Git Session by Savitoj Singh <savsingh@redhat.com>

Introduction

- Created by Linux Torvalds who created Linux.
- Written in Perl and C
- GIT is Distributed version control system (DVCS)

Little History

`First Generation:`

- Single-File system
- No Network support
- SCCS, RCS

`Second Gen:`

- Multi-file
- Centralized
- SVC, VSS, SVN

`Third Gen:`

- Distributed
- Changeset
- Git, HG, Bitkeeper

Centralized Model:

- dev send to one repo in Centralized place.
- CVS and Subversion

Hierarchical model:

- dev push to substtem based repos
- periodically merged with main repo

Distributed model:

- Dev push to their own repo
- project maintainers pull changes into the official repo
- Backups are easy, each clone is full backup
- Reliable branching and merging
    - feature branches
    - Always work under version control.
    - Apply fixes to different branches
    - Full local history

Configure:

```
$ git config --system
```

`User level:`

```
git config --global stored in .gitconfig
```

`Repo level`

```
git config
```

`Examples:`

```
git config --global --list
git config --global user.name "Savitoj Singh"
git config --global user.email "savsingh@redhat.com"
git config --global core.editor vim
git config --global help.autocorrect 1
git config --global color.ui auto
```

WorkFlow and Repos

- Create repo
- add files
- commit changes
- view history
- view a diff
- working copy, staging and repositoru
- delete files
- clean the working copy
- ignore files with .gitignore

```
git init
git status
git add README.md
git commit
git log

git diff blabla..blabla2

git diff HEAD~1..HEAD

git diff HEAD~1..

touch file1 file2

modify the file and then checkout

vim README.md

git checkout README.md   - Undo file

multiple changes
git reset --hard

git reset --soft HEAD~1 -> moves files to working copy
git reset --hard HEAD~1 -> clean up and discard all change
s

(git fetch origin master && git reset --hard origin/master
)

Git Ignore file
vim .gitignore
/logs/*.txt
/logs/*
```

Working with remotes

- Clone
- Remotes
- Fetch changes
- Merge changes to local
- Pull from Remotes
- Push to remote
- working with tags

```
Clone

git clone https://blabla.git
git log
git log --oneline

git log --graph
git shortlog
git shortlog -sne

git show HEAD
git show HEAD~1

git show 3244221

git remote -v
```

Protocols used in Git

- Protos:
    - HTTP
    - GIT
    - SSH
    - file

Branching:

```
git branch
git branch -r
git tag
git remote add origin https://blabla
git merge origin/master
Fast-Forward - pull/push the delta
git branch -r
git pull -> git fetch ; git merge origin/master
```

Tagging

```
git push
git tag
git tag show v1.0
git tag -l "v1.8.5*"
git tag v1.0 -> tag to the latest master branch or latest
commit.
git tag (Unsigned tag)
git tag -a v1.0_with_message
git tag -s v1.0_signed
git tag -v v1.0_with_message
git tag -v v1.0_signed
git tag -a v1.2 9fceb02 - tag later
```

- Push tags remotely

```
git push origin v1.5
git push origin --tags
```

- Delete tag:

```
git tag -d v1.4-lw
git push origin --delete <tagname>
```

- Local branches
- stash changes
- merging branches
- rebase commits
- cherry pick commits
- working with remote branches

Understanding makes easy

```
git log --graph --oneline
git log --graph --oneline --all --decorate
git config --global alias.lga "log --graph --oneline --all
 --decorate"
git lga
git checkout feature1
git commit -am ""added test"
git lga
git checkout master
git branch fix1 877552
git checkout fix1
git commit -sam "Fixed bug 1233"
git lga
git checkout master
git branch -m fix1 bug123
git branch -d bug123
git checkout -b feature2
echo "Feature2" >> README.md
git reflog ( 30 days - default)
git branch bug123 55522sd2
git lga
```

Stash

```
git stash
git stash list
git stash apply (just apply the changes) vs git stash pop
(Remove from list)
git stash branch feature2_more
```

Git Hooks

- Client Side:
    - pre-commit
    - prepare-commit-msg
    - commit-msg
    - post-commit
    - post-checkout
    - pre-rebase
- Server Side:
    - `pre-receive`

      The first script to run when handling a push from a client is pre-receive.It should always reside in the remote repository that is the destination of the push, not in the originating repository.
    - `update`

      The update hook is called after pre-receive, and it works much the same way. It’s still called before anything is actually updated, but it’s called separately for each ref that was pushed. That means if the user tries to push 4 branches, update is executed 4 times.
    - `post-receive`

      The post-receive hook gets called after a successful push operation, making it a good place to perform notifications. For many workflows, this is a better place to trigger notifications than post-commit because the changes are available on a public server instead of residing only on the user's local machine. Emailing other developers and triggering a continuous integration system are common use cases for post-receive.The script takes no parameters, but is sent the same information as pre-receive via standard input.

More reading here:

- https://www.atlassian.com/git/tutorials/git-hooks
- https://git-scm.com/book/en/v2/Customizing-Git-Git-Hooks

FAQs

1. **fetch vs pull**

In the simplest terms, `git pull` does a `git fetch` followed by a `git merge`
As per Git documentation – git pull:
`In its default mode, git pull is shorthand for git fetch followed by git merge FETCH_HEAD.`

- What is the Git Flow do you use?

```
git clone git@gitlab.example.com:savsingh/ansible.git
cd ansible
git remote add upstream git@gitlab.example.com:eng-ops/ansible.git
git pull --rebase upstream master
```

```
###  if you want to change something, like jira-db
git checkout master
git pull --rebase upstream master
git push origin master
git checkout -b jira-db
git push origin  jira-db
<make changes>
git push origin jira-db
<create MR>

# if you want to make new changes, repeat steps under "###
"
```

2. **rebase vs merge**

Example:

```
It's simple, with rebase you say to use another branch as
the new base for your work.
If you have for example a branch master and you create a b
ranch to implement a new feature, say you name it omg-feat
ure, of course the master branch is the base for your new
feature.

Now at a certain point you want to add the new feature you
 implemented in the master branch. You could just switch t
o master and merge the omg-feature branch:

```

```
$ git checkout master
$ git merge omg-feature
but this way a new dummy commit is added, if you want to a
void spaghetti-history you can rebase:

$ git checkout omg-feature
$ git rebase master
and then merge it in master:

$ git checkout master
$ git merge omg-feature
This time, since the topic branch has the same commits of
master plus the commits with the new feature, the merge wi
ll be just a fast-forward.
```

3. What is FETCH_HEAD?

```
`FETCH_HEAD` is a short-lived ref, to keep track of what h
as just been fetched from the remote repository. git pull
first invokes git fetch, in normal cases fetching a branch
 from the remote; `FETCH_HEAD` points to the tip of this b
ranch (it stores the SHA1 of the commit, just as branches
do). `git pull` then invokes git merge, merging `FETCH_HEA
D` into the current branch
```

Resources:

- Stackoverflow.com
- atlassian.com
- git-scm.com