**GIT BASICS**

## **Getting a Git Repository**

You typically obtain a Git repository in one of two ways:

* You can take a local directory that is currently not under version control, and turn it into a Git repository, or
* You can clone an existing Git repository from elsewhere.

In either case, you end up with a Git repository on your local machine, ready for work.

## **Cloning an Existing Repository**

* git clone <https://github.com/fvenkat/ReactjsPhotoGallery.git>
  + - * creates ReactjsPhotoGallery dir and it clones
* git clone <https://github.com/fvenkat/ReactjsPhotoGallery.git> <Reactrepo>
* creates Reactrepo dir and it clones

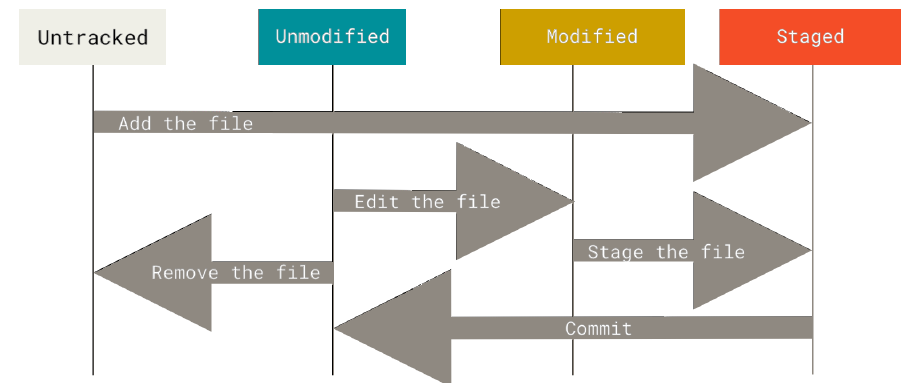
## **Initializing a Git Repository in an Existing Directory**

* $ cd /home/user/my\_project
* $ git init

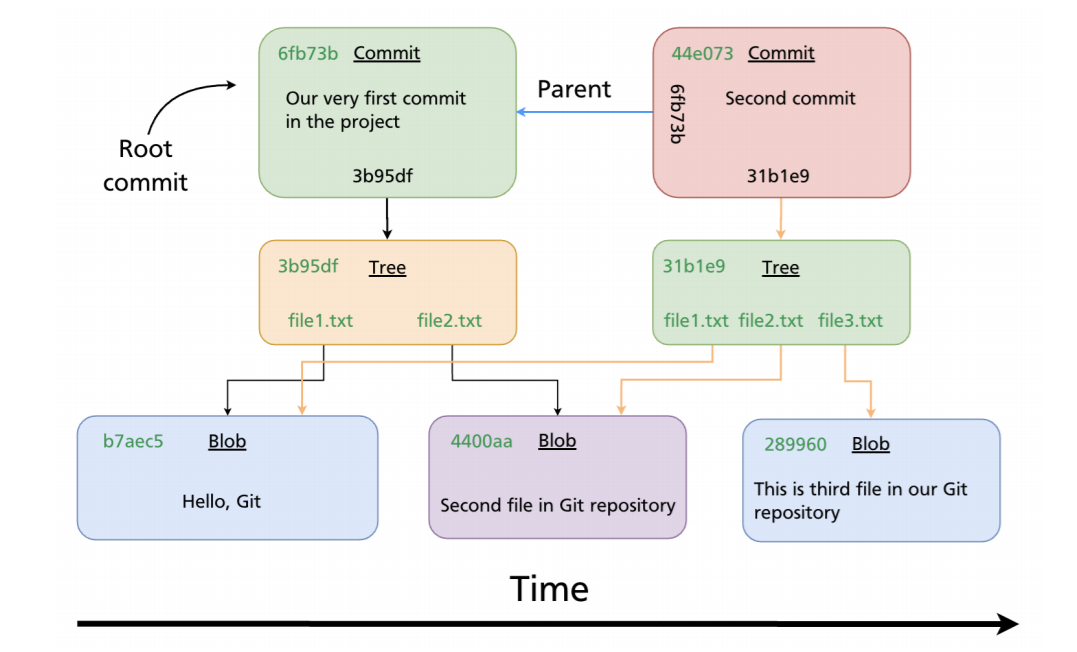
This creates a new subdirectory named .git that contains all of your necessary repository files.

## **Recording Changes to the Repository**

* Each file in your working directory can be in one of two states: tracked or untracked.
* **Tracked** files are files that were in the last snapshot; they can be unmodified, modified, or staged. In short, tracked files are files that Git knows about.
* **Untracked** files are everything else — any files in your working directory that were not in your last snapshot and are not in your staging area.
* When you first clone a repository, all of your files will be tracked and unmodified because Git just checked them out and you haven’t edited anything



## **What is commit ?**



## **Git Initialization**

* $git init

## **Git Basic configuration**

$ git config --global user.name "John Doe"

$ git config --global user.email [johndoe@example.com](mailto:johndoe@example.com)

$ git config --list

## **Staging [Tracking new files]**

* $git add . or -A
* $git add <filename>
* $git add -u updated only

-A add changes from all tracked and untracked files

-n, --dry-run dry run

-v, --verbose be verbose

## **Commiting your changes**

* $git commit –m “<commit msg>” filename or .
* $git commit –a –m “<commit msg>”

[Skips staging for **all modified files**]

## **Checking the Status of Your Files**

$ git status

$ git status –s or git status --short

* **M** README
* **MM** Rakefile
* **A** lib/git.rb [New /untracked file are staged]
* **M** lib/simplegit.rb [Modified /Tracked commited files]
* **??** LICENSE.txt [New files/ untracked]

There are two columns to the output — the lefthand column indicates the status of the staging area and the right-hand column indicates the status of the working tree.

## **Ignoring Files**

* Generally automatically generated files such as log files or files produced by your build system can be ignore if needed. In such cases, you can create a file listing patterns to match them named **.gitignore**
* # ignore all .a files

**\*.a**

* # but do track lib.a, even though you're ignoring .a files above

**!lib.a**

* # only ignore the TODO file in the current directory, not subdir/TODO

**/TODO**

* # ignore all files in any directory named build

**build/**

* # ignore doc/notes.txt, but not doc/server/arch.txt

**doc/\*.txt**

* # ignore all .pdf files in the doc/ directory and any of its subdirectories

**doc/\*\*/\*.pdf**

## **Viewing Your Staged and Unstaged Changes [Git diff]**

If the git status command is too vague for you — you want to know exactly what you changed, not just which files were changed — you can use the git diff command.

git diff shows you the exact lines added and removed — the patch, as it were.

**Working Directory VS Staging**

* $ git diff
* $ git difftool

**Staging vs Local Repo**

* $ git diff --staged or cached
* $ git difftool --staged or cached
* $ git diff --check [identifies possible whitespace]

## **Removing Files**

To remove a file from Git, you have to remove it from your tracked files (more accurately, remove it from your staging area) and then commit.

The git rm command does that, and also removes the file from your working directory so you don’t see it as an untracked file the next time around.

* $git rm -rvf <filename> [Deletes file from staged + working dir] [changes need to be commited]

If you simply remove the file from your working directory, it shows up under the “Changes not staged for commit” (that is, *unstaged*) area of your git status output.

* $git rm --cached <filename> [remove file from staging + keeps it in work dir] changes need to be commited]
* $rm –rvf <filename> [remove file from staging + keeps it in work dir] [Changes yet to be staged] [u has use git add . so That file is deleted frm staging as well]

## **Moving Files [Renaming]**

* $ git mv file\_from file\_to

## **Viewing the Commit History [log]**

* lists the commits made in that repository [most recent commits show up first]

$ git log

$git log –p [which shows the differenceintroduced in each commit]

$git log –p -2 [shows onle last 2 entry]

$git log --stat [Prints each commit entry]

$git log –stat --pretty [changes log output format]

$git log --oneline

$git log --oneline –decorate --graph –all

$ git log featureA..origin/featureA

[it will print out the history of your commits, showing where your branch pointers are and how your history has diverged]

$ git log --pretty=oneline

$git log --grapgh

$git log --since=2.weeks [until]

## **Undoing Things**

**Amend**

* $ git commit --amend -m “updated”

* The git commit --amend command is a convenient way to modify the most recent commit. It lets you combine staged changes with the previous commit instead of creating an entirely new commit.
* It can also be used to simply edit the previous commit message without changing its snapshot. But, amending does not just alter the most recent commit, it replaces it entirely, meaning the amended commit will be a new entity with its own ref. To Git, it will look like a brand new commit
* Amended commits are actually entirely new commits and the previous commit will no longer be on your current branch

.

### Don’t amend public commits

**Undo changes in Working tree [Unmodifying a Modified File]**

* $git restore <file> or $ git checkout -- <file>

**Unstaging a Staged File**

* $git restore --staged <file> or $git reset HEAD <filename>

## **Working with Remotes**

**Showing Your Remotes**

* $ git remote
* $ git remote –v [with url]

**Inspecting a Remote**

* $ git remote show <remote name>

**Adding Remote Repositories**

* git clone command implicitly adds the origin remote for you
* $git remote add <remote name> <url>

**Renaming and Removing Remote names**

* $ git remote rename pb paul
* $ git remote remove paul

## **Tracking Branches**

* Tracking branches are local branches that have a direct relationship to a remote branch. If you’re on a tracking branch and type git pull, Git automatically knows which server to fetch from and which branch to merge in.
* When you clone a repository, it generally automatically creates a master branch that tracks origin/master.
* However, you can set up other tracking branches if you wish — ones that track branches on other remotes, or don’t track the master branch.

**$ git branch -u origin/serverfix or**

**$ git branch –set-upstream-to origin/serverfix or**

**$ git branch –track origin/serverfix**

**$ git branch –vv**

If you want to see what tracking branches you have set up, you can use the -vv option to git branch.

## **Git Push**

* $ git push By default it push the upstream branch [master]
* $ git push <remote name> <branch name>
* $ git push origin master
* $ git push <remote name> <local branch name>:<remote banch name>
* $ git push origin feature1:feature1

*If feature1 not exist in remote then it will create it in remote & then push your changes.*

$ git push origin feature1:master

*Also can be pushed to already existing branch as well.*

## **Git FETCH**

**$ git fetch <remote\_name>**

Fetch all of the branches from the repository. This also downloads all of the required commits and files from the other repository.

**$ git fetch <remote\_name> <branch>**

Same as the above command, but only fetch the specified branch

**$git fetch –all**

A power move which fetches all registered remotes and their branches

**$git fetch –dry-run**

**$git log –oneline master..origin/master**

**$git log origin/master**

## **Git Pull**

$ git pull <remote\_name>

* Fetch the specified remote’s copy of the current branch and immediately merge it into the local copy. This is the same as git fetch ＜remote＞ followed by git merge origin/＜current-branch＞

$ git pull –no-commit <remote>

* Similar to the default invocation, fetches the remote content but does not create a new merge commit.

**$ git pull –rebase <remote>**

* Same as the previous pull Instead of using git merge to integrate the remote branch with the local one, use [git rebase](https://www.atlassian.com/git/tutorials/rewriting-history/git-rebase).
* While the git fetch command will fetch all the changes on the server that you don’t have yet, it will not modify your working directory at all. It will simply get the data for you and let you merge it yourself.
* However, there is a command called git pull which is essentially a git fetch immediately followed by a git merge in most cases

## **Tagging**

Git has the ability to tag specific points in a repository’s history as being important. Typically, people use this functionality to mark release points (v1.0, v2.0 and so on).

**Listing Your Tags**

* $ git tag
* $ git tag –l
* $ $ git tag -l "v1.8.5\*" [particular version]
* $ git show <tag name>

**Creating Tags**

Git supports two types of tags: *lightweight* and *annotated*

A lightweight tag is very much like a branch that doesn’t change — it’s just a pointer to a specific commit.

Annotated tags, however, are stored as full objects in the Git database. They’re checksummed; contain the tagger name, email, and date; have a tagging message; and can be signed and verified with GNU Privacy Guard (GPG).

**Annotated Tags**

* **$ git tag –a <tag name> -m <tag msg>**
* **$** git tag -a v1.4 -m "my version 1.4"

**Lightweight Tags**

* $ git tag <tagname>
* $ git show <tag name>

**Tagging Later**

* **$ git tag –a <tag name> <commit id>**

**Sharing Tags**

By default, the git push command doesn’t transfer tags to remote servers. You will have to explicitly push tags to a shared server after you have created them. This process is just like sharing remote branches — you can run git push origin <tagname>.

$ git push <remote name> <tag name>

$ git push origin v1.5

$ git push origin --tags [push all tags]

**Deleting Tags**

To delete a tag on your local repository, you can use git tag -d <tagname>.

$ git tag -d v1.4-lw

Remote tag delete

$ git push origin --delete <tagname>

**Checking out Tags**

If you want to view the versions of files a tag is pointing to, you can do a git checkout of that tag, although this puts your repository in “detached HEAD” state, which has some ill side effects.

In “detached HEAD” state, if you make changes and then create a commit, the tag will stay the same, but your new commit won’t belong to any branch and will be unreachable, except by the exact commit hash. Thus, if you need to make changes — say you’re fixing a bug on an older version, for instance — you will generally want to create a branch.

$ git checkout -b version2 v2.0.0

Switched to a new branch 'version2'

## **Git Aliases**

$ git config --global alias.co checkout

$ git config --global alias.br branch

$ git config --global alias.ci commit

$ git config --global alias.st status

This means that, for example, instead of typing git commit, you just need to type git ci. As you go on using Git, you’ll probably use other commands frequently as well; don’t hesitate to create new aliases.