Git

Notes by rajeeshvengalapurath@gmail.com

Introduction

- Git doesn't store data as a series of changesets or differences, but instead as a series of snapshots

Initializing a Repository in an Existing Directory

```
cd /home/user/my_project
git init
```

Start version-controlling existing files

Begin tracking those files and do an initial commit

```
git add *.c
git add myfile.ts
git commit -m 'Initial project version'
```

Cloning an Existing Repository

```
git clone https://github.com/rajeesh-vp/raptoroid
```

Checking the Status of Your Files

```
git status
```

Short Status

```
git status -s or git status --short

?? file2.txt (New files that aren't tracked)
A  file3.txt (New files that have been added to the staging area)
M  folder1/file4.txt (Modified files)
AM modified.txt (Staged then modified)
A  untracked.txt
```

Tracking New Files

```
git add myfile.txt
```

File is now tracked and staged to be committed

Modified Files

```
Will go under a section: “Changes not staged for commit”
Modified in the working directory but not yet staged.
```

To stage Files

```
git add modifiedfile.txt
```

git add command

git add is a multipurpose command. It may be helpful to think of it more as “add precisely this content to the next commit” rather than “add this file to the project”.

- use it to begin tracking new files
- to stage files
- and to do other things like marking merge-conflicted files as resolved.

Viewing Staged and Unstaged Changes

`git diff`	To see what you’ve changed but not yet staged
`git diff --staged`	To see what you’ve staged that will go into your next commit
`git diff --cached`	To see what you’ve staged so far

Difference between two commits

```
git diff d30f535971c048f2774628e0db36e5dd0f5844a3
ecf6b4468158850b616492c3d4960d3d762da3a4
```

Committing Changes

```
git commit -m "first commit"
```

Skipping the Staging Area

The staging area is sometimes a bit more complex than you need in your workflow. If you want to skip the staging area, adding the -a option to the git commit command makes Git

automatically stage every file that is already tracked before doing the commit, letting you skip the git add part.

Commit modified files without staging (without `git add`)

```
git commit -a -m "first commit"
```

Git Log (Viewing the Commit History)

```
git log
```

git log -p or **git --patch** (Shows the difference (the patch output) introduced in each commit)

git log -p -2 (show only the last two entries)

git log --stat (To see some abbreviated stats for each commit)

git log -S function_name (Takes a string and shows only those commits that changed the number of occurrences of that string)

git log --pretty=format:"%h - %an, %ar : %s" (To specify your own log output format)

Specifier	Description of Output
%H	Commit hash
%h	Abbreviated commit hash
%T	Tree hash
%t	Abbreviated tree hash
%P	Parent hashes
%p	Abbreviated parent hashes
%an	Author name
%ae	Author email
%ad	Author date (format respects the --date=option)
%ar	Author date, relative
%cn	Committer name
%ce	Committer email
%cd	Committer date
%cr	Committer date, relative
%s	Subject

Output:

```
d30f535 - Rajeesh VP, 4 days ago : get test
ecf6b44 - Rajeesh VP, 4 days ago : Add project files.
6d9c3b2 - Rajeesh VP, 4 days ago : Add .gitignore and .gitattributes.
```

```
git log
       -<n>   Show only the last n commits
       --since, --after    Limit the commits to those made after the specified
                            date.
       --until, --before   Limit the commits to those made before the specified
                           date.
```

```
    --author            Only show commits in which the author entry matches the
                         specified string.
    --committer         Only show commits in which the committer entry matches
                         the specified string.
    --grep              Only show commits with a commit message containing the
                         string
    -S                  Only show commits adding or removing code matching the
                         string
```

```
git log --all --graph --oneline
git log --oneline --decorate --graph --all
git log --oneline --decorate --graph --all --pretty=format:"%h - %an, %ar : %s"
```

Git Shortlog

To quickly get a sort of changelog of what has been added to your project since your last release

```
git shortlog --no-merges master --not v1.0.1
```

Author vs Committer

The author is the person who originally wrote the work, whereas the committer is the person who last applied the work

Undoing Things

If you want to redo that commit, make the additional changes you forgot, stage them, and commit again

```
git commit --amend
```

Example

```
git commit -m 'Initial commit'
git add forgotten_file
git commit --amend
```

Unstaging a Staged File

```
git reset HEAD <file>
```

Unmodifying a Modified File

```
git checkout -- <file>
```

Tags

The tag object contains a tagger, a date, a message, and a pointer. The main difference is that a tag object generally points to a commit rather than a tree. It's like a branch reference, but it never moves — it always points to the same commit but gives it a friendlier name.

Lightweight tag

```
$ git update-ref refs/tags/v1.0 cac0cab538b970a37ea1e769cbbde608743bc96d
```

Annotated tag

An annotated tag is more complex, however. If you create an annotated tag, Git creates a tag object and then writes a reference to point to it rather than directly to the commit. You can see this by creating an annotated tag (using the -a option)

```
$ git tag -a v1.1 1a410efbd13591db07496601ebc7a059dd55cfe9 -m 'Test tag'
```

Remotes

Remote commit has the same hash that of local.

Showing Remotes

```
$ git remote -v
origin https://github.com/rajeesh-vp/Raptoroid (fetch)
origin https://github.com/rajeesh-vp/Raptoroid (push)
```

Adding Remote Repositories

`git clone` command implicitly adds the origin remote. To add a new remote repository explicitly as a shortname, run

```
git remote add <shortname> <url>:
```

Fetching and Pulling from Your Remotes

git fetch

`git fetch <remote>`
Pulls down all the data from that remote project that you don't have yet.git fetch command only downloads the data to your local repository — it doesn't automatically merge it with

any of your work or modify what you're currently working on. You have to merge it manually into your work when you're ready

git pull

`git pull <remote>`
Automatically fetch and then merge that remote branch into your current branch.

Pushing to Your Remotes

```
git push <remote> <branch>
git push origin master
git push <LocalRemoteName> master
git push Raptoroid master
git push --all origin (push all branches as it is to remote for first the time)
```

Perform after commit. If you and someone else clone at the same time and they push upstream and then you push upstream, your push will rightly be rejected. You'll have to fetch their work first and incorporate it into yours before you'll be allowed to push.

Inspecting a Remote

```
git remote show origin
git remote show Raptoroid
```

Renaming and Removing Remotes

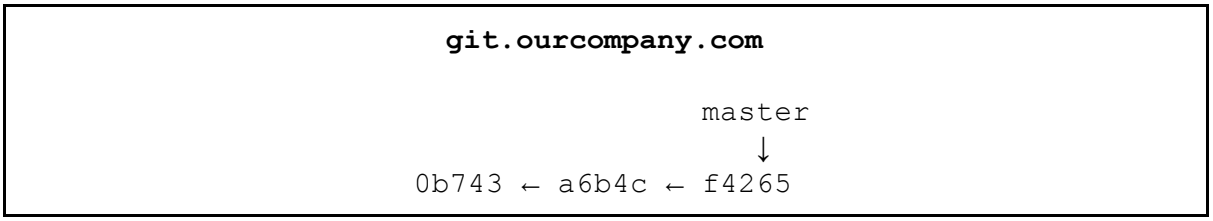
```
git remote rename pb paul
```

Remote Branches

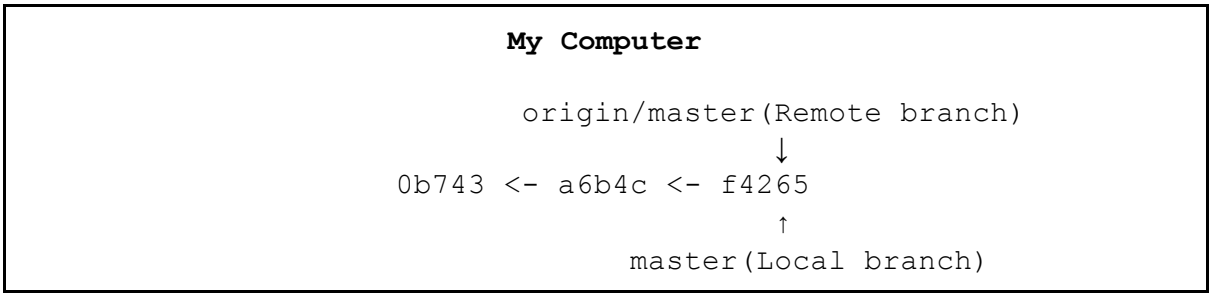
- Remote references are references (pointers) in your remote repositories, including branches, tags, and so on
- If you do some work on your local master branch, and, in the meantime, someone else pushes to git.ourcompany.com and updates its master branch, then your histories move forward differently. Also, as long as you stay out of contact with your origin server, your origin/master pointer doesn't move.
- `git fetch` updates your remote-tracking branches
- when you do a fetch that brings down new remote-tracking branches, you don't automatically have local, editable copies of them. you have only an origin/something pointer that you can't modify (but files are downloaded. Which only becomes editable on merge command)

Diagram

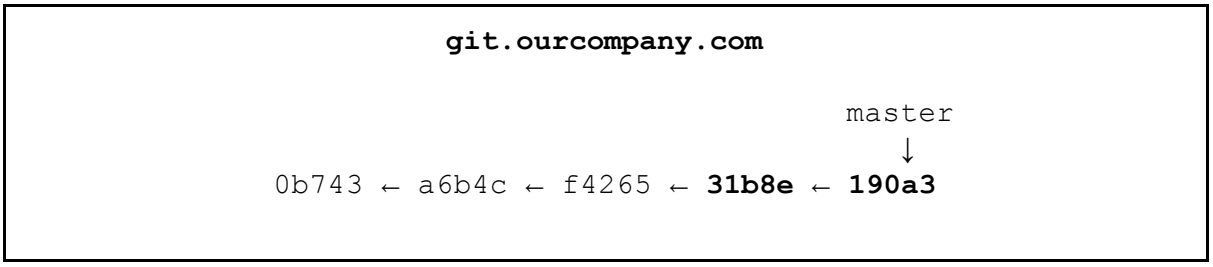
Git clone



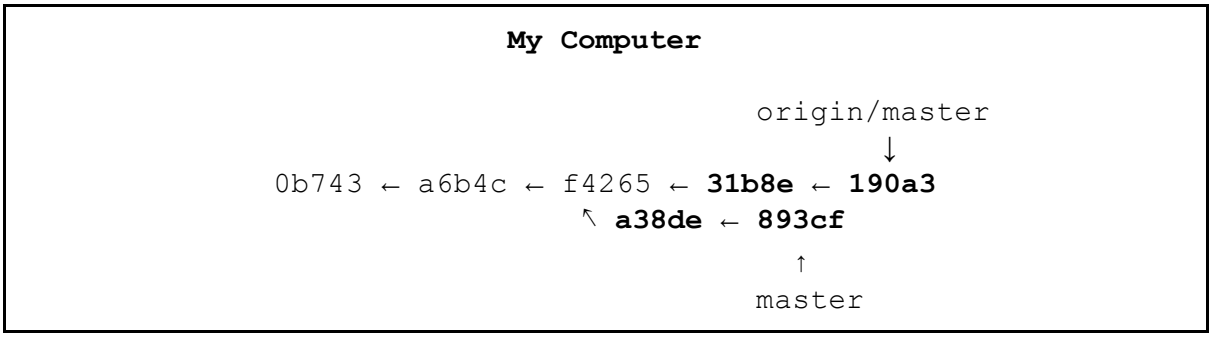
↓ git clone me@git.ourcompany.com:project.git



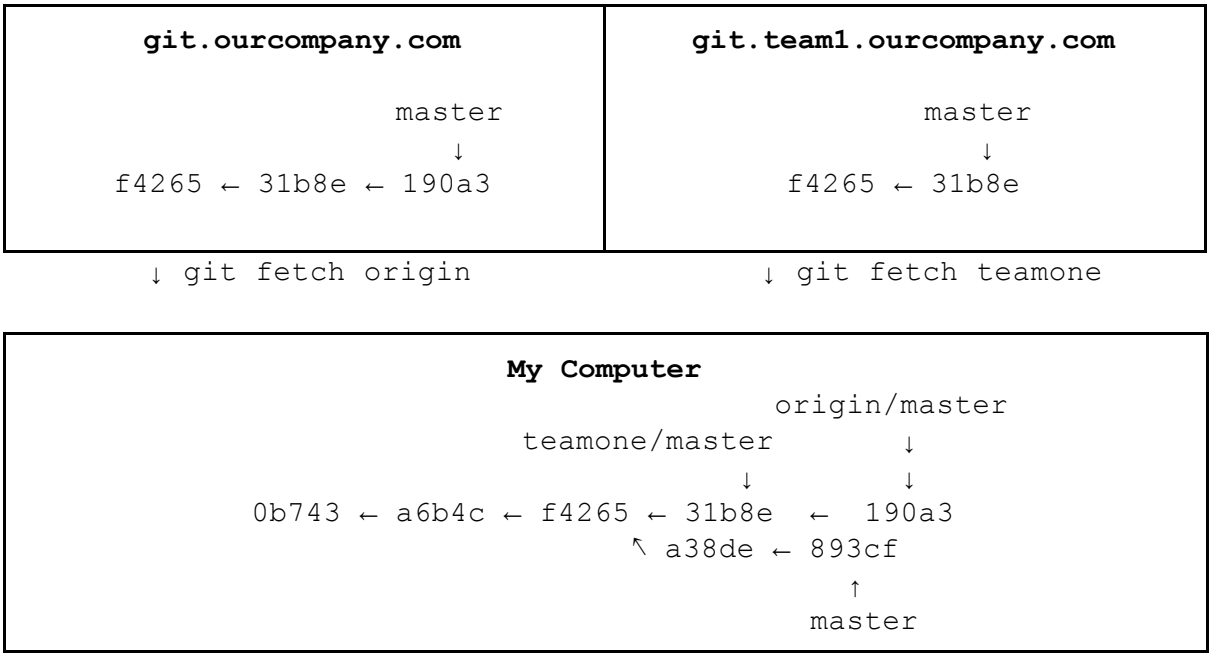
Git fetch (after updates in local and someone else's update in server)



↓ git fetch origin



Multiple Servers



Get a full list of remote references explicitly

```
git ls-remote <remote> or git remote show <remote>

Output:
d29024eb2253a1bbfdc4c8f89a862216f6d7f65b        HEAD
34492de534448ce1d127c91dbec338f037b62a75        refs/heads/branch1
93040c383b9bbeb947f5f32cf495bab20ef7967f        refs/heads/branch2
d29024eb2253a1bbfdc4c8f89a862216f6d7f65b        refs/heads/master
```

Tracking Branches

- Checking out a local branch from a remote-tracking branch automatically creates what is called a “tracking branch” (and the branch it tracks is called an “upstream branch”)
- 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

Manually creating tracking and non-tracking branches

```
git checkout --track origin/serverfix
```

Pushing

```
$ git push origin serverfix
```

Pulling

- While the `git fetch` command will simply get the data for you and let you merge it yourself. `git pull` command do a `git fetch` immediately followed by a `git merge`
- Generally it’s better to simply use the fetch and merge commands explicitly as the magic of git pull can often be confusing

Checking Out Remote Branches

To test a remote branch

```
$ git remote add jessica git://github.com/jessica/myproject.git
$ git fetch jessica
$ git checkout -b rubyclient jessica/ruby-client
```

Later with another branch

```
$ git checkout -b rubyclient2 jessica/ruby-client
```

To pull and doesn't save the URL as a remote reference

```
$ git pull https://github.com/onetimeguy/project
```

Determining What Is Introduced in a branch

To show branch1's commits only

```
$ git log branch1 --not master (-p will show changes also)
```

Branching

Introduction

- The way Git branches is incredibly lightweight
- Unlike many other VCSs, Git encourages workflows that branch and merge often, even multiple times in a day
- Git doesn't store data as a series of changesets or differences, but instead as a series of snapshots
- Git doesn't store data as a series of changesets or differences, but instead as a series of snapshots
- As you start making commits, you're given a master branch that points to the last commit you made
- Every time you commit, the master branch pointer moves forward automatically.
- Creating a new branch creates a new pointer for you to move around
- Git knows the branch you're currently on by keeping a special pointer called HEAD
- Because a branch in Git is actually a simple file that contains the 40 character SHA-1 checksum of the commit it points to, branches are cheap to create and destroy. Creating a new branch is as quick and simple as writing 41 bytes to a file (40 characters and a newline).

See Current Branch

HEAD pointing to which branch can easily be seen running a simple `git log` command

```
commit d8f972db6aee3c50648771bb9b28b69efab165a8 (HEAD -> branch1, master)
Author: Rajeesh VP <rajeeshvp@gmail.com>
Date:   Thu Aug 20 11:19:29 2020 +0530
```

Switching Branches

1. **git checkout** branch1
2. $ git checkout -b iss53
   Switched to a new branch "iss53"

   (This is shorthand for:)

```
$ git branch iss53
$ git checkout iss53
```

Delete Branch

```
git branch -d branch1
```

git branch -d will fail if the branch is never merged

```
$ git branch -d testing
error: The branch 'testing' is not fully merged.
If you are sure you want to delete it, run 'git branch -D testing'.
```

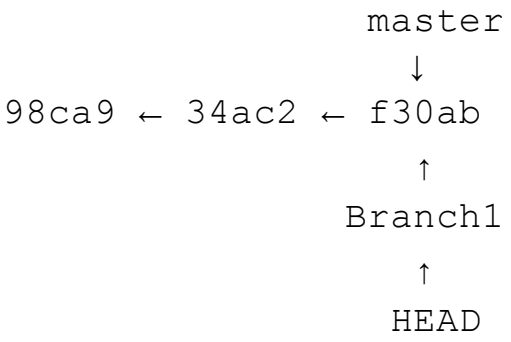
Changing a branch name

- Do not rename branches that are still in use by other collaborators.
- Do not rename a branch like master/main/mainline without having read the section "Changing the master branch name".

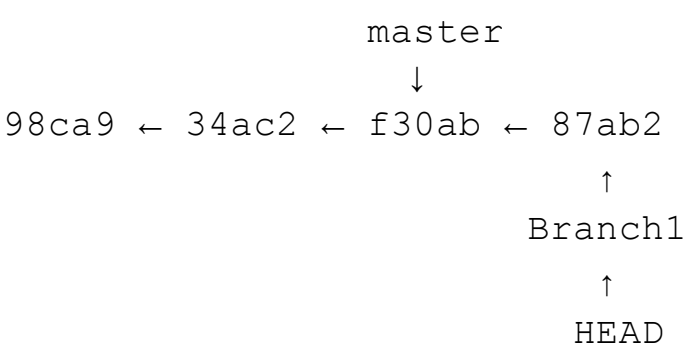
```
$ git branch --move bad-branch-name corrected-branch-name
$ git push --set-upstream origin corrected-branch-name
```

Branch Diagram

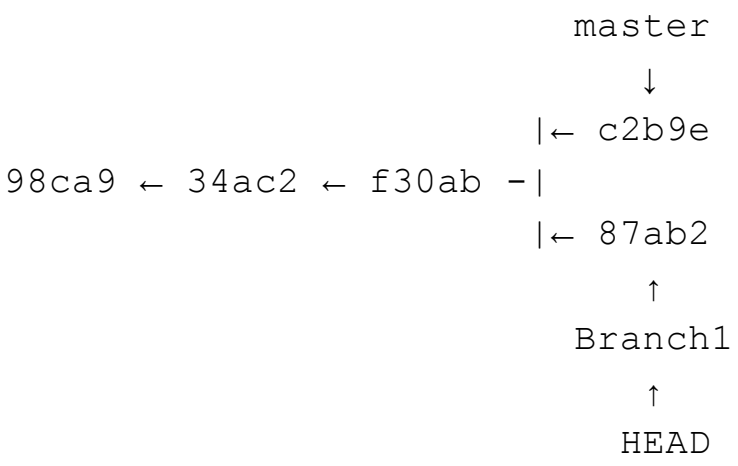
Create new branch



New commit on new branch



New commit on master branch

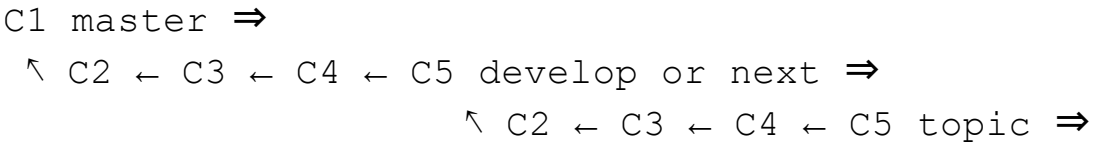


Branch from a previous commit

```
git branch newBranchName <sha1-of-commit>
```

Long-Running Branches

https://git-scm.com/book/en/v2/Git-Branching-Branching-Workflows



Many Git developers have a workflow that embraces this approach, such as having only code that is entirely stable in their **master** branch — possibly only code that has been or will be released. They have another parallel branch named **develop** or next that they work from or use to test stability — it isn't necessarily always stable, but whenever it gets to a stable state, it can be merged into master. It's used to pull in topic branches **(short-lived branches, like your earlier iss53 branch)** when they're ready, to make sure they pass all the tests and don't introduce bugs.

Topic Branches

https://git-scm.com/book/en/v2/Git-Branching-Branching-Workflows

A topic branch is a **short-lived branch** that you create and use for a single particular feature or related work. This is something you've likely never done with a VCS before because it's generally too expensive to create and merge branches

Rebase

You have two options for integrating your feature into the master branch: merging directly or rebasing and then merging. The former option results in a 3-way merge and a merge commit, while the latter results in a fast-forward merge and a perfectly linear history. Rebasing is like saying, “I want to base my changes on what everybody has already done.”

Git on the Server

A remote repository is generally a bare repository — a Git repository that has no working directory. Because the repository is only used as a collaboration point, there is no reason to have a snapshot checked out on disk; it’s just the Git data. In the simplest terms, a bare repository is the contents of your project’s .git directory and nothing else.

Local Protocol

Local protocol, in which the remote repository is in another directory on the same host. This is often used if everyone on your team has access to a shared filesystem such as an NFS mount. (Shared filesystem)

```
$ git clone /srv/git/project.git
$ git clone file:///srv/git/project.git
```

The HTTP Protocols

```
$ git clone https://example.com/gitproject.git
```

The SSH Protocol

```
$ git clone ssh://[user@]server/project.git
$ git clone [user@]server:project.git
```

Getting Git on a Server

```
$ git clone --bare my_project my_project.git
```

Create a bare version, and place it on a server to which you and your collaborators have SSH access

GitHub

Forking Projects

When you “fork” a project, GitHub will make a copy of the project that is entirely yours; it lives in your namespace, and you can push to it and contribute your changes back to the

original repository by creating what’s called a Pull Request. GitHub is designed around a particular collaboration workflow, centered on Pull Requests