Git

*$ git clone https://github.com/udacity/course-git-blog-project*

If you want to clone the repository into a directory named something other than libgit2, you can specify the new directory name as an additional argument:

$ git clone https://github.com/libgit2/libgit2 mylibgit

GIT Status

$ git status

To check the status of the file

**Git Log Recap**

Fantastic job! Do you feel your Git-power growing?

Let's do a quick recap of the git log command. The git log command is used to display all of the commits of a repository.

$ git log

By *default*, this command displays:

* the SHA
* the author
* the date
* and the message

...of every commit in the repository. I stress the "By default" part of what Git displays because the git log command can display a lot more information than just this.

Git uses the command line pager, Less, to page through all of the information. The important keys for Less are:

* to scroll down by a line, use j or ↓
* to scroll up by a line, use k or ↑
* to scroll down by a page, use the spacebar or the Page Down button
* to scroll up by a page, use b or the Page Up button
* to quit, use q

We'll increase our git log-wielding abilities in the next lesson when we look at displaying more info.

We've been looking closely at all the detailed information that git log displays. But now, take a step back and look at all of the information as a whole.

Let's think about some of these questions:

* **the SHA** - git log will display the complete SHA for every single commit. Each SHA is unique, so we don't really need to see the *entire* SHA. We could get by perfectly fine with knowing just the first 6-8 characters. Wouldn't it be great if we could save some space and show just the first 5 or so characters of the SHA?
* **the author** - the git log output displays the commit author for *every single commit*! It could be different for other repositories that have multiple people collaborating together, but for this one, there's only one person making all of the commits, so the commit author will be identical for all of them. Do we need to see the author for each one? What if we wanted to hide that information?
* **the date** - By default, git log will display the date for each commit. But do we really care about the commit's date? Knowing the date might be important occasionally, but typically knowing the date isn't vitally important and can be ignored in a lot of cases. Is there a way we could hide that to save space?
* **the commit message** - this is one of the most important parts of a commit message...we usually always want to see this

What could we do here to not waste a lot of space and make the output smaller? We can use a **flag**.

*TIP: This isn't a course on the command line, but a flag is used to alter how a program functions. For example, the ls command will list all of the files in the current directory. The ls command has a -l flag (i.e. ls -l) that runs the same ls command but alters how it works; it now displays the information in the long format (the -l for long).*

*Flags can be used to alter how a program functions and/or what is displayed. To learn more about command line programs and flags, check out our course*[*Linux Command Line Basics*](https://www.udacity.com/course/linux-command-line-basics--ud595)*.*

**git log --oneline**

The git log command has a flag that can be used to alter how it displays the repository's information. That flag is --oneline:

$ git log --oneline

The flag is --stat ("stat" is short for "statistics"):

$ git log --stat

**git log --stat Recap**

To recap, the --stat flag is used to alter how git log displays information:

$ git log --stat

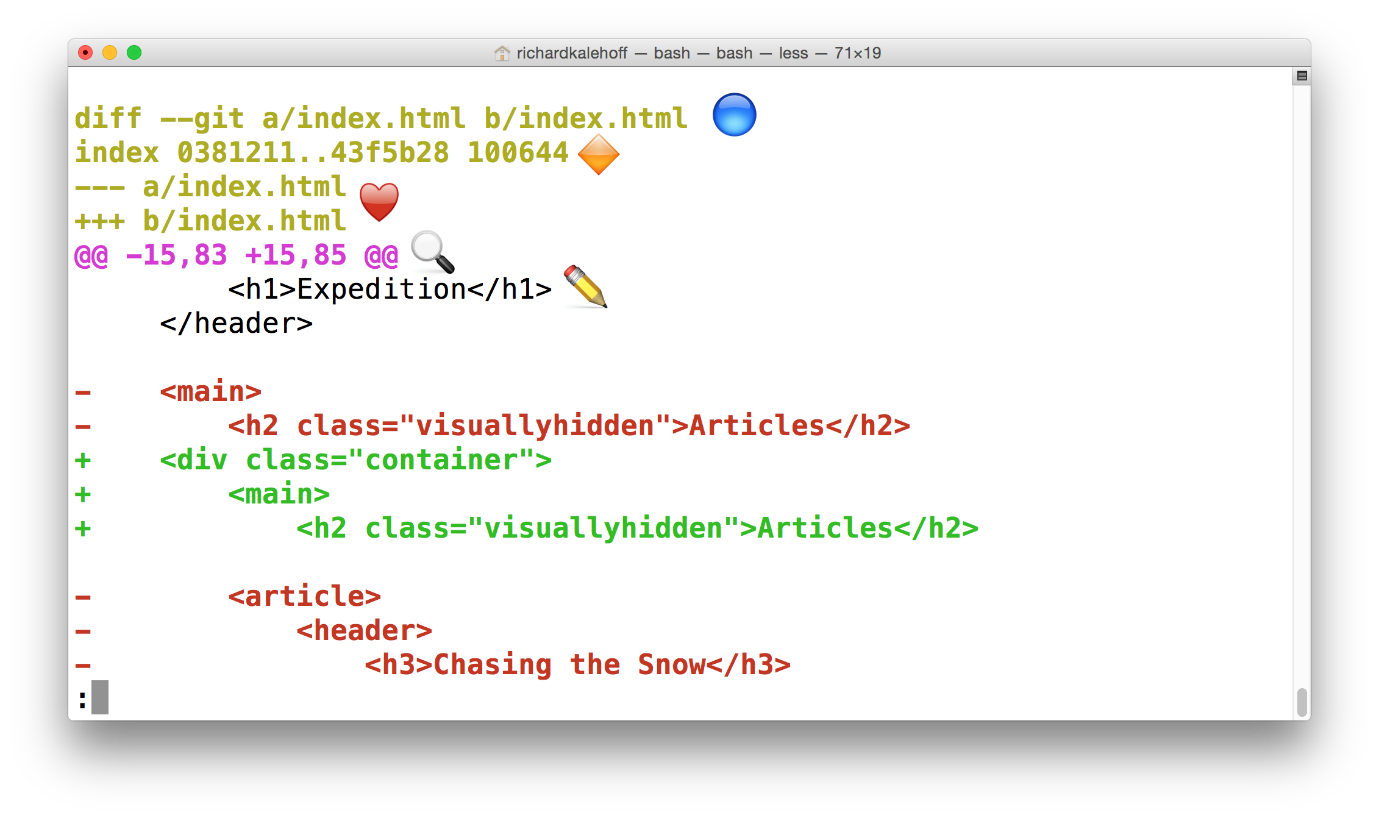
This command:

* 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

## 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:

$ git log -p



**Annotated git log -p Output**

Using the image above, let's do a quick recap of the git log -p output:

* 🔵 - the file that is being displayed
* 🔶 - the hash of the first version of the file and the hash of the second version of the file
  + not usually important, so it's safe to ignore
* ❤️ - the old version and current version of the file
* 🔍 - the lines where the file is added and how many lines there are
  + -15,83 indicates that the old version (represented by the -) started at line 15 and that the file had 83 lines
  + +15,85 indicates that the current version (represented by the +) starts at line 15 and that there are now 85 lines...these 85 lines are shown in the patch below
* ✏️ - the actual changes made in the commit
  + lines that are red and start with a minus (-) were in the original version of the file but have been removed by the commit
  + lines that are green and start with a plus (+) are new lines that have been added in the commit

git log -p --stat  it both displays both the stats --stat (Statistics) info above the -p patch info

But did you know, you can supply the SHA of a commit as the final argument for all of these commands? For example:

$ git log -p fdf5493

$ git log –p --stat fdf5493

### What does git show do?

The git show command will show only one commit. So don't get alarmed when you can't find any other commits - it only shows one. The output of the git show command is exactly the same as the git log -p command. So by default, git show displays:

* the commit
* the author
* the date
* the commit message
* the patch information

However, git show can be combined with most of the other flags we've looked at:

* --stat - to show the how many files were changed and the number of lines that were added/removed
* -p or --patch - this the default, but if --stat is used, the patch won't display, so pass -p to add it again
* -w - to ignore changes to whitespace

GIT ADD

$ git add

To add files from working directory to the staging index

GIT COMMIT

$ git commit

Take files from index area and save them in the repository

GIT DIFF

$ git diff

Displays difference two versions of a file

## Big Picture Review

That's really cool, isn't it! We haven't done anything specific with Git just yet, but it's watching this directory (since it's a Git project), and it knows that we've created a couple of new files. What's also pretty neat about the output of the git status command is that it's telling us that the files are untracked by Git.

Let's do a quick review of what's going on and what we're about to do:

* we have some new files that we want Git to start tracking
* for Git to track a file, it needs to be committed to the repository
* for a file to be committed, it needs to be in the Staging Index
* the git add command is used to move files from the Working Directory to the Staging Index
* there are currently three, untracked files in the Working Directory
  + index.html
  + app.css in the css directory
  + app.js in the js directory

So the first step to getting any files committed to the repository is to add them from the Working Directory to the Staging Index. We will be using the git add command to move all three of these files to the Staging Index.

## Changes To Be Committed

There's now a new section in the output of git status - the "Changes to be committed" area! This new "Changes to be committed" section displays files that are in the Staging Area! Right now it only displays the index.html file, so this file is the only item on the Staging Index. To continue this train of thought, if we made a commit right now, only the index.html file would be committed.

*TIP: Did you also notice the helpful text that's located just beneath "Changes to be committed"? It says*

*(use "git rm --cached <file>..." to unstage)*

*This is a hint of what you should do if you accidentally ran git add and gave it the wrong file.*

*As a side note, git rm --cached is not like the shell's rm command. git rm --cached will not destroy any of your work; it just removes it from the Staging Index.*

*Also, this used the word "unstage". The act of moving a file from the Working Directory to the Staging Index is called "staging". If a file has been moved, then it has been "staged". Moving a file from the Staging Index*back*to the Working Directory will unstage the file. If you read documentation that says "stage the following files" that means you should use the git add command.*

$ git add css/app.css js/app.js

### The Period  .

The period refers to the current directory and can be used as a shortcut to refer to all files and directories (including all nested files and directories!).

$ git add css/app.css js/app.js

*# would become*

$ git add .

## Git Add Recap

The git add command is used to move files from the Working Directory to the Staging Index.

$ git add <file1> <file2> … <fileN>

This command:

* takes a space-separated list of file names
* alternatively, the period . can be used in place of a list of files to tell Git to add the current directory (and all nested files)

## *Bypass The Editor With The -m Flag*

*TIP: If the commit message you're writing is short and you don't want to wait for your code editor to open up to type it out, you can pass your message directly on the command line with the -m flag:*

*$ git commit -m "Initial commit"*

## Git Diff Recap

To recap, the git diff command is used to see changes that have been made but haven't been committed, yet:

$ git diff

This command displays:

* the files that have been modified
* the location of the lines that have been added/removed
* the actual changes that have been made

## Globbing Crash Course

Let's say that you add 50 images to your project, but want Git to ignore all of them. Does this mean you have to list each and every filename in the .gitignore file? Oh gosh no, that would be crazy! Instead, you can use a concept called [globbing](https://en.wikipedia.org/wiki/Glob_(programming" \t "_blank).

Globbing lets you use special characters to match patterns/characters. In the .gitignore file, you can use the following:

* blank lines can be used for spacing
* # - marks line as a comment
* \* - matches 0 or more characters
* ? - matches 1 character
* [abc] - matches a, b, \_or\_ c
* \*\* - matches nested directories - a/\*\*/z matches
  + a/z
  + a/b/z
  + a/b/c/z

So if all of the 50 images are JPEG images in the "samples" folder, we could add the following line to .gitignore to have Git ignore all 50 images.

samples/\*.jpg

## Git Ignore Recap

To recap, the .gitignore file is used to tell Git about the files that Git should not track. This file should be placed in the same directory that the .git directory is in.

**GIT Tag**

This will tag the most recent commit.

Or Used to add tag to specific commits

## Git Tag Command

Pay attention to what's shown (just the SHA and the commit message)

The command we'll be using to interact with the repository's tags is the git tag command:

$ git tag -a v1.0

The below command will avoid launching the editor to add tag

$ git tag -m -a v1.0

## Git Log's --decorate Flag

As you've learned, git log is a pretty powerful tool for letting us check out a repository's commits. We've already looked at a couple of its flags, but it's time to add a new one to our toolbelt. The --decorate flag will show us some details that are hidden from the default view.

Try running git log --decorate now!

## Deleting A Tag

What if you accidentally misspelled something in the tag's message, or mistyped the actual tag name (v0.1 instead of v1.0). How could you fix this? The easiest way is just to delete the tag and make a new one.

A Git tag can be deleted with the -d flag (for delete!) and the name of the tag:

$ git tag -d v1.0

$ git tag --delete v1.0

## Adding A Tag To A Past Commit

Running git tag -a v1.0 will tag the most recent commit. But what if you wanted to tag a commit that occurred farther back in the repo's history?

All you have to do is provide the SHA of the commit you want to tag!

$ git tag -m –a v1.0 a87984

**Git Tag Recap**

To recap, the git tag command is used to add a marker on a specific commit. The tag does not move around as new commits are added.

$ git tag -a beta

This command will:

* add a tag to the most recent commit
* add a tag to a specific commit *if a SHA is passed*

**GIT Branch**

Allows multiple lines of development

**The git branch command**

The git branch command is used to interact with Git's branches:

$ git branch

It can be used to:

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

## Create A Branch

To create a branch, all you have to do is use git branch and provide it the name of the branch you want it to create. So if you want a branch called "sidebar", you'd run this command:

$ git branch sidebar

**GIT Checkout**

Switch between different branches and tags

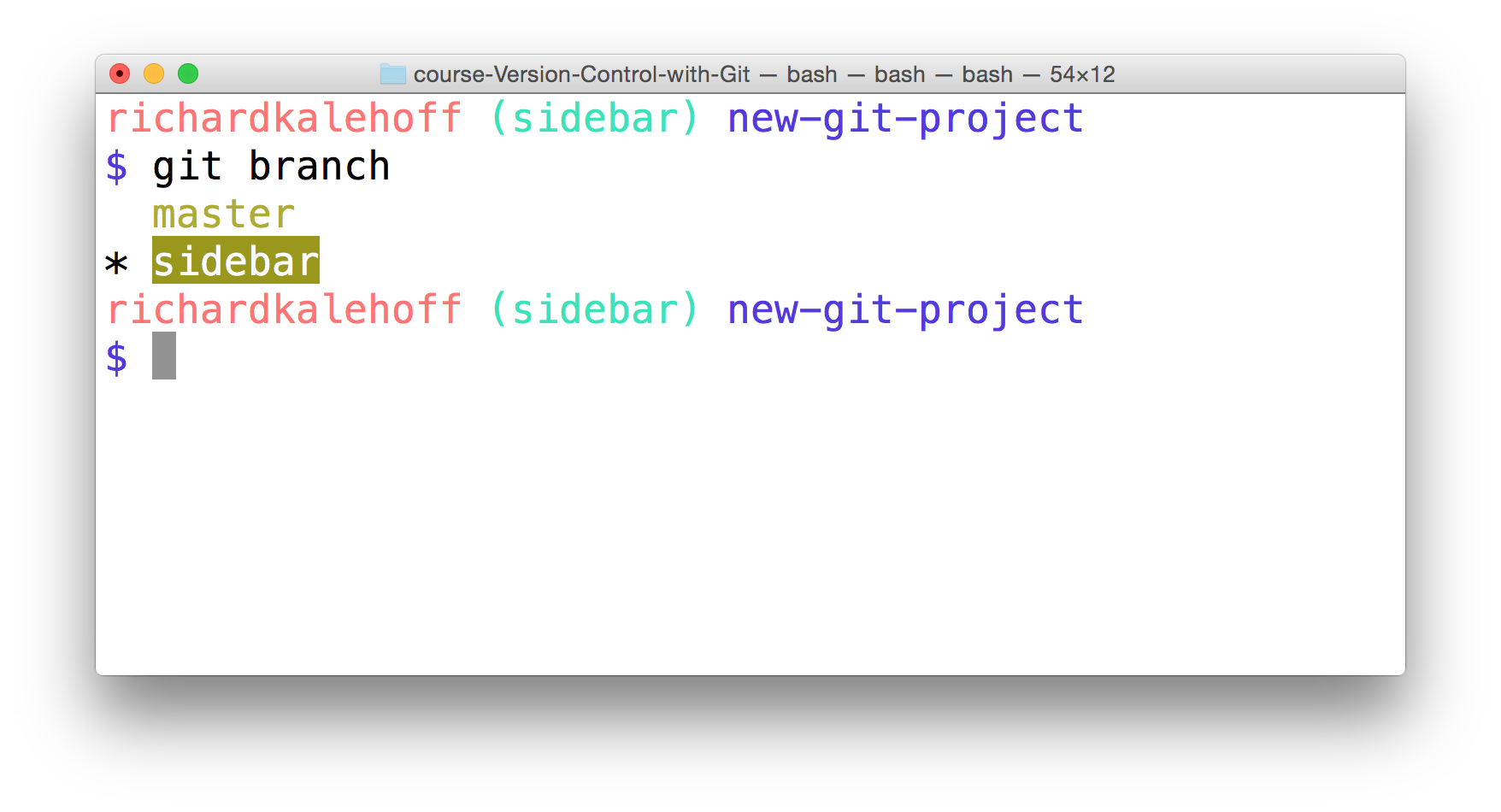
## The git checkout Command

Remember that when a commit is made that it will be added to the current branch. So even though we created the new sidebar, no new commits will be added to it since we haven't switched to it, yet. If we made a commit right now, that commit would be added to the master branch, not the sidebar branch. We've already seen this in the demo, but to switch between branches, we need to use Git's checkout command.

$ git checkout sidebar

## The Active Branch

The command prompt will display the active branch. But this is a special customization we made to our prompt. If you find yourself on a different computer, the fastest way to determine the active branch is to look at the output of the git branch command. An asterisk will appear next to the name of the active branch.

[[](https://classroom.udacity.com/courses/ud123/lessons/61776182-df3f-4767-9558-5d1591c1a709/concepts/89a3a3ca-c47d-4e5f-87c7-cbc838f62232)](https://classroom.udacity.com/courses/ud123/lessons/61776182-df3f-4767-9558-5d1591c1a709/concepts/89a3a3ca-c47d-4e5f-87c7-cbc838f62232)

## Delete A Branch

A branch is used to do development or make a fix to the project that won't affect the project (since the changes are made on a branch). Once you make the change on the branch, you can combine that branch into the master branch (this "combining of branches" is called "merging" and we'll look at it shortly).

Now after a branch's changes have been merged, you probably won't need the branch anymore. If you want to delete the branch, you'd use the -d flag. The command below includes the -d flag which tells Git to delete the provided branch (in this case, the "sidebar" branch).

$ git branch -d sidebar

One thing to note is that you can't delete a branch that you're currently on. So to delete the sidebar branch, you'd have to switch to either the master branch or create and switch to a new branch.

Deleting something can be quite nerve-wracking. Don't worry, though. Git won't let you delete a branch if it has commits on it that aren't on any other branch (meaning the commits are unique to the branch that's about to be deleted). If you created the sidebar branch, added commits to it, and then tried to delete it with the git branch -d sidebar, Git wouldn't let you delete the branch because you can't delete a branch that you're currently on. If you switched to the master branch and tried to delete the sidebar branch, Git *also* wouldn't let you do that because those new commits on the sidebar branch would be lost! To force deletion, you need to use a capital D flag - git branch -D sidebar.

**GIT Merge**

Combines changes from different branches