Learn Git and Version Control from Scratch

Version Control

Version control is a software system that manages changes to computer code or any other set of files over time. It allows multiple people to collaborate on the same files, keep track of changes, and revert to earlier versions of the files if necessary.

Types of version Control:

1. Git
2. Subversion
3. Mercurial

There are two main types of version control system models:

1. Centralised: Here all the users connect to the central, master repository.
2. Distributed: Each user has the entire repository on their computer.

Git is a distributed version control system.

Version Control System / Source Code Manager

A **version control system** (abbreviated as **VCS**) is a tool that manages different versions of source code. A **source code manager** (abbreviated as **SCM**) is another name for a version control system.

Git terms pdf ::
https://video.udacity-data.com/topher/2017/March/58d31eb5_ud123-git-keyterms/ud123-git-keyterms.pdf

Git Commands:

Git init: To create a new repository with Git, we'll use the `git init` command. it's the command that will do all of the initial setup of a repository.

Commands required:

- `ls` - used to list files and directories
- `mkdir` - used to create a new directory
- `cd` - used to change directories
- `rm` - used to remove files and directories

Command: `git init`

`Git clone:` The `git clone` command is used to create an identical copy of an existing repository. git clone is primarily used to point to an existing repo and make a clone or copy of that repo in a new directory, at another location.

Command: `git clone repo link`

`Ex: git clone https://github.com/udacity/course-git-blog-project`

Clone with a different name than default:

`git clone repo-link new-repo-name-we-want`

Ex: `git clone https://github.com/udacity/course-git-blog-project blog-app`

`Git Status:` The `git status` command will display the current status of the repository.

Command:

```
git status
```

Git status output:

```
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

This output comes when there are no changes pending to commit.

Git log: The `git log` command is used to display all of the commits of a repository. press `q` to **quit** out of the log (returns to the regular command prompt)

Command: `git log`

By default, this command displays:

1. the SHA
2. the author
3. the date
4. and the message

Git log –oneline:
This command:

1. lists one commit per line
2. shows the first 7 characters of the commit SHA
3. shows the commit's message

Command: `git log --oneline`

Git log –stat: The `git log` command has a flag that can be used to display the files that have been changed in the commit, as well as the number of lines that have been added or deleted.
This command:

- displays the file(s) that have been modified
- displays the number of lines that have been added/removed
- displays a summary line with the total number of modified files and lines that have been added/removed

Command: `git log --stat`

Git log -p: The `git log` command has a flag that can be used to display the actual changes made to a file. The flag is `--patch` which can be shortened to just `-p`:
Command: `git log -p`

Git log -p SHA: By supplying a SHA, the `git log -p` command will *start at that commit.*
Command: `git log -p SHA`

git show: Running it like the example above will only display the most recent commit. Typically, a SHA is provided as a final argument. The `git show SHA` command will show *only one commit*.

So by default, `git show` displays:

- the commit
- the author
- the date
- the commit message
- the patch information

git add: This command is used to move files to the staging index.
Command to add a single file: `git add index.html`
Command to add more than one files: `git add css/app.css js/app.js`
Command to add all the untracked files: `git add .`

git rm –cached: This command is used to remove files from the staging index.
Command: `git rm --cached files...`

git commit: This command is used to commit the staged files.
Command: `git commit -m "Commit message"`

git diff: This command shows changes which were made but not committed yet.
Command: `git diff →` This will show all the changes made in the project.
Command: `git diff fileName` This will show all the changes made in the specified file.

.gitignore file: Add all the files and folders which we want to ignore at the time of commit to this file in project structure.

git Tag: This is used to provide tags and versions to commits.
Add tag to current commit: `git tag -a v1.0` , Always use -a because it contains more information. This will open your code editor and wait for you to supply a message for the tag.
Verify tag: `git tag` This command is used to find the current version of an app.
Adding A Tag To A Past Commit: `git tag -a v1.0 SHA`

git branch: The `git branch` command is used to interact with Git's branches:
It can be used to:

- list all branch names in the repository
- create new branches
- delete branches

Command to list all branches: `git branch`
Command to create a new branch: `git branch branchName`
Command to change the branch: `git checkout branchName`
Command to delete a branch: `git branch -d branchName`
Command to create a new branch using checkout: `git checkout -b branchName`
Command to create a new branch and start from a specific branch: `git checkout -b NewBrach Specific Branch.`
Command to show graph of all the commits: `git log --oneline --decorate --graph --all`

Merging: Combining branches together is called **merging**. There are two main types of merges in Git, a regular **merge** and a **Fast-forward merge**.

Command to undo the merge changes: `git reset --hard HEAD^`

`git merge` command is used to merge the branches.
Command: `git merge <name-of-branch-to-merge-in>`

When a merge happens, Git will:

- look at the branches that it's going to merge
- look back along the branch's history to find a single commit that *both* branches have in their commit history
- combine the lines of code that were changed on the separate branches together
- makes a commit to record the merge

when a merge is performed, the *other* branch's changes are brought into the branch that's currently checked out.
Let me stress that again - When we merge, we're merging some other branch into the current (checked-out) branch. We're not merging two branches into a new branch. We're not merging the current branch into the other branch.

Fast-forward merge: Now, since `footer` is directly ahead of `master`, this merge is one of the easiest merges to do. Merging `footer` into `master` will cause a **Fast-forward merge**. A Fast-forward merge will just move the currently checked out branch *forward* until it points to the same commit that the other branch (in this case, `footer`) is pointing to.

Regular merge: To merge in the `sidebar` branch, make sure you're on the `master` branch and run: `git merge branchName`
Because this combines two divergent branches, a commit is going to be made. And when a commit is made, a commit message needs to be supplied. Since this is a *merge commit* a default message is already supplied. You can change the message if you want, but it's common practice to use the default merge commit message. So when your code editor opens with the message, just close it again and accept that commit message.

Merge Conflicts: Most of the time Git will be able to merge branches together without any problem. However, there are instances when a merge cannot be *fully* performed automatically. When a merge fails, it's called a merge conflict.

A merge conflict occurs when *the exact same line(s) are changed in separate branches*.

Merge Conflict Indicators Explanation

The editor has the following merge conflict indicators:

- `<<<<<<< HEAD` everything below this line (until the next indicator) shows you what's on the current branch
- `||||||| merged common ancestors` everything below this line (until the next indicator) shows you what the original lines were
- `=======` is the end of the original lines, everything that follows (until the next indicator) is what's on the branch that's being merged in
- `>>>>>>> heading-update` is the ending indicator of what's on the branch that's being merged in (in this case, the `heading-update` branch)

Command to abort the changes after a merge conflict: `git merge —abort`

Modify the last commit:

You've already made plenty of commits with the `git commit` command. Now with the `--amend` flag, you can alter the *most-recent* commit.
Command to amend the changes in last commit: `git commit --amend`
Forgot to add the files to commit:
Alternatively, `git commit --amend` will let you include files (or changes to files) you might've forgotten to include. Let's say you've updated the color of all navigation links across your entire website. You committed that change and thought you were done. But then you discovered that a special nav link buried deep on a page doesn't have the new color. You *could* just make a new commit that updates the color for that one link, but that would have two back-to-back commits that do practically the exact same thing (change link colors). Instead, you can amend the last commit (the one that updated the color of all of the other links) to include this forgotten one. To do get the forgotten link included, just:

- edit the file(s)
- save the file(s)
- stage the file(s)
- and run `git commit --amend`

So you'd make changes to the necessary CSS and/or HTML files to get the forgotten link styled correctly, then you'd save all of the files that were modified, then you'd use `git add` to stage all of the modified files (just as if you were going to make a new commit!), but then you'd run `git commit --amend` to update the most-recent commit instead of creating a new one.

Reverting A Commit

When you tell Git to **revert** a specific commit, Git takes the changes that were made in commit and does the exact opposite of them. Let's break that down a bit. If a character is added in commit A, if Git *reverts* commit A, then Git will make a new commit where that character is deleted. It also works the other way where if a character/line is removed, then reverting that commit will *add* that content back!

Command for git revert: `git revert <SHA-of-commit-to-revert>`

This command:

- will undo the changes that were made by the provided commit
- creates a new commit to record the change

Resetting Commits

At first glance, *resetting* might seem coincidentally close to *reverting*, but they are actually quite different. Reverting creates a new commit that reverts or undos a previous commit. Resetting, on the other hand, *erases* commits!

Command for reset: `git reset <reference-to-commit>`

It can be used to:

- move the HEAD and current branch pointer to the referenced commit
- erase commits
- move committed changes to the staging index
- unstage committed changes