# GIT: Fast and Furious

Shahab Rezaee <shahabrezaee@gmail.com>

## What is Git?

Git is:

- A Distributed version control, means:
  - Each user maintain their own repository
- A tool that tracks changes instead of versions, specially for text content

Every single updates are **change sets** which shapes repository state. There is no master repository but many working copies which contains their own **change sets**. Example: All of the followings may be valid:

```
Repo 1: A -- B -- C -- D -- E -- F
Repo 2: A -- B -- C
Repo 3: A -- B -- E -- F
```

So, there is no centralized copy on server tracking every movements! ***Goodbye dictators! Hello FREEDOM!***

Triumphs:

- No network access required
- No single point of *FAILIER*
- Developers could works independently
- Easier code review process

Git generally is not good for tracking binary files. So, storing many binary objects in git is bad habit! (see Git LFS)

## Creation!

```
$ cd path/to/repository/root/
$ git init
```

See .git directory!

Create your first files. Commit your changes.

```
$ touch first.txt
$ git add .
$ git commit -m "Initial commit!"
```

-m means message. You could use `git commit` to access editor to write multi line commit message. You could specify editor in eather ~/.bashrc or ~/.bash_profile :

```
export EDITOR=nano
```

Write good messages for commits. (How to write a Git Commit Message) Use meaningful conventions such as issue numbers, support tickets, git refs and etc in message title.

Now, see current changes:

```
$ git log
$ git log --since="2016-06-04"
$ git log --until="2016-06-04"
$ git log --author="Shahab"
$ git log --grep="Init"

$ git help
$ git help init
```

## Git structure

### Git levels

```
  +---------------+
  |  repository   |
/ +---------------+
|
| git commit
|
\ +------------------+
  |  staging index   |
/ +------------------+
|
| git add
|
\ +----------------------+
  |  working directory   |
  +----------------------+
```

## Change Sets

- **change set** is a single patch file may affects many files in repository.
- Git generates a SHA-1 checksum for every **change set**.
- SHA-1 guaranties that same data have a same checksum.
- SHA-1 generate 40 character hexadecimal ([0-9a-f]) string.

```
+------------+
|  checksum  |
+------------+
|  parent    |
+------------+
|  author    |
+------------+
|  message   |
+------------+
```

Example: Consider followings are initial commits of repository:

```
A -- B -- C
```

thus:

```
A.parent == NULL
B.parent == A.checksum
C.parent == B.checksum
```

## HEAD

HEAD is:

- pointer to tip of current branch of repository
- last state of repository checked out
- parent of next commit

```
$ cd .git
$ ls # there is a file named HEAD
$ cat HEAD  # location of current head
$ cd refs/heads
$ cat master  # checksum of lastest commit
```

# Create changes!

```
$ git status  # there is no changes
$ touch second.txt
$ touch third.txt
$ git status  # there are untracked files (changes not staged)
$ git add second.txt
$ git status  # second.txt staged to be commited, third.txt still untracked
$ git commit -m "Add second.txt to the project"
$ git status  # third.txt is untracked
$ git log  # new commit is listed
$ git add third.txt
$ git commit -m "Add third file to the project"
$ git log  # another commit is listed
$ echo "a new line at the end" >> first.txt
$ git status  # first.txt is modified. changes are tracked but not staged yet
$ git diff  # see difference between current working directory and HEAD
$ echo "hellllllllllllllllllllo again!" >> third.txt
$ git diff  # multiple file changes
$ git add first.txt
$ git diff  # Oh staged changes are not available
$ git diff --staged  # dada! see differences between staged index and HEAD (git 1.7+, --
$ git add third.txt
$ git commit -m "Minor bottom changes ;)"
$ touch to_be_deleted.txt
$ git commit -am "Add files to delete later"
$ rm to_be_deleted.txt
$ git status  # changes not staged
$ git rm to_be_deleted.txt
$ git status  # delete is staged to be commited
$ git commit -m "to_be_deleted is now deleted!"
$ touch return_of_to_be_deleted.txt
$ git commit -am "To Be Deleted: The Return"
$ git rm return_of_to_be_deleted.txt
$ ls  # Horay! It's gone!
$ git status  # It's also staged!
$ git commit -m "Fixing consistant to_be_deleted.txt bug"
$ mv first.txt primary.txt
$ git status  # seems first.txt is deleted (tracked), primary.txt is created (untracked)
$ git add primary.txt
$ git rm first.txt
$ git status  # OH! Shit! git got it! it's renamed!
$ git mv second.txt secondary.txt
$ git status  # Horay! renamed and staged!

$ git reset HEAD --hard  # we describe it later!
$ nano first.txt  # change something in one line
$ git diff first.txt
$ git diff --color-words first.txt
```

## Time machine! Get things back or OMG! I am in love with Git!

### From working directory to HEAD

#### Undo changes of one file

```
$ echo "I gonna fallllllll%@#%$#^#%&" >> first.txt
$ git status  # Next day when I check the repo: Ohhhh!
$ git checkout -- first.txt  # git checkout first.txt will work on many cases but what i
```

### From staged to working directory

```
$ echo "I gonna fallllllll%@#%$#^#%&" >> first.txt && git add first.txt
$ git status  # my god! they are staged!
$ git reset HEAD first.txt
```

### From committed to working directory

```
$ echo "I gonna fallllllll%@#%$#^#%&" >> first.txt && git commit -am "sombjhgsv f hsafhi
$ git status
$ git log  # HOLY CRAP! They are commited
```

Consider this:

```
A -- B -- C -- D -- E
```

We already know that each commit is depends to the previous commit. So if we remove one, repository would loose its data integrety. However, we may able to change the last commit!

#### Change last commit

```
$ git commit -amend -m "A nice good morning message!"
$ git status  # Nothing!
$ git log  # message of last commit is looking nice! Note that checksum and date are cha
$ echo "one more other line" >> first.txt
$ echo "one more other line" >> second.txt
$ git add first.txt
$ git commit -m "Append one more line to first and second"  # oops! we forgot second one
$ git add second.txt
$ git commit --amend --no-edit  # Save or change message! Done!
```

### Bring commited changes out of commit

```
$ git checkout CHECKSUM -- first.txt
$ git status
$ git reset HEAD first.txt
```

### Revert last commit!

```
$ git checkout -- first.txt  # Back to last commit
$ git log
$ git revert CHECKSUM  # revert last commit: git revert HEAD
$ git log  # a revert commit!
$ git revert CHECKSUM  # try to revert to older commits may cause confilicts
$ git revert --abort  # return to last state
```

## To infinity... and beyond!

There are three kind of reset:

- soft: no changes in niether staged index nor working directory
- mixed (default): change staged index to match HEAD but apply no changes to the working directory
- hard: change both staged index and working directory to match HEAD

### Soft reset

```
$ cat .git/HEAD
$ cat .git/refs/heads/master
$ git log
$ git reset --soft CHECKSUM
$ git log
$ git status
$ git diff --staged
$ cat .git/refs/heads/master
$ git reset --soft CHECKSUM_OF_LATESET  # it's not destructive
```

### Mixed reset

```
$ git log
$ git reset --mixed CHECKSUM
$ git log
$ git status  # committed changes are unstaged now
$ git add .
$ git reset HEAD FILENAME
$ git reset --mixed CHECKSUM_OF_LATESET  # it's not destructive too
```

#### Hard reset

```
$ git log
$ git reset --hard CHECKSUM
$ git log
$ git status  # Nothing here!
$ echo "this would be gone!" >> first.txt
$ git reset --hard HEAD
$ git status  # nothing here! may be destructive or painful!
$ git reset --hard CHECKSUM_OF_LATESET  # it's here again but we are lucky because it's
```

#### Reset the Reset

```
$ git reflog  # git help reflog
$ git reset --hard HEAD~3
$ git log
$ git reflog
$ git reset --hard HEAD@{1}
```

### From dirty to clean!

```
$ touch jerk1.txt
$ touch jerk2.txt
$ git status
$ git clean
$ git clean -n  # Test run of clean command
$ git add jerk1.txt
$ git clean -f  # jerk1.txt still here
$ git reset HAED jerk1.txt
$ git clean -f
$ git status
```

## Ignorance!

```
$ touch to_be_ignored.txt
$ git status
$ touch .gitignore
```

.gitignore accept patterns in each line:

- Very basic regular experssins * ? [aeiou] [0-9]
- `!` means NOT! Example: `*.jpg !logo.jpg`
- Triling slash ignores directoris: `static/`

```
$ echo "*_ignored.*" >> .gitignore
$ git status  # It's ignored
$ git commit -am "add gitignore"
```

See github .gitignores You could create global ignorance with `git config`.

Now if we want to ignore tracked files:

```
$ echo "third.txt" >> .gitignore
$ git status
$ echo "some changes!" >> third.txt
$ git status
$ git rm --cached third.txt
$ git commit -am "Untrack third.txt but still available from repository"
```

Git ignores empty directories by default, so if you want add an empty directory to project add `.gitkeep` to it!

## Go to a git trip!

Every Tree-ish are a git trip ticket! But wait a min! What is a Tree-ish:

- something that references part of a tree
- is `ish` because it's vary widly
- is referencing a commit

Tree-ish:

- Full 40 char checksum
- Part of checksum (4 char at least) from left.
- HEAD pointer
- branch refs
- tags refs
- ancestry

Ancestry:

- Parent commit:
  - `HEAD^`, `1f9c5bb^`, `master^`
  - `HEAD~1`, `HEAD~`
- Grandparent Commit:
  - `HEAD^^`, `1f9c5bb^^`, `master^^`
  - `HEAD~2`
- Great-grandparent commit:
  - `HEAD^^^`, `1f9c5bb^^^`, `master^^^`
  - `HEAD~3`

### Tree-ish contents

```
$ git ls-tree HEAD
$ git ls-tree --name-only
$ git show HEAD
$ git show CHECKSUM_OF_BLOB
$ git show first.txt  # ambiguous
$ git diff HEAD~2..HEAD
$ git diff --stat HEAD~2..HEAD
$ git diff -b TREEISH..TREEISH  # ignores space changes
$ git diff -w TREEISH..TREEISH  # ignores all spaces
```

### Logs

```
$ git log --oneline
$ git log --since="2 days ago"
$ git log --until=2.days
$ git log HEAD~5..HEAD~
$ git log -- first.txt
$ git log --oneline --stat --summary
$ git log --format=short
$ git log --graph --decorate
```

## Branches

### Create, Move and Delete

```
$ git branch
$ git branch new_feature
$ git brance
$ ls -la .git/refs/heads
$ cat .git/refs/heads/new_feature
$ git checkout new_feature
$ cat .git/HEAD
$ echo "in new branch" >> second.txt
$ git commit -am "new feature aded"
$ git log
$ git checkout master
$ git log
$ git checkout new_feature
$ git checkout -b feature_hotfix
$ echo "it is too hoooooooot" >> second.txt
$ git commit -am "it's feel warmer now"
$ git checkout master
$ git log --oneline --graph --decorate --all
$ echo "some annoing not commited" >> first.txt
$ git checkout new_feature  # Cause error! It should be mostly clean
$ git reset --hard HEAD
$ touch temp.txt
$ git checkout new_feature  # this is the meaning of 'mostly clean'
$ git status
$ rm temp.txt
$ git diff master..new_feature
$ git branch --merged
$ git checkout master
$ git branch -m new_feature feature
$ git branch delete_me
$ git branch -d delete_me
$ git checkout -b delete_again
$ echo "this will be deleted" >> first.txt
$ git commit -am "rest in peace"
$ git checkout master
$ git branch -d delete_again  # Are you sure?
$ git branch -D delete_again
```

### Merge

```
      A---B---C   [feature]
     /
    D---E---F---G---H   [master]

      A---B---C   [feature]
     /         \
    D---E---F---G---H   [master]
```

```
$ git merge feature
$ git merge --no-ff hotfix
$ git branch --merged
```

Strategies to avoid conflicts: - Keep lins short - keep commit short - merge often

## Stash

```
$ echo "I am editing this file" >> first.txt
$ git stach save "some changes"
$ git stash list
$ git stash show REF
$ git stash show -p REF
$ git stash pop REF
$ git stash list
$ git stash
$ git stash list
$ git stash apply REF
$ git stash list
$ git stash drop REF
$ git stash clear
```

## Remotes

```
$ git remote add origin URL
$ cat .git/config
$ git remote
$ git remote -v
$ cat .git/config
$ git remote rm origin
$ git remote add origin URL
$ git push -u origin master
$ cat .git/config
$ ls .git/refs/remote
$ ls .git/refs/remote/origin
$ cat .git/refs/remote/master
$ git branch -r
$ git branch -a
$ git diff master origin/master
$ git merge origin/feature
$ git push origin :delete_me
$ git push origin --delete delete_me
```

## Rebase

```
      A---B---C   [feature]
     /
D---E---F---G   [master]

              A'--B'--C'   [feature]
             /
D---E---F---G   [master]
```

```
$ git checkout -b extra_feature master
$ echo "landing giant boeing" >> first.txt
$ git commit -am "initial commit -- extra feature"
$ git checkout -b hotfix_chili master
$ echo "An chili pepper here!" >> second.txt
$ git commit -am "fix a hot a possible"
$ git checkout master
$ git merge hotfix_chili
$ git branch -d hotfix_chili
$ git log --oneline --decorate
$ git checkout extra_feature
$ git log --oneline --decorate
$ git rebase master
$ git log --oneline --decorate
$ git reflog
$ git reset --hard HEAD@{NUM}
$ git rebase -i master
$ git push -f origin extra_feature  # should push forcefully
```

The Golden Rule of Rebasing: **Once you understand what rebasing is, the most important thing to learn is when not to do it. The golden rule of git rebase is to never use it on public branches.** see: Merge vs Rebase

## Tagging

```
$ git log --oneline --decorate
$ git tag v0.1.0
$ git tag
$ git show v0.1.0
$ git tag -a v0.1.0rc HAED~1 -m "First RC realse"
$ git log --oneline --decorate
$ git reflog
$ git push origin --tags
```

## Git workflow

see: Comparing Workflows

### Centralized Workflow

- Everyone works on master!
- Needs many rebases
- Confilicts are totaly accepted!
- Only one branch make the way clean (!?)

### Feature Branch Workflow

- Every feature have thier own brach
- Based on pull requests
- Less confilicts may happens
- There may be many waits for a branch owner to finish their work

### Gitflow Workflow

see: A successful Git branching model

- There are 5 basic branch type:
- master: stable release
- develop: main development branch
- feature: branch for new feature development
- release: branch from develop for every single feature releases e.g. 1.5.2 -> 1.6 not 1.5.3 nor 2.0
- hotfix: branch from master to apply hotfixes to master
- Compatible with semver
- Rare confilicts
- Descentralized while Centralized
- Very collaborative atmosphere
- Never breaks stable releases
- A bit complex to handle, nevertheless there is a handy automation tool

git-flow is a tool to automate this model. see: git-flow cheatsheet

### Forking Workflow

- Every member works on thier own copy of repository (fork)
- Based on pull requests
- There is a maintainer responsible to merge everything
- Prevents unwanted changes to main repository
- Usually opensource/public projects use this model

### Patch Workflow

see: Patch Workflows

- Every member works on thier own copy of repository (fork)
- Based on emailed patches! (see: A3.9 Git Commands - Email)
- There is a maintainer responsible to merge everything
- Linux Kernel uses this model!