git

Table of Contents

1. **GIT** 
   1. History of GIT
   2. What is GIT?
2. **About version control** 
   1. Centralized Version Control
   2. Distributed Version Control
3. **GIT Setup**
4. **GIT Basics**
   1. Getting a Git Repository
   2. Initialise a repository
   3. Cloning a Existing Repository
   4. Recording changes to your Repository
   5. Checking the status of your file
   6. Tracking new files
   7. Ignoring files
   8. Viewing your Staged Changes
   9. Committing you changes
   10. Skipping the Staged area
   11. Removing files
   12. Moving files
   13. Viewing the Commit History
5. **Branching Strategy**
   1. Selecting a Branching Strategy
6. **GIT FLOW**
   1. GITHUB Flow
   2. Creating Remote Branches
   3. Delete Branches
   4. Switch Branches
   5. Merging Branches
   6. GIT Merge Conflict
   7. Pushing changes to GITHUB
7. **Creating a Snapshot**

**1. GIT:**

**1.1. History of GIT:**

The Linux kernel is an open-source software project of large scope. During the early years of the Linux kernel maintenance (1991–2002), changes to the software were passed around as patches and archived files. In 2002, the Linux kernel project began using a proprietary DVCS called Bit Keeper.

In 2005, the relationship between the community that developed the Linux kernel and the commercial company that developed Bit Keeper broke down, and the tool’s free-of-charge status was revoked. This prompted the Linux development community to develop their own tool based on some of the lessons they learned while using Bit Keeper. Some of the goals of the new system were as follows:

Speed

Simple design

Strong support for non-linear development (thousands of parallel branches)

Fully distributed

Able to handle large projects like the Linux kernel efficiently

* 1. **What is GIT:**
* Git is a version control system used for tracking changes in computer files. It is generally used for source code management in software development.
* Git is used to tracking changes in the source code
* The distributed version control tool is used for source code management
* It allows multiple developers to work together
* It supports non-linear development through its thousands of parallel branches.

1. **About Version Control:**

* 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.
* Version Control System (VCS) is a very wise thing to use.
* 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.
  1. **Centralized Version Control:**

**Centralized Version Control** is a version control system using server/client model and server contains all the history of source code.

Repository

push pull

Working Copy

Working Copy

Working Copy

* 1. **Distributed Version Control:**

**Distributed Version Control** is a version control where each client can have same copy of source code as server has and both server and client maintain history of source code.

|  |  |  |  |
| --- | --- | --- | --- |
| Sr. No. | Key | Centralized Version Control | Distributed Version Control |
| 1 | Working | In CVS, a client need to get local copy of source from server, do the changes and commit those changes to central source on server. | In DVS, each client can have a local branch as well and have a complete history on it. Client needs to push the changes to branch which will then be pushed to server repository. |
| 2 | Learning Curve | CVS systems are easy to learn and set up. | DVS systems are difficult for beginners. Multiple commands need to be remembered. |
| 3 | Branches | Working on branches in difficult in CVS. Developer often faces merge conflicts. | Working on branches in easier in DVS. Developer faces lesser conflicts. |
| 4 | Offline Access | CVS system do not provide offline access. | DVD systems are workable offline as a client copies the entire repository on their local machine. |
| 5 | Speed | CVS is slower as every command need to communicate with server. | DVS is faster as mostly user deals with local copy without hitting server every time. |
| 6 | Backup | If CVS Server is down, developers cannot work. | If DVS server is down, developer can work using their local copies. |

Install GIT on Linux:

1.From your shell, install Git using apt-get:

$ sudo apt-get update

$ sudo apt-get install git

2.Verify the installation was successful by typing git --version:

$ git –version

$ git version 2.34.1

1. **Git setup:**

Get a GITHUB account.

Download and install [git](https://git-scm.com/downloads).

Set up git with your user’s name and email

Open a terminal/shell and type:

$ git config --global user.name "Your name here"

$ git config --global user.email [your\_email@example.com](mailto:your_email@example.com)

Copy your public key (the contents of the newly-created id\_rsa.pub file) into your clipboard. On a Mac, in the terminal/shell, type:

$ pbcopy < ~/.ssh/id\_rsa.pub

Paste your ssh public key into your GitHub account settings.

Go to your GitHub [Account Settings](https://github.com/settings/profile)

Click “[SSH Keys](https://github.com/settings/ssh)” on the left.

Click “Add SSH Key” on the right.

Add a label (like “My laptop”) and paste the public key into the big text box.

In a terminal/shell, type the following to test it:

$ ssh -T [git@github.com](mailto:git@github.com)

If it says something like the following, it worked:

* + - Hi username! You have successfully authenticated,
    - but GitHub does not provide shell access.

1. **GIT Basics:**

**Working Directory and Staging Area or Index:**

The working directory is the place where files are checked out. In other CVCS, developers generally make modifications and commit their changes directly to the repository.

But Git uses a different strategy. Git does not track each and every modified file. Whenever you do commit an operation, Git looks for the files present in the staging area. Only those files present in the staging area are considered for commit and not all the modified files.

Step 1: You modify a file from the working directory.

Step 2: You add these files to the staging area.

Step 3: You perform commit operation that moves the files from the

staging area.

After push operation, it stores the changes permanently to the Git

repository.

Working directory

Git add operation

Staging area

Git commit operation

Git repository

* 1. **Getting a Git Repository:**

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

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

Use a command **cd** to get into the repository then you must initialise the git command,

**git init**

* 1. **Cloning an Existing Repository:**

You clone a repository with git clone <URL>. For example, if you want to clone the Git linkable library called name2

$ git clone https://github.com/name1/name2

That creates a directory named name2, initializes a  .git directory inside it, pulls down all the data for that repository, and checks out a working copy of the latest version.

If you go into the new name2 directory that was just created, you will see the project files in there, ready to be worked on or used.

If you want to clone the repository into a directory named something other than name2, you can specify the new directory name as an additional argument:

$ git clone <https://github.com/name1/name2> name

* 1. **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, as well as any newly staged files; 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 your files will be tracked and unmodified because Git just checked them out and you haven’t edited anything.
  1. **Checking the Status of Your Files:**

The main tool you use to determine which files are in which state is the git status command,

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

nothing to commit, working tree clean

If you add a new file to your project, a simple README file. If the file did not exist before, and you run git status, you see your untracked file like so:

$ echo 'My Project' > README

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Untracked files:

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

README

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

You can see that your new README file is untracked, because it is under the “Untracked files” heading in your status output.

* 1. **Tracking new files:**

Tracking a new file, you use the command git add. To begin tracking the

README file, you can run this:

$ git add README

Now status command again, you can see that your README file is now tracked and staged to be committed:

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes to be committed:

(use "git restore --staged <file>..." to unstage)

new file: README

You can tell that it is staged because it’s under the “Changes to be committed” heading.

* 1. **Ignoring Files:**

You will have a class of files that you don’t want Git to automatically add or even show you as being untracked.

These are generally automatically generated files such as log files or files produced by your build system.

In such cases, you can create a file listing patterns to match them named .gitignore.

$ cat .gitignore

\*.[oa]

\*~

The first line tells Git to ignore any files ending in “.o” or “.a” — object and archive files that may be the product of building your code.

The second line tells Git to ignore all files whose names end with a tilde (~), which is used by many text editors such as Emacs to mark temporary files.

The rules for the patterns you can put in the .gitignore file are as follows:

* Blank lines or lines starting with # are ignored.
* Standard glob patterns work, and will be applied recursively throughout the entire working tree.
* You can start patterns with a forward slash (/) to avoid recursivity.
* You can end patterns with a forward slash (/) to specify a directory.
* You can negate a pattern by starting it with an exclamation point (!).
  1. **Viewing Your Staged and Unstaged Changes:**

If you want to see what you have staged that will go into your next commit, you can use git diff --staged. This command compares your staged changes to your last commit:

$ git diff --staged

diff --git a/README b/README

new file mode 100644

index 0000000..03902a1

--- /dev/null

+++ b/README

@@ -0,0 +1 @@

* 1. **Committing your changes:**

Now that your staging area is set up the way you want it, you can commit your changes. You ran git status, you saw that everything was staged, so you’re ready to commit your changes.

$ git commit

* 1. **Skipping the staging area:**

If you want to skip the staging area, Git provides a simple shortcut. Adding the -a option to the git commit command makes Git automatically stage every file that is already tracked before doing the commit, letting you skip the git add part:

$ git commit -a -m 'Add new benchmarks'

[master 83e38c7] Add new benchmarks

1 file changed, 5 insertions(+), 0 deletions(-)

* 1. **Removing files:**

To remove a file from Git, you must 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 \\*~

This command removes all files whose names end with a ~.

* 1. **Moving files:**

Git has a**mv**command. If you want to rename a file in Git, you can run

$ git mv file\_from file\_to

$ git mv README.md README

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

renamed: README.md -> README

git mv command:

$ mv README.md README

$ git rm README.md

$ git add README

* 1. **Viewing the commit history:**

The most basic and powerful tool to do this is the git log command.

$ git clone https://github.com/sanjayvg0612/docker-nginx.git

When you run **git log** in this project, you should get output that looks something like this:

$ git log

commit a8ce3e1e026fea7c60377576a06878ed5a12106a (HEAD -> master)

Author: sanjayvg0612 <[sanjaysanjay80609@gmail.com](mailto:sanjaysanjay80609@gmail.com)>

Date: Wed Apr 26 07:05:16 2023 +0530

| **Specifier** | **Description of Output** |
| --- | --- |
| %H | Commit hash |
| %h | Abbreviated commit hash |
| %T | Tree hash |
| %t | Abbreviated tree hash |
| %P | Parent hashes |
| %p | Abbreviated parent hashes |
| %an | Author name |
| %ae | Author email |
| %ad | Author date (format respects the --date=option) |
| %ar | Author date, relative |
| %cn | Committer name |
| %ce | Committer email |
| %cd | Committer date |
| %cr | Committer date, relative |
| %S | Subject |

**Common options to git log:**

|  | |
| --- | --- |
| **Option** | **Description** |
| -p | Show the patch introduced with each commit. |
| --stat | Show statistics for files modified in each commit. |
| --shortstat | Display only the changed/insertions/deletions line from the --stat command. |
| --name-only | Show the list of files modified after the commit information. |
| --name-status | Show the list of files affected with added/modified/deleted information as well. |
| --abbrev-commit | Show only the first few characters of the SHA-1 checksum instead of all 40. |
| --relative-date | Display the date in a relative format (for example, “2 weeks ago”) instead of using the full date format. |
| --graph | Display an ASCII graph of the branch and merge history beside the log output. |
| --pretty | Show commits in an alternate format. Option values include oneline, short, full, fuller, and format (where you specify your own format). |
| --oneline | Shorthand for --pretty=oneline --abbrev-commit used together. |

**Options to limit the output of git log:**

|  | |
| --- | --- |
| **Option** | **Description** |
| -<n> | Show only the last n commits. |
| --since, --after | Limit the commits to those made after the specified date. |
| --until, --before | Limit the commits to those made before the specified date. |
| --author | Only show commits in which the author entry matches the specified string. |
| --committer | Only show commits in which the committer entry matches the specified string. |
| --grep | Only show commits with a commit message containing the string. |
| -s | Only show commits adding or removing code matching the string. |

1. **Branching strategy:**

A branching strategy is something a software development team uses when interacting with a version control system for writing and managing code. As the name suggests, the branching strategy focuses on how branches are used in the development process.

One major purpose of a version control system is to enable a collaborative development environment without overlapping or affecting the codebase. There, each team member modifying the same source code will inevitably be making conflicting code changes.

However, we can avoid such conflicts with a version control system by using branches when writing and merging code to a master branch to create the product.

* 1. **Selecting a branching strategy:**

The selection process for a branching strategy depends entirely on the users and the project requirements. Factors like the development method, scale, user preferences highly impact this selection.

Branching strategies that do not align or make it more difficult to implement Continuous Integration and Continuous Delivery in DevOps pipelines should not be used in a DevOps environment.

A good branching strategy should have the following characteristics:

Provides a clear path for the development process from initial changes to production.

Allows users to create workflows that lead to structured releases.

Enables parallel development Optimizes developer workflow without adding any overhead.

Enables faster release cycles.

Efficiently integrates with all DevOps practices and tools such as different version control systems.

1. **Git Flow:**

Git Flow is the most widely known branching strategy that takes a multi-branch approach to manage the source code.

This approach consists of two main branches that live throughout the development lifecycle.

Primary Branches:

Master: The primary branch where all the production code is stored. Once the code in the “develop” branch is ready to be released, the changes are merged to the master branch and used in the deployment.

Develop: This is where all the actual development happens. All the pre-production code is stored here, and the completed code of all the supporting branches is merged directly to the develop branch.

Support Branches:

During the development, developers create different branches for specific use cases using the develop branch as the base. The following are some branches created like that:

Feature: Feature branches are used to develop new features and branches off exclusively from the develop branch.

Hotfix: This is to deal with production issues where quick fixes are required. They can branch off from the master itself, but need to be merged to both master and develop branches.

Release: This branch is used to aggregate fixes and improvements and prepare for the production release. It will be branched from the develop branch and merged to both develop and master.

* 1. **GitHub Flow:**

As the name suggests, this strategy was introduced by GitHub, aiming to provide a simple and lightweight approach to manage the development. It adheres to the following guidelines when managing the source control with a single primary branch.

Master:

The primary branch where code is branched off from and merged to. Anything in the master branch is deployable.

Any change (feature/bug) is made in a new branch derived from the master with a descriptive branch name describing the development.

Commit to the development branch locally and regularly push to the branch.

Create a pull request once the development is done so that the code can be reviewed.

Once the code is reviewed and approved, it must be tested in the branch before merging to the master branch.

From this point, users can immediately deploy the master branch with the new changes.

git branch

List all the branches in your repository. This is synonymous with git branch --list.

git branch <branch>

Create a new branch called ＜branch＞. This does *not* check out the new branch.

git branch -d <branch>

Delete the specified branch. This is a “safe” operation in that Git prevents you from deleting the branch if it has unmerged changes.

git branch -D <branch>

Force deletes the specified branch, even if it has unmerged changes. This is the command to use if you want to permanently throw away all of the commits associated with a particular line of development.

git branch -m <branch>

Rename the current branch to ＜branch＞.

git branch -a

List all remote branches.

* 1. **Creating remote branches:**

The git branch command not only operates on the local branches but also remote branches. Creating a branch on a remote repository must be configured and added to the local repository config:

git remote add new-remote-repo <Repo URL>

git push <new-remote-repo> test\_branch

To update the local repository (pull changes):

>>> git pull

To update the remote repository (push changes):

>>> git push origin master

When pushing the first time, do

>>> git push -u origin master

# -u tells the remote to track this branch in the future

A quick word on origin and master : these are the default names of the remote repository and the first branch.

* 1. **Deleting Branches:**

After finishing the work on a branch and merging it into the main master, you can delete it:

Git branch -d test\_branch

If unmerged changes remain, an error will occur:

Error: The branch 'test\_branch' is not fully merged.

if you are sure you want to delete it, run 'git branch -D test\_branch'.

If you are sure that you want to delete the branch permanently you can use the git branch command with the capital -D option:

* 1. **Switching Branches:**

Switching branches is a straightforward operation. Executing the following will point HEAD to the tip of ＜branchname＞.

git checkout ＜branchname＞

Git tracks a history of checkout operations in the reflog.

You can execute git reflog to view the history.

* 1. **Merging Branches:**

In Git, the merging is a procedure to connect the forked history. It joins two or more development history together. The git merge command facilitates you to take the data created by git branch and integrate them into a single branch. Git merge will associate a series of commits into one unified history. Generally, git merge is used to combine two branches.

Commit 2

Commit 1

Commit 3

Dev branch

To merge a specified commit into master, first discover its commit id. Use the log command to find the commit id.

$git log

To merge the commits into the master branch, switch over to the master branch.

$ git checkout master

Now, Switch to branch 'master' to perform merging operation on a commit. Use the git merge command along with master branch name.

$ git merge master

* 1. **Git Merge Conflict:**

When two branches are trying to merge, and both are edited at the same time and in the same file, Git will not be able to identify which version is to take for changes. Such a situation is called merge conflict.

If such a situation occurs, it stops just before the merge commit so that you can resolve the conflicts manually.

Remote Repository

User B

User A

Pull request

Pull request

1

2

3

4

5

6

7

4

5

6

7

1

2

3

4

* 1. **Push changes to GitHub:**

To add the changes to your git repo files on your computer to the version of your repository on GitHub, you need to **push** them GitHub.

You can push your changes to GitHub with:

$ git push origin master

1. **Creating Snapshot:**

**Initializing a repository**

git init

**Staging files:**

git add file1.js # Stages a single file

git add file1.js file2.js # Stages multiple files

git add \*.js # Stages with a pattern

git add . # Stages the current directory and all its content

**Viewing the status:**

git status # Short status

git status -s # Full status

**Committing the staged files:**

git commit -m “Message” # Commits with a one-line message

git commit # Opens the default editor to type a long message

Skipping the staging area

git commit -am “Message”

**Removing files:**

git rm file1.js # Removes from working directory and staging area

git rm --cached file1.js # Removes from staging area only

Renaming or moving files

git mv file1.js file1.txt

**Viewing the staged/unstaged changes:**

git diff # Shows unstaged changes

git diff --staged # Shows staged changes

git diff --cached # Same as the above

**Viewing the history:**

git log # Full history

git log --oneline # Summary

git log –reverse # Lists the commits from the oldest to the newest

**Browsing History:**

**Viewing the history:**

git log --stat # Shows the list of modified files

git log --patch # Shows the actual changes (patches)

**Comparing commits:**

git diff HEAD~2 HEAD # Shows the changes between two commits

git diff HEAD~2 HEAD file.txt # Changes to file.txt only

**Checking out a commit:**

git checkout dad47ed # Checks out the given commit

git checkout master # Checks out the master branch

**Branching & Merging:**

**Managing branches:**

git branch bugfix # Creates a new branch called bugfix

git checkout bugfix # Switches to the bugfix branch

git switch bugfix # Same as the above

git switch -C bugfix # Creates and switches

git branch -d bugfix # Deletes the bugfix branch

**Merging:**

git merge bugfix # Merges the bugfix branch into the current branch

git merge --no-ff bugfix # Creates a merge commit even if FF is possible

git merge --squash bugfix # Performs a squash merge

git merge –abort # Aborts the merge

**Viewing the merged branches:**

git branch --merged # Shows the merged branches

git branch --no-merged # Shows the unmerged branches

**Collaboration**

**Cloning a repository**

git clone URL

**Syncing with remotes:**

git fetch origin master # Fetches master from origin

git fetch origin # Fetches all objects from origin

git fetch # Shortcut for “git fetch origin”

git pull # Fetch + merge

git push origin master # Pushes master to origin

git push # Shortcut for “git push origin master”

**Sharing branches:**

git branch -r # Shows remote tracking branches

git branch -vv # Shows local & remote tracking branches

git push -u origin bugfix # Pushes bugfix to origin

git push -d origin bugfix # Removes bugfix from origin

**Managing remotes:**

git remote # Shows remote repos

git remote add upstream url # Adds a new remote called upstream

git remote rm upstream # Remotes upstream

**Amending the last commit:**

git commit --amend