit information * “full” 🡪 more commit information * “fuller” 🡪 complete commit information. |
| Common Git log errors | * Fatal bad default HEAD 🡪 no commit is made. * HEAD 🡪 points to the branch we are on. * Detached HEAD state 🡪 you are not on master branch or you are looking at a commit that is not labelled with a branch. |

# Tag

Ability to tag specific points in history as being important.

Example: commits, release versions, important branches etc.

|  |  |
| --- | --- |
| To Display TAGS | |
| $ git tag  $ git tag --list | **Listing the available tags in Git in alphabetical order.** |
| $ git tag -l [pattern]  $ git tag -l "string"  $ git tag -l "string" | Search for tags with the exact pattern.  Example: $ git tag -l “V1.2” 🡪 displays only V1.2 tag if present.  Search for tags with particular pattern.  Example: $ git tag -l "v1\*" 🡪 displays V1, V1.2, V13, V1w etc tags |
| To Create TAGS | |
| Lightweight tag  $ git tag v1.4-lw | LW tags are stored as checksum (just bit entry) in git database.  -lw creates a lightweight tag with version v1.4 |
| Annotated tag  $ git tag -a v1.4  $ git tag –a v1 –m “hi” | These are stored as full objects in the Git database.  They’re check summed, contain the tagger name, email, date, and have a tagging message, and can be signed and verified with GNU Privacy Guard.  **Example:**  -a creates annotated tag with value v1.4  -m adds message to the tag. |
| $ git show [tag name] | Gives entire detail of tag.  Example: $ git show v1.2 🡪 shows tag details of tag v1.2 |
| Tag a commit | |
| $ git tag –a v1 [commit ID] | Example: if commit ID = 9014254162hgy26172 “version 1”   * Git tag –a v1 9014254 ; marks tag on the above commit. |

## Tagging options

|  |  |
| --- | --- |
| -a, --annotate | annotated tag, needs a message |
| - m, --message <message> | tag message  $ git tag v1-lw –m “this is lightweight tag” |
| -F, --file <file> | read message from file |
| -v, --verify | verify tags  $ git tag –v V1 |
| -s, --sign | annotated and GPG-signed tag |
| -n[<n>] | Print <n> lines of each tag message.  $ git tag –l –n3 🡪 prints 3 lines of all listed tags in project. |
| -d, --delete | delete tags  $ git tag –d v1 🡪 deletes v1 tag |
| --cleanup <mode> | how to strip spaces and #comments from message |
| -u, --local-user <key-id> | use another key to sign the tag |
| -f, --force | replace the tag if exists |
| --contains <commit> | Print only tags that contain the commit. |

# Adding Remote Repositories

|  |  |
| --- | --- |
| $ git remote | Lists the short names of each remote handle you’ve specified.  If you’ve cloned your repository, you should at least see origin – that is the default name Git gives to the server you cloned from. |
| $ git remote add <remotename> <url> | To add a new remote Git repository as a shortname you can reference easily.  Example: $ git remote add ticgit https://github.com/paulboone/ticgit |
| $ git fetch [remote-name] | The command goes out to that remote project and pulls down all the data from that remote project that you don’t have.  Example: git fetch pb |
| $ git remote -v | Shows the URLs that Git has stored for the shortname to be used when reading and writing to that remote. |
| Git FETCH <remote name> | Downloads the data to your local repository. (so have to merge manually) |
| Git PULL <remote name> | Automatically fetch and then merge that remote branch into your current branch. |
| Git CLONE <remote name> | Automatically sets up your local master branch to track the remote master branch |
| $ git push [remote-name] [branch-name] | To push your repository data to a remote repository.   * This command works only if you cloned from a server to which you have write access and if nobody has pushed in the meantime. * 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   Example: git push newproject branch1. |
| $ git remote rename <old remote name> <new remote name>: | To rename a remote repo. |
| $ git remote rm <remote name>: | To remove a remote repo. |

# ALIASES

Git helps to set up an alias for each command using git config.

|  |  |
| --- | --- |
| $ git config --global alias.[short] [full] | instead of using full command name, type short name. |
| $ git config –global alias.b branch | $ git branch = $ git b |
| $ git config –global alias.st status | $ git branch = $ git b |
| $ git config --global alias.unstage 'reset HEAD --' | This makes the following two commands equivalent:  $ git unstage fileA  $ git reset HEAD – fileA |
| $ git config --global alias.last 'log -1 HEAD' | $ git last = $ git log -1 HEAD  🡪 display last commit |

# Debug with GIT

[**https://git-scm.com/book/en/v2/Git-Tools-Debugging-with-Git**](https://git-scm.com/book/en/v2/Git-Tools-Debugging-with-Git)

|  |  |
| --- | --- |
| $ git blame [filename] | Displays commitID, committer, Commitdate, number of lines changed/added, lines/changes added in file.  Example1: $ git blame you.txt   * 8609102 (Supriya 2018-04-05 11:11:10 1) hi 🡪 output   Example2: $ git blame –L 2,9 you.txt   * Limits o/p of file between lines 2 to 9. |
| $ git bisect | Command does a binary search through your commit history to help you identify as quickly as possible which commit introduced an issue.  $ git bisect start 🡪 to start debugging  $ git bisect bad 🡪 to indicate current commit has bugs  $ git bisect good [commit ID] 🡪 insert commit ID which was bug free. |

# GITHUB

* Github: only commits are stored in your github repository.
* To add data to github: remote push from your local repository.
* To fetch to working directory: remote pull from github repo.

# **DATA TRANSFER**

Git can use FOUR major protocols to share data between users.

1. Local
2. HTTP
3. Secure Shell (SSH)
4. Git

ACCESS: <https://git-scm.com/book/en/v2/Git-on-the-Server-The-Protocols>

## **Local Protocol**

* In this the remote repository to be fetched/pulled is in another directory on same machine.
* This is often used if everyone on your team has access to a shared filesystem like a single linux server hosting many users for building code or integration.
* To add a local repository to an existing Git project, you can run this:

$ git remote add local\_proj /opt/git/project.git

* To clone a local repository into your directory, you can run this:

$ git clone /opt/git/project.git Or $ git clone file:///opt/git/project.git

The main reason to specify the file:// prefix is if you want a clean copy of the repository with extraneous references or objects left out – generally after an import from another VCS.

## **HTTP**

Git can communicate over HTTP in two different modes:

### **Smart HTTP** – newer version

* Introduced after Git 1.6.6
* It’s simpler for the user and smarter about how it communicates.
* The “smart” HTTP protocol operates similar to the SSH or Git protocols but runs over standard HTTP/S ports and can use various HTTP authentication mechanisms.
* Thus there will be a single URL for all type of access, having server to prompt for username/password only if required (i.e. no need to generate SSH keys for SSH type git access)

Ex: Clone: $ git clone <https://example.com/gitproject.git>

### **Dumb HTTP** - older version just for information

* Prior to Git 1.6.6,
* Very simple and generally read-only.
* If the server does not respond with a Git smart HTTP service, the Git client will try to fall back to the simpler “dumb” HTTP protocol.

## **SECURED SHELL (SSH)**

A common transport protocol for Git when self-hosting is over SSH.

* SSH is also an authenticated network protocol; and because it’s ubiquitous, it’s generally easy to set up and use.
* access over SSH is secure – all data transfer is encrypted and authenticated
* SSH is efficient, making the data as compact as possible before transferring it.
* People must have access to your machine over SSH to access it, even in a read-only capacity, which doesn’t make SSH access conducive to open source projects.

Ex: Clone: $ git clone ssh://[user@]server/project.git

## **GIT PROTOCOL**

This is a special daemon that comes packaged with Git; it listens on a dedicated port (9418) that provides a service similar to the SSH protocol, but with absolutely no authentication.

* Either the Git repository is available for everyone to clone or it isn’t.
* No security, no restriction in push access to your project.
* The Git protocol is often the fastest network transfer protocol available. If you’re serving a lot of traffic for a public project or serving a very large

# **Set up GIT on server**

<https://git-scm.com/book/en/v2/Git-on-the-Server-Getting-Git-on-a-Server>

# **GIT COMMANDS (stepwise to commit on to build machine)**

# To checkout Entire Source code.

repo init -u ssh://<Git\_USERNAME>@10.66.10.95:29418/R4GDevices/RJIL/tools/manifest.git -b dev\_stb\_apps

repo sync -j4 -f

repo forall -c "pwd ; scp -p -P 29418 10.66.10.95:hooks/commit-msg .git/hooks/ ; git config remote.origin.push refs/heads/\*:refs/for/\* "

repo forall -c "pwd ; git checkout -b local-vendor\_release HEAD"

# To clone and checkout particular Module/Git

git clone ssh://<Git\_USER\_NAME>@10.66.10.95:29418/R4GDevices/RJIL/packages/apps/MobileRemote.git

git checkout -b local\_dev\_stb\_apps remotes/origin/dev\_stb\_apps

# To commit changes on

git add <File\_Name>

git commit -a

# To push changes to git via gerrit.

git push origin local\_dev\_stb\_apps:refs/for/dev\_stb\_apps

git push origin local:refs/for/dev\_stb\_apps

git push origin local:refs/for/3327