Git - A practical guide

Author: Jose Hernandez
Date: Jan 2020

Setup

- Install from: http://git-scm.com
- Online Documentation: http://git-scm.com/docs

Configure user identity for signing commits:

```
git configure --global user.name “Firstname Lastname”
git configure --global user.email “youremail@somewhere.com”
```

Show help for a command:

```
git [command] --help
(or)
git help [command]
```

Local Repos

Create a local repository:

```
mkdir MyProject
cd MyProject
git init
```

Note: a new ***.git*** folder is created containing git’s supporting objects and structures.

Add a file (or files) to version control (a.k.a. tracking):

```
git add todo.txt

# includes all new, modified and deleted files
git add .
```

Remove a file (or files) from version control:

```
git rm todo.txt
git rm *.txt
```

Check the status of the repository:

```
git status
```

Create a commit:

```
git commit -m “some message about the commit”
```

Amend the last commit:

```
git commit --amend
```

List commits:

```
git log
git log --format=fuller
git log --format=raw
git log --oneline
git log --graph
git log --decorate
git log --all
```

Pro Tips:

To dig a bit more into the internals of Git by inspecting the content of *.git*:

- Inspect (and decompress) objects:

  ```
  git cat-file -p [object hash]
  ```

- Calculate object hash:

```
echo “something” | git hash-object --stdin
```

Excluding files from Git tracking (.gitignore)

Define exclusions for all repos:

```
cat /home/user/.gitignore_global
.DS_Store
```

Define exclusions per repo:

```
cat /projects/ios/MyApp
.DS_Store
*.xcodeproj/*.pbxuser
*.xcodeproj/*.perspective*
*.xcodeproj/*.mode*
*.xcodeproj/xcuserdata/*
*.xcodeproj/project.xcworkspace/xcuserdata/*
```

Define exclusions per folder:

```
cat /projects/ios/MyApp/tests
**/*.xml
!summary.xml
```

Note: "!" can be used to exempt a pattern from the exclusion, effectively including the matching items

Working Tree (or directory)

Folder structure containing changes.

```
# what's changed between Staging and Working Three?
git diff

# discard changes on file (Destructive!!!)
git checkout -- [file]
```

Staging Area (a.k.a. index, cache)

Contains changes that will be included in the next commit.

```
# add to staging area
git add [file|.]

# what's changed between the HEAD Commit and Staging Area
git diff --cached

# unstaging changes
git reset -- [file]
```

HEAD Commit

Last commit (tip) on the current branch.

```
# what's changed between the HEAD Commit and the Working Tree
git diff HEAD
git commit -m “some message”
```

Stashing

Saving the current state of the Working Tree and Staging Area while going back to a clean working space by reverting the Working Tree to match the HEAD Commit. This operation may be known as *shelving* in other SCM systems.

```
# stash current state
git stash push
(or)
git stash

# list stashed changes
git stash list

# show details of the stash
git stash show

# restore stashed state and remove from the list
git stash pop

# restore stashed state and keep in the list
git stash apply
```

Note: Restoring a stash may conflict with current changes.

Branches

List local branches:

```
git branch   (* indicates current)
```

Create a branch:

```
git branch [branch name]
```

Switch to a branch:

```
# switch to branch by name
git checkout [branch name]

# switch to previous branch
git checkout -
```

Create a branch and switch:

```
git checkout -b [branch name]
```

Delete a branch (locally):

```
git branch -d [branch name]
```

Checkout to a commit (***detached HEAD***):

```
git checkout [hash other than latest]
```

Note: If some commits are created and then a branch is checked out, those commits would be lost since they will be ***unreachable***. Unless a new branch is created using the hash of the last commit as reported by the branch checkout command.

To recover commits by “reattaching the HEAD”:

```
git branch [branch name] [last commit hash]
```

Pro Tips:

- Reset branch A to the tip of branch B:

```
git checkout A
git reset --hard B
```

- Reset a branch to a commit:

```
# reset to a specific commit
git reset --hard [hash]

# reset to previous commit
git reset --hard HEAD^

# reset to two commits back
git reset --hard HEAD^^

# reset to 1 commit back (by count)
git reset --hard HEAD~1

# reset to 25 commits back
git reset --hard HEAD~25
```

Tags

Add a tag (locally):

```
# tag the lastest commit
git tag [tag name]

# tag a specific commit
git tag [tag name] [hash]
```

Delete a tag (locally):

```
git tag -d [tag name]
```

List tags:

```
git tag
```

Rebasing

Move the point where a branch started to a more recent point of the three:

```
# rebase current branch to master
git rebase master
```

Pro Tip:

- Reassemble commits with *interactive rebase*:

  ```
  git rebase -i HEAD-2   (rebase last two commits)
  ```

Merging

Attempt a merge:

```
# switch to target branch
git checkout [target branch]

# merge source branch into current branch
git merge [source branch]
```

Possible outcomes:

- Files do not conflict (happy path).
- Commits in the target branch are right behind commits in the source branch, therefore a ***fast-forward*** is done. Optionally use **--no-ff** to avoid fast-forward and force the creation of a merge commit.
- Conflicts are found requiring manual resolution.

Check conflicts:

```
git diff
```

Edit the files with conflicts. Once all conflicts are resolved, add the updated files to the staging area and commit. Optionally, abort the merge.

Cherry Picking

```
# switch to target branch
git checkout [target branch]

# copy changes from commit to current branch
git cherry-pick [commit hash]
```

Resolve conflicts, add updated files to staging area and commit. Only changes are copied over, no relationship between the commits is stored.

Pro Tip:

- To track the the cherry-picked commit:

  ```
  git cherry-pick [commit hash] -x

  # do not include a message, note the missing -m option
  git commit
  ```

  An editor will open to allow adding a message in addition to the hash of the cherry-picked commit.

Remote Repos

Those other than your local copy of a repository and accessible via common protocols like *SSH*, *HTTP*, *git* or even your local file system.

Remotes are usually ***bare repositories***, which only constain a *.git* folder, and don't have working trees or HEAD references.

Cloning

```
# clone with the same remote name on the current path
git clone [repo URL/path] .

# clone with a different name for local repository
git clone [repo URL/path] [target path/name]
```

Pro Tips:

- Cloning from *Git* hosting sites (Ex: GitHub) only copies the ***default*** branch locally, usually the *master* branch. Other branches can be copied locally by checking them out. Declaring a ***default branch*** is not a Git feature but a hosting platform one.
- When migrating a repository between hosting platforms (Ex: from GitLab to GitHub), the cloning of the source repository should produce a ***mirror***, which includes all the information about the repository.

  ```
  git clone --mirror [source repo URL] [local folder]
  cd [local folder]
  git push --mirror [empty target repo URL]

  (optionally)
  cd ..
  rm -rf [local folder]
  git clone [target repo URL] [local folder]
  ```

Origin

It's the default name of a remote, it's just a convention.

List current remotes (could be more than one):

```
git remote -v
```

Show origin remote status:

```
git remote show origin
```

Add an origin remote to the local repository:

```
git remote add origin [remote URL]
```

Update the origin remote (Ex: when renaming a remote repo):

```
git remote set-url origin [new remote URL]
```

Pushing and Pulling

While on the working local branch:

```
git push [remote] [branch]
(or)
# defaults to origin and current branch
git push

git pull [remote] [branch]
(or)
# defaults to origin and current branch
git pull
```

Note: git pull = git fetch + git merge

Fetch remote changes into local repo:

```
git fetch
```

Merge remote changes (already fetched) into local branch:

```
git merge
```

To rebase instead of merging with pull:

```
git pull --rebase [remote] [branch]
(or)
# defaults to origin and current branch
git pull --rebase
```

Note: git pull --rebase = git fetch + git rebase

Branches and Tags

A *tracking branch* is set to track or follow a remote branch.

List remote branches:

```
git branch -r

# includes local and remote branches
git branch -a
```

Push a branch to origin remote:

```
git push -u origin [branch name]
```

Delete a branch from origin remote:

```
git push -d origin [branch name]
```

Push a tag to origin remote:

```
git push origin [tag name]

# push tags along with commits:
git push origin master --tags
```

Delete tag from origin remote:

```
git push origin :refs/tags/[tag name]
```

GitHub

http://github.com

- Free personal accounts with free public repos (with unlimited collaborators), and private repos (with limited number of collaborators)
- Access control
- Branch protection rules
- Pull requests management
- Actions (workflows for CI automation), with access to third party developed actions
- Project management
- Wiki
- Web Hooks
- Personal Access Tokens (PAT)
- Issue tracking

Forking

Fork is not a *Git* term but a GitHub one. Forking allows to have a clone of another party's

repository in your account. Any local clone of the fork repository would only show *origin* as a remote but may also add an ***upstream*** remote linking it to the forked repository.

Add an *upstream* remote:

```
git remote add upstream [forked repo URL]
```

Verify there is now an *upstream* remote in addition to *origin*:

```
git remote -v
```

This setup allows to pull changes from the original (forked) repo and merge them with the local one. Then they can be pushed to the personal fork remote repo along with any local changes.

Note: An alternative to *forking* is becoming a contributor to the original repository.

Pull Requests

Allow to present contributions to a repository for approval and subsequent integration via merging.

This practice promotes ***Code Reviews*** and ensure no code is integrated into the main code base without being looked at by someone other than the author.

In GitHub you can even add comments to each line of code during the review. The author is then notified about these comments and can take any necessary action and update the code before the Pull Request is finally approved.

Generally in shared projects, the main branch is ***protected*** so only adminstrators can directly push changes to it. The only way for non-admin contributors to get their changes integrated is through a Pull Request.

Pull requests require the proposed changes to be in a separate branch:

```
git checkout -b [feature branch]
(commit some changes)
git push origin [feature branch]
```

Then create the pull request from GitHub's user interface.

Git Workflows

Centralized

- One **master** branch, all contributors can commit to it directly
- Regular CPR (Commit, Pull-Rebase) cycles required
- No approval process enforced
- Master branch may contain "broken" code
- Great for very small teams

Feature Branch

- A new branch is required for any new feature
- New ***feature*** branches are created from the *master* branch
- Feature branches are pushed to a central repository and may be shared between contributors
- Feature branches are submitted via Pull Requests for approval and merging into the *master* branch
- Promotes collaboration and code reviews
- Provides the foundation for other workflows

Gitflow

- Defines a strict branching model designed around a release process
- Ideal for large projects with a well defined release schedule
- The *master* branch stores the release history and a ***develop*** branch is used to integrate features
- Feature branches are branched from the *develop* branch instead of *master*, and merged back when the feature is completed
- A ***release*** branch is created from the *develop* branch when enough features have been completed
- Only bug fixes, documentations and minor changes go into the *release* branch
- Once ready, the *release* branch is merged into the *master* branch
- Also the *release* branch is merged back into the *develop* branch to incorporate any new changes
- Commits in the *master* branch are tagged with a version number
- Once a release is completed, the *release* branch is deleted
- Available ***git-flow*** toolset which wraps and extends the Git standard CLI and provides an

abstraction oriented to manage the repository following Gitflow (Ex: init will create the develop branch)

Nice tutorial: https://www.atlassian.com/git/tutorials/comparing-workflows

---

GUI Applications (free)

- Source Tree: https://www.sourcetreeapp.com
- GitAhead: https://gitahead.github.io/gitahead.com
- gitk: https://git-scm.com/docs/gitk
- git-gui: https://git-scm.com/docs/git-gui

The end. Enjoy!