PUBLISHED BY MKPROJECTS

MKPROJ.COM

What is Git? Git is a version control program that is used in many organizations to help organize, track changes & control various versions of a project. As said before, a version control program essentially tracks the changes made in files in a project. It is a very important skill to learn, and an excellent tool to use in your own personal projects, and hey it's how the MKProjects operates!!!

Git was introduced at April 7th, 2005 by the founder of the Linux Kernel, Linus Torvalds, and was used as the official version control program for the Linux Kernel, and is maintained by Junio Hamano.

1.1 Install

To use Git, you obviously need to install it, so here's the different download instructions

Windows: Visit `https://git-scm.com/download/win/`
Mac: `git --version`
Debian Linux: `apt install git-all`

Visit `https://git-scm.com/download/` for more download information!

2.1 Initializing a Repo

To start a git repository, you must first initialize the directory in which the project is located in. To do this you will need to use the `git init` command on the terminal (Mac/Linux) or command prompt (Windows). This command creates a `.git` folder that contains all the tools and data necessary to contain various versions of our project.

2.2 Adding Changes

To add a file's changes to the staging area, you will need to use the `git add` command, like the following:
`$ git add <file>`
If you want to add all the files, then use the wildcard, `git add *`.

Now let's say there's files or directory you don't want added to the git repository (possibly build directories or large files), you can create a `.gitignore` file, and in it add the files or directories you want ignored.

.gitignore example

```
example_directory/
example.txt
*.zip # this will ignore all zip files
```

2.3 Committing Changes and Checking Status

So let's say you're satisfied with the changes you've made to the various files you've worked on. You could first see which file has been changed, and/or changes been added, once you've added the files that's had their changes made, what now? Well when you add these files, they're in a staging area, to commit these changes to the repository, well it's exactly what you think, use `git commit`.

Common Usages:

```
git commit -m "A nice message about changes" # Standard commit command
git commit -a -m "A nice message about changes + git add" # Adds and commits
```

2.4 Differences and Logs

Let's say you want to figure out the different changes between a file from the directory and the staging area version of the file, well as straightforward the commands are, it is the git diff command, and all you need to do is `git diff <file>`.
Now let's say you want to view the various commits that you've done in the git repository, use the `git log` command, and it will show you the commits you've done with the following information:

- A 40-character code (SHA), that creates a unique id for the commit
- The commit author
- The date and time of which the commit occurred
- The message written on the commit

Here's an example of a git log done on this repo:

```
commit e5af29e89846a396690e9d73d771ef6b298ca1a0 (HEAD -> main)
Author: mustafif0929@gmail.com <mustafif0929@gmail.com>
Date:   Thu Apr 22 22:37:33 2021 -0400

    update

commit 3fc278bcf2e74d151ead9192ef1991a235044c71 (origin/main)
Author: mustafif0929@gmail.com <mustafif0929@gmail.com>
Date:   Thu Apr 22 22:21:18 2021 -0400

    first commit
```

2.5 Git Project Workflow

A Git Project contains the following three parts:

1. Working Directory: where files are created, edited, deleted and organized (worked on).
2. Staging Area: where changes that are made to the working directory is listed
3. Repository: where Git permanently stores changes as different versions of the project

The Git workflow consists of editing files in the working directory, adding files to the staging area, and committing changes to a Git repository.

3.1 Showing Latest Commit Logs

In Git, the commit you are currently on is known as the `HEAD` commit. The `git show HEAD` command will output everything the `git log` command display for the `HEAD` commit and all the file changes that were made when committed.

3.2 Reset Using SHA

Let's say you want to go back to a previous commit, how does one do that? Well you use the `git reset commit_SHA` command, and what it does is set the `HEAD` commit to the `commit_SHA` commit.

To use `git reset`, you will need to use the first seven digits of the previous commit's SHA, and that can be found using the `git log` command to see records of previous commits.

3.3 Removing A File from Staging

Let's say you've added a file's changes to the staging area, but you change your mind, you can do the following, `git reset HEAD <filename>`, and what it does is remove the file from the staging area.

Note: This does not discard changes made in the working directory

You can use `git status` to make sure your file was properly removed from the staging area.

3.4 Going Back to the Last Commit

Let's say you want to go back to the changes made in a file in your last commit, or the `HEAD` commit. To do this, you can use `git checkout HEAD <filename>` and it will make the file in the working directory to go back to the changes made in the last commit.

You can make sure it worked by using `git diff` command to see if the rollback was successful. If nothing is outputted, that means it worked.

4.1 The Master Branch

When you initialize a git project, all work by default is done on the `master` branch. When you make your first commit, the `master` branch is automatically created.

Note: You can have the master branch be main using `git branch -M main` *after the first commit.*

You can create new branches from the `master` branch when you develop new features in your project or when you do testing. You can also see which branch you're on by using the `git branch` command.

```
# Here's an example of the Phaktionz-CLI branches
    $ git branch
    * beta
      edge
      main
      stable
# As can be seen, the master branch is switched to main
# and the current branch being worked on is the beta branch.
```

4.2 Creating a New Branch

In Git, the `git branch branch_name` command is used to create a new branch called `branch_name`. Branches should be named something that describes the purpose of the branch.
For example: The `beta` *branch represents the beta channel of Phaktionz-CLI.*

Also a branch name cannot contain white spaces: `some_name` or `some-name` are valid, however `some name` is invalid.

4.3 Deleting a Branch

In Git, you can delete a branch by using `git branch -d branch_name`, and then the branch named, `branch_name` will be removed.
Note: Usually it's a good idea to merge the branch with the `master` *branch before deleting.*

4.4 Merging a Branch

In Git, if you would like to merge a branch with another, use the `git merge` command. When you use the command `git merge branch_name`, it will merge the branch, `branch_name` to the branch you're currently on.

This is useful when your new feature works, so you can merge all the changes in that branch to your `master` branch.

5.1 Git Remote

When you work in a group, it is typical to collaborate with others using a Git server, or a website like Github (highly recommended).

A `remote` is essentially a shared Git repository that allows for multiple collaborators. Now keep in mind collaborators work independently, and merge their changes when ready.

5.2 Cloning a Remote Repository

Now how do you get to work on your partner's git repository...on let's say your organization's github? Well why not clone the repository using `git clone`.

It works locally:

```
$ ls
git-repository

# git clone remote-repo output-directory(optional)
$ git clone git-project/ git-clone-project
Cloning into 'git-clone-project'...
done.

$ ls
git-clone-project git-project
```

It works on remote sites, such as Github:

```
# HTTP Clone
$ git clone https://github.com/user/repo.git

# SSH Clone
$ git clone git@github.com:user/repo.git
```

5.3 Fetching Changes

The command `git fetch` downloads objects and changes from a remote repository `origin`.

It however doesn't automatically merge these changes to your current branch, but instead keeps all the changes in a new branch `origin/branch-name`.
Now let's see this in action:

```
$ git branch
* master

$ git fetch
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From /home/user/git-project
* [new branch]      master     ->  origin/master

$ git branch
* master
  remotes/origin/master
```

To merge these changes, it's like last section, use the command, `git merge` in the branch you want to have merge with the target branch.

To merge the fetch changes for our example, we do `git merge origin/master` on branch `master`.

5.4 Pushing Changes

In Git, the `git push origin branch-name` command pushes the branch `branch-name`, and all of the committed changes, to the remote `origin`. This branch can now be reviewed and fetched by collaborators.

5.5 Looking at the Remotes

In Git, the `git-remote -v` command returns a list of remote repositories that the current project is connected to.

- Git lists the name of the remote repository as well as its locations.
- Git automatically names this remote origin, because it refers to the remote repository of origin.
  - However, it is possible to safely change its name.
- The remote is listed twice: once for (fetch) and once for (push).

```
$ git-remote -v
origin  /home/user/git-project/
(fetch)
origin  /home/user/git-project/
(push)
```

5.6 A Typical Collaboration Workflow

A typical collaboration workflow is:

1. Fetch and merge changes from the remote
2. Create a branch to work on a new project feature
3. Develop the feature on a branch and commit the work
4. Fetch and merge from the remote again (in case new commits were made)
5. Push branch up to the remote for review

Note: Steps 1 and 4 are a safeguard against merge conflicts, which occur when two branches contain file changes that cannot be merged with the git merge command.

Here's a bash script that can be made to ease the need of the many commands:

```
#!/bin/bash
git fetch
git merge origin/master

echo "New Branch Name: "
read branch

git branch $branch
git checkout $branch

git fetch
git add *

echo "Message: "
read message

git commit -m $message

git merge origin/master

git push -u origin $branch
```