What is Version Control

Version control systems are software tools that help software team to manage their changes in code over time. Version control keeps track of every modification to the code, if a mistake is made, developers can revert back the changes or compare the code with earlier version of the code. One of the popular and widely used version control system is GIT.

What is GIT

GIT is by far the most widely used version control system in the world used by the developer community and other professionals. GIT is a mature, actively maintained open source project developed by Linux Trovalds(creator of LINUX). Because of it’s reliablity and vast features, a staggering number of software projects reply on GIT for version control, including huge commercial software companies as well as open source projects. GIT is a de facto standard at LEAPFROG, all of our projects source code is managed in GIT.

What is GitHub

GitHub is Git’s cloud-based **publishing tool** and **hosting platform**. It also has a desktop application for locally storing projects. With GitHub, you can:

- **Bring projects to life**. Git repositories are hosted on GitHub

and made “live.” This enables developers to post a site or application when it’s in development stages. By sending a link to a GitHub project, clients can easily test-drive a site in progress with functionality, rather than just looking at flat mockups.

- **Browse the most popular development projects**. Browse GitHub for “trending” repositories—an interesting way to check out other developers’ work and check out “starred” projects that are recommended by GitHub staff members. Public repository files can be downloaded as zip files and saved locally on your computer.

On GitHub, you can **Star**, **Watch**, and **Fork**:

- **Fork**: Make a copy of a project and start working on it yourself.
- **Watch**: Get updates when changes are made to a project you’re following.
- **Star**: GitHub’s version of the “Like” button on Facebook, it’s a voting system that enables developers to vouch for projects they think are excellent.

Why use GIT

GIT is a fast, scalable, distributed version control system with an unusually rich command set that provides both high-level operations and full access to the internals. Here are some of the major features of GIT:

- **Distributed in Nature**

  GIT is a distributed version control system where each developer gets their own local repository, complete with a full history of commits and branches.
- **Save Time**

  GIT is lightning fast, although we're talking about only a few seconds per command, it quickly adds up to your day.
- **Work Offline**

  GIT allows you to work while you're offline, with GIT, almost everything is possible to do on your local machine, be it commiting the code, browsing the code history, create branches.
- **Undo Mistakes**

  It's almost inevitable that developers make mistake while working on a project. A good thing about GIT is that we can undo almost every havoc we created in GIT. Since GIT rarely deletes the event history, this provides great peace of mind to developers.

Install GIT

- **Install GIT on Mac OS X**

  - Download the latest Git for Mac installer, and follow the prompts to install the GIT

    OR
  - ***Install Git with Homebrew***

1. Open your terminal and install Git using Homebrew:
   `$ brew install git`

- **Install GIT on Windows**
  - Download the latest Git for Windows installer.
- **Install GIT on Linux**

  Copy following commands in your terminal.

  > *On Debian / Ubuntu (apt-get)*

  `sudo apt-get update`
  `sudo apt-get install git`

  > *Fedora (dnf/yum)*

  `sudo dnf install git`
  or
  `sudo yum install git`

Once installed verify the installation by typing following in your terminal/(GIT bash in windows)
`git --version`
you should get the result something like
`git version 2.7.4`

Configure your GIT username and email, these details will be

associated with all the events you do in GIT.

```
git config --global user.name "Ram"
git config --global user.email "ram@jam.com"
```

Set up a repository

Initializing a new repository

To create a new repo, use the `git init`. It is a one-time command, used during the initial setup of a new repo. Executing this command will create a new `.git` directory in your current working directory. This will also create a new master branch by default.

Cloning an existing repository

If a project has already been set up in a repository, the `clone` command is used to obtain the clone of the repository. `git clone <repository-url>` is similar to `git init` such that it is also a one-time operation.

Using Branches

git branch

A branch in GIT represents an independent line of development. The `git branch` command allows you to create, list, rename and delete the branches.

Usage

> *create new branch*

`git branch <branch-name>`

> *delete branch*

`git branch -d <branch-name>`

> *rename/udpate branch*

`git branch -m <branch-name>`

git checkout

The `git checkout` command lets you navigate between the branches, tags, commits in git. Checking out a branch updates the files in the working directory to match the version stored in that branch/tag/commit.

Saving the Changes

> *git add*

The `git add` command adds the changes in the working directory and tells GIT that you want to include the updates of a file(s) in the next commit. The `git add` doesn't really affect the repository in any significant way; that changes are not actually recorded until you run `git commit`.

Usage

`git add <file> | <directory>`

> *git commit*

The `git commit` command commits the added file(s) snapshot to the project history. Once you run `git commit` you create a new version in your repository.

Usage

`git commit -m "<commit message>"`

> *git ignore*

You can ignore a file or directory by adding a **.gitignore** file on your repository. For example, if you use `java` in your project and your complier generates `.class` files which you don't want to share with other team members, so now what you can do is tell git to ignore all `.class` files in your project and don't track the changes of that file.

Usage

> *#you can add files in .gitignore file*
> *.class*
> *.less*
> *.log*
> ***.pyc # all the files with extension .pyc also within sub-directories*

> *#similarly you can also add directories to .gitignore files*
>
> *dist/*
>
> *downloads/*
>
> *.eggs/*

Updating the changes

> *git pull*

The `git pull` fetches the files from the remote repository and merges it with the local version. The `git pull` is similar to other git commands like `git fetch` and `git merge`. The `git pull` is equivalent to `git fetch` + `git merge`.

Usage
`git pull <remote> <branch-name>`

> *git push*

The `git push` is used to push changes to an upstream remote repository from the local machine. This command takes two command

- A remote name, eg *origin*
- A branch name, eg *master*

Inspecting the Repository

> *git status*

The `git status` command displays the state of the working directory and repository in whole. It lets you see which changes have been added, and which isn't.

Usage

`git status`

> *git log*

The `git log` command displays committed snapshots and lets you list the branch history, filter it and search for the specific changes.

Usage

`git log`

> *git diff*

The `git diff` shows the changes between two commits, branches or tags.

Usage

`git diff <commit1> <commit2>`

> *git difftool*

The `git difftool` command shows the changes between two commits, branches or tags using common GUI difftools, eg Meld,

Kdiff3 etc.

Usage

`git difftool -d <commit1> <commit2>`

Configure merge/diff tools

For this example we are using Meld.

`git config --global merge.tool meld`

`git config --global diff.tool meld`

Conflicts

While merging git branches (`git merge` / `git pull`), git may report a **merge confict** error with the files that have conflict. This conflict should be resolved.

With `git status` , the changes in the files and the files that are to be resolved could be figured out.

GIT leaves markers in files to indicate where the conflict arose:

> *<<<<<<<< Head*
> *this part indicates the state of the current branch*
> *======== this indicates the break between the conflicts*
> *this part indicates the state of the other branch*

To resolve conflicts:

- Edit the area between <<<<<< and >>>>>>
- Remove the status lines (<<<<<< , ====== , >>>>>>)

- `git add` to mark it resolved and finally `git commit` to finish the merge

Other useful GIT commands

Following are some handy GIT commands that will save your time and headache.

> *git cherry-pick*

We can use `git cherry-pick` to *cherry-pick* (merge the changes made in the commit) with a given SHA and merge it to the current branch

Usage
`git cherry-pick <SHA hash>`

> *git stash*

Stashing takes the dirty state of the current working branch and push it in the stash stack. Stash can be reapplied any time just by popping out the stash stack element.

- We can stash our work by using `git stash`
- Similarly, we can list the stash stack element using `git stash list`
- To apply the stash changes to the current brach, use `git stash pop <index>`. It removes a single stashed element from

the stash stack and applies it on top of the current working branch, ie. do the inverse operation of `git stash` save.

- `git stash apply <index>` is like `git stash pop`, but do not remove the state from the stash stack.

> *git bisect*

The `git bisect` is a tool that allows you to find an error-prone commit in your GIT history. To get started, checkout the buggy branch and find the good commit(do `git log` and find the *SHA hash* that is good).

- Start the bisect with `git bisect start`
- Then mention the bad commit to bisect `git bisect bad <SHA hash>`
- Then mention the good commit `git bisect good <SHA hash>`

After this GIT will checkout to one of the commits from the range of good and bad commit, if your code looks good in it mention it as good(`git bisect good`) or else if it is not what you're expecting then note it as bad(`git bisect bad`). Repeat the process until you find the ultimate buggy commit.

> *git blame*

`git blame` will reveal for every line in the file the author, the commit hash that saw the last change in that file, and the timestamp of the commit

Usage

```
git blame <filename>
```