Version Control with Git

Sunday, December 22, 2024 8:29 AM

Resources

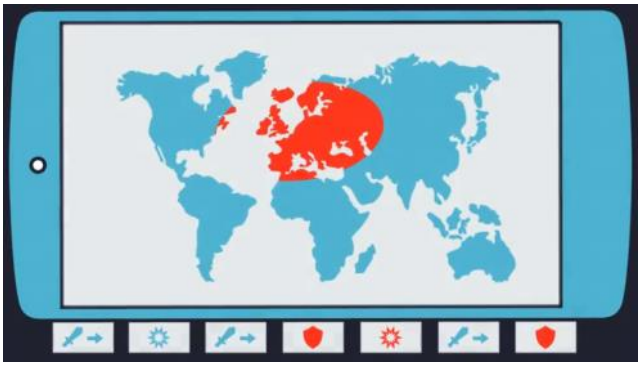
Version Control with Git | Udacity
Git and GitHub | شخبط وانت متطمن

Table of Contents

What is Version Control

Hello and welcome. Today we'll be learning all about version control, and we'll be using the version control tool Git to help us manage different versions of our project. Now, before we dive into some of the nitty gritty details, let me tell you a story, because who doesn't love story time? Back in the day, my brother and his best friend would play games all the time. But, one of their favorite board games was a popular World War II strategy board game.

The board game shows a map of the world, and you put pieces on the board and move them around. I didn't really care for the game all that much because it would take so long to play. Sometimes, they'd be playing a single game for a day or more. Now, because there's a big board with lots of pieces that move around, if they had to pause the game, they would snap a picture with their phone.



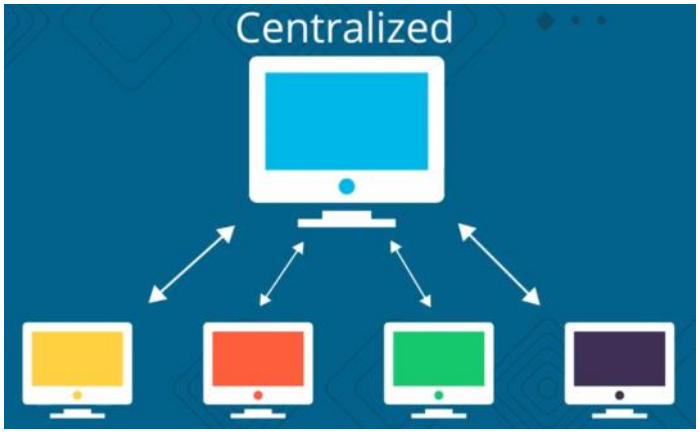
They'd use these pictures as save points in case anything happened to the board or the pieces, like someone bumping the table or the dog coming along and knocking the pieces everywhere. If there were a problem, they could use a picture to recreate the exact setup and jump back to that point in the game. So each of these pictures is a save point. The pictures record the time, who took the picture, and the state and location of the pieces.

This idea of a save point is exactly what version control does. You don't seem all that impressed. Well, let me tell you that it's incredibly powerful and I love having it because when you use version control, you create safe points that save your project. This lets you have the freedom to change absolutely anything about your project and be confident that nothing will be lost. Total security and total freedom, who wouldn't want that? And you get all of that with a version control system. Sounds pretty awesome, right? So hi, I'm Richard, and I've been using version control for years, and Git has saved me from countless sticky situations. For example, one time I'd realized I deleted an important image, but I was able to recover it because the project was being tracked with version control. It's an incredibly helpful tool and you'll grow to depend on it as much as I do. You ready to get started? Awesome, let's get going.

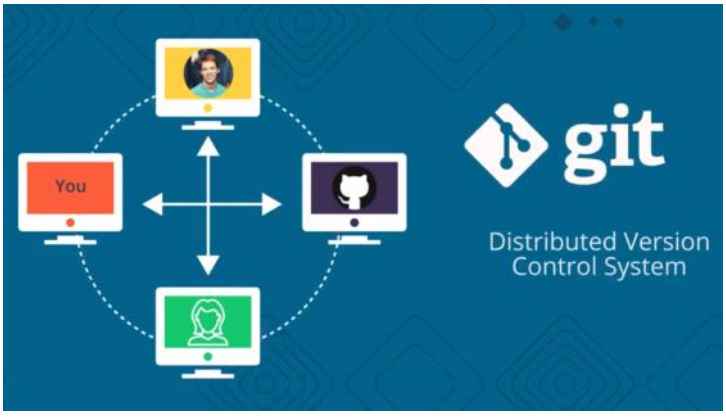
I just said "version control". Now since you're in this course, you might already know a bit about version control. But if you don't, a handy trick I've learned is to try reading the words in reverse - so "version control" would become "control version"! So a Version Control System is just software that helps you control (or manage) the different versions...of something (typically source code).

There are a number of version control systems out there, but three of the most popular are **Git**, **Subversion** and **Mercurial**.

Not only are there dozens and dozens of different version control systems, there are actually two different categories. There's the **centralized** model and then the **distributed** model.
In a centralized model there's one all-powerful central computer that hosts the project; every interaction must go through this central computer.



In the distributed model there's no central repository of information. Each developer has a complete copy of the project on their computer, which is cool because that means you can also work off line. In this course we'll be using the version control system Git, which is a distributed version control system.



Now, you might have heard of Git hub. Git and GitHub are quite different. **Git, is the version control tool, while GitHub is the service that hosts Git projects.** To work with Git projects you're not required to use GitHub, but it's an awesome tool and you should definitely check it out. For now, you can think of it as basically, just another computer where you can upload a copy of your project.

Centralized, distributed, if these don't make a lot of sense right now, that's totally fine. This is to give you a big picture idea of where things fit together. These terms are more important if you're working with a team but you'll be working locally on your computer for the entirety of this course.

Recap

Remember that the main point of a version control system is to help you maintain a detailed history of the project as well as the ability to work on different versions of it. Having a detailed history of a project is important because it lets you see the progress of the project over time. If needed, you can also jump back to any point in the project to recover data or files.
In this course, we'll be using Git which is a distributed version control system. You might be surprised to discover that you're already using version control all the time!

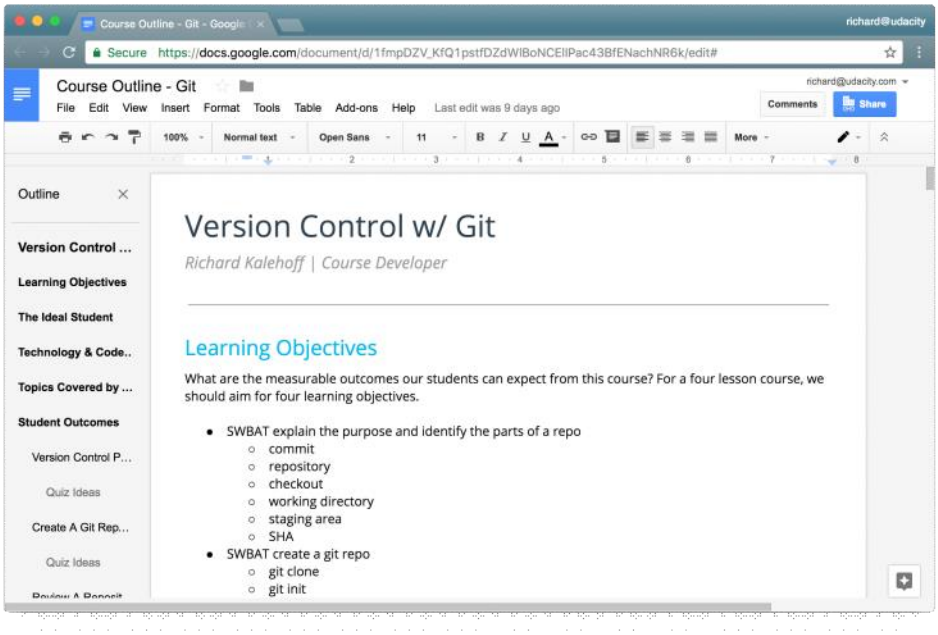
Version Control Is Everywhere

My job revolves around working with documents. I create new documents all the time, I fill them with information (hopefully informative information!) and then edit...edit...edit! Is your job like this? Perhaps it's not documents of text, but you're probably working with data in some form or another that changes over time.

Now you might not think that you're using version control when working with documents, and you'd be right...sort of. You're not *actively* maintaining different versions of a document as you write it. But that doesn't mean there aren't different versions of the document. The computer is keeping track of the different versions for you!
Don't believe me? Aside from pondering your propensity towards doubting, let's prove I'm right:

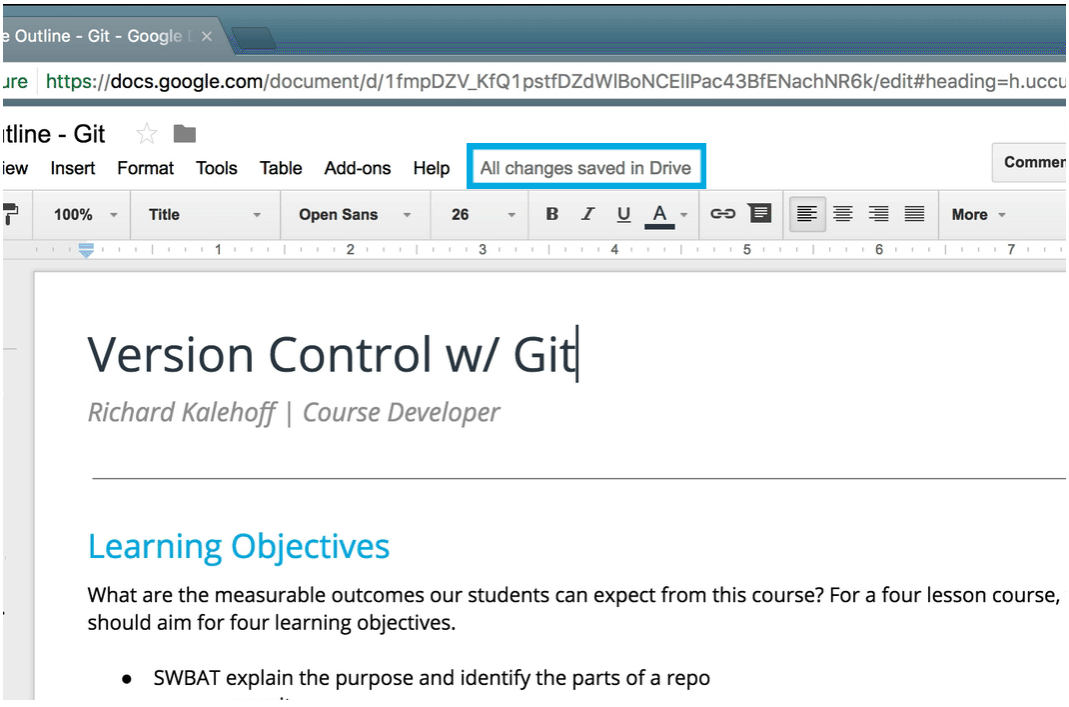
- open up your favorite text editor/code editor
- type some content (how about "version control is dull!")
- change one of the words in you wrote (e.g. change "dull" to "life-changing awesome")
- now (here it comes...!) press cmd + z or ctrl + z

🕸 Version control in action! (See?...told you I wasn't lying) I bet you use the "undo" command all the time. I know that I sure do!
Practically every application I've ever used has an undo feature. You can think of this as a form of version control, but it's a rather limited form of version control. Let's look at a more powerful form by checking out a Google Docs document.



The Google Doc outline for this Git Course.

If you've ever written in a Google doc, have you noticed the small gray text at the top that tells you about the status of the document? Ever noticed that as you type, it's actively saving the document? Then, when you finish typing, it tells you that the document has saved.



Status of a Google Docs document. The status says "Saving..." while the document is being edited, and changes to "All changes saved in Drive" after the content is saved.

The real question is, did you know that is a link that you can click on? Wanna see for yourself? Try it out in one of your own Google Docs.
Clicking on the link takes you to a "Revision history" page. (Ooo! Did you notice the word "revision"? The word "version" is a synonym for "revision"!)

Revision History Isn't Powerful Enough

Google Docs' Revision history page is incredibly powerful! I've used it on several occasions to salvage text that I'd written at one point, erased, and then realized I actually did want to keep.
But for all its ability, it's not as powerful as we'd like. What's it missing? A few that I can think of are:

- the ability to label a change
- the ability to give a detailed explanation of why a change was made
- the ability to move between different versions of the same document
- the ability to undo change A, make edit B, then get back change A without affecting edit B

- The version control tool, Git, can do all of those things - *and more*!!! (bet you didn't see *that* coming!) So have I sold you yet on the awesomeness that is Git? I hope so, cause we're about to dive into it in the next section.

Git and Version Control Terminology

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 is an SCM (and therefore a VCS!). The URL for the Git website is https://git-scm.com/**(opens in a new tab)** (see how it has "SCM" directly in its domain!).

Commit

Git thinks of its data like a set of snapshots of a mini filesystem. Every time you **commit** (save the state of your project in Git), it basically takes a picture of what all your files look like at that moment and stores a reference to that snapshot. You can think of it as a save point in a game - it saves your project's files and any information about them.
Everything you do in Git is to help you make commits, so a commit is *the* fundamental unit in Git.

Repository / repo

A **repository** is a directory which contains your project work, as well as a few files

(hidden by default on Mac OS X) which are used to communicate with Git. Repositories can exist either locally on your computer or as a remote copy on another computer. A repository is made up of commits.

Working Directory

The **Working Directory** is the files that you see in your computer's file system. When you open your project files up on a code editor, you're working with files in the Working Directory.
This is in contrast to the files that have been saved (in commits!) in the repository. When working with Git, the Working Directory is also different from the command line's concept of the *current working directory* which is the directory that your shell is "looking at" right now.

Checkout

A **checkout** is when content in the repository has been copied to the Working Directory.

Staging Area / Staging Index / Index

A file in the Git directory that stores information about what will go into your next commit. You can think of the **staging area** as a prep table where Git will take the next commit. Files on the Staging Index are poised to be added to the repository.

SHA

A **SHA** is basically an ID number for each commit. Here's what a commit's SHA might look like: `e2adf8ae3e2e4ed40add75cc44cf9d0a869afeb6`.
It is a 40-character string composed of characters (0–9 and a–f) and calculated based on the contents of a file or directory structure in Git. "SHA" is shorthand for "Secure Hash Algorithm". If you're interested in learning about hashes, check out our **Intro to Computer Science course(opens in a new tab)**.

Branch

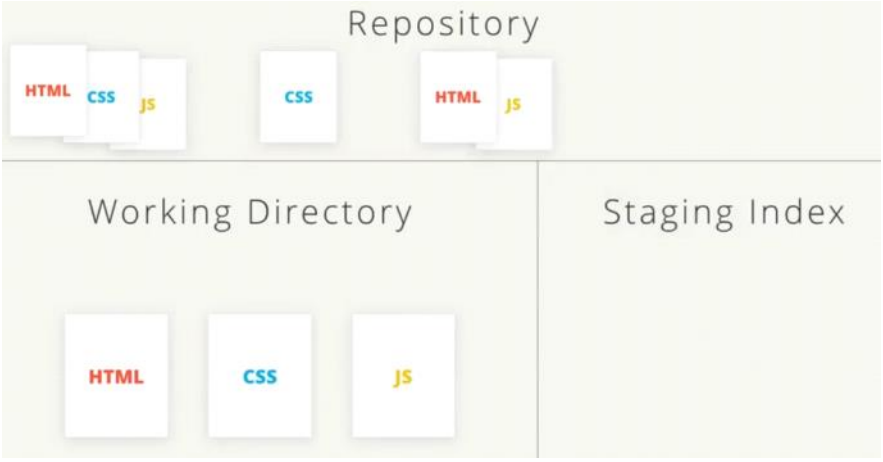
A **branch** is when a new line of development is created that diverges from the main line of development. This alternative line of development can continue without altering the main line.
Going back to the example of save point in a game, you can think of a branch as where you make a save point in your game and then decide to try out a risky move in the game. If the risky move doesn't pan out, then you can just go back to the save point. The key thing that makes branches incredibly powerful is that you can make save points on one branch, and then switch to a different branch and make save points there, too.
With this terminology in mind, let's take a high-level look at how we'll be using Git by looking at the typical workflow when working with version control.

Git The Big Picture

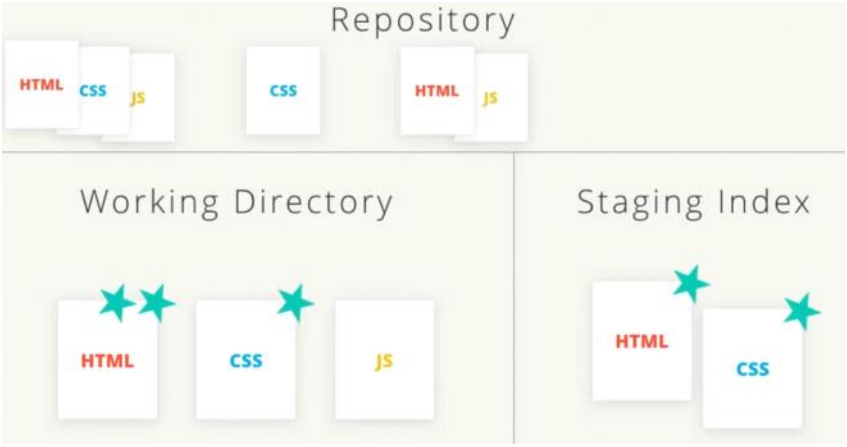
Let's say we have a repository that is made up of these three files. We want to start using version control in this project. So we create a new git repository. In a git repository, there are three main areas we need to know about. First, there's the working directory, then there's the staging index, and last is the repository. So we have these three distinct areas in git.

When you're first working with git, it can be a bit hard to realize that there are these three distinct areas, because you don't really see any visible change to the files on your computer. Git manages these three different areas though, and we'll be using the git shell command to interact with files and move them from section to section. Okay, so we've created this empty git repository, we can tell it's an empty repository because the repository section is empty. Git knows about files that it hasn't seen before. Let's indicate that a file is new with a green star.

Git keeps track of files by adding them to the repository. We can't move these files right into the repository though, they have to be moved to the staging index first. So will use git to move the changes to the staging index. Remember that the staging index is where you can place all of the files that are about to be committed. Now let's commit them, which moves them into the repository. Now the files in their current state are safe. Let's say we then make some changes to our sites CSS file. Git sees this change and marks the file as modified. We take the same steps to move the changes in the CSS file to the staging index, and then commit them. I think you get the idea, but let's look at it one more time.



Let's say the HTML and JavaScript files have changed. Remember that git knows what files are new or changed. We'll move these changes to the staging index, and then commit them. So the process is, add a new file or modify an existing file, move the changes to the staging index, commit the changes to the repository. You got it? Okay then, what about this scenario, let's say the HTML and CSS files have changed and we staged the changes. If we then decide to modify the HTML file again, what would happen if we made a commit right now? Seems like a good time for a quiz.



Quiz Question

The HTML file has HTML and CSS changes on the Staging Index and an additional HTML change in the Working Directory. From what you've learned so far with how committing works, what do you think will get committed if a commit were made right now?

- ○ only the HTML file changes on the Staging Index
- ○ only the CSS file changes on the Staging Index
- ◉ the HTML and CSS file changes on the Staging Index ✓
- ○ the HTML file changes in *both* the Working Directory *and* the Staging Index

Awesome Job! When a commit is made, only the changes that are in the Staging Index are saved in the repository.

When a commit is made, only the changes that are on the staging index are moved into the repository. If we want to include the second set of changes in the commit, then we need to stage the changes.

Now that the second change to the HTML file has been staged, the two sets of changes to the HTML file are combined into just one set of changes. And then we can commit like normal to move these changes into the repository. We have all of these changes saved in our repository, but how do we access these commits? What if we want to look at this specific commit? When each commit is made, git creates an ID for it. The ID for the commit is its SHA, the first seven characters what each commits SHA might look like.

Recap

At its core, git records changes to files as commits. These commits are all saved in a repository. Everything else, things like the working directory branches and SHAs are there to help make and reference commits. As you've seen the terminology that version control uses is kind of cryptic at first. Hopefully you had the hand out with you and reviewed it as you went through the high level overview of how git it is used. In the next section we're going to install and configure git. Make sure to follow each of the steps in order so that your computer will be configured like mine.

Now, configuring your terminal isn't required for git to work correctly or to continue with this course. You can do the entire course without this step, but not only does it make it easier to use, all professionals reconfigure their terminal. The reconfiguration we'll be doing will add color to the terminal and display specific git information. Sounds good. Let's get to it.

Create a Git Repo

The init subcommand is short for "initialize", which is helpful because it's the command that will do all of the initial setup of a repository. We'll look at what it does in just a second.

Git Init

Fantastic work - we're all set up and ready to start using the `git init` command!

This is one of the easiest commands to run. All you have to do is run `git init` on the terminal. That's it! Go ahead, why not give it a try right now!

Quiz Question

Did you run `git init` yet? If not, do it now because you'll need it to answer this quiz!

After running `git init`, the text "Initialized empty Git repository in " followed by a path. should have appeared. The question is, has anything changed with your command prompt? If so, what?

- (●) Yes - I now see the word "master". ✓
- ( ) No, it looks exactly the same.

Git Init's Effect

Running the `git init` command sets up all of the necessary files and directories that Git will use to keep track of everything. All of these files are stored in a directory called `.git` (notice the `.` at the beginning - that means it'll be a hidden directory on Mac/Linux). This `.git` directory *is the "repo"*! This is where git records all of the commits and keeps track of everything!

Let's take a brief look at the contents of the `.git` directory.

> WARNING: Don't directly edit any files inside the `.git` directory. This is the heart of the repository. If you change file names and/or file content, git will probably lose track of the files that you're keeping in the repo, and you could lose a lot of work! It's okay to look at those files though, but don't edit or delete them.

.Git Directory Contents

We're about to take a look at the `.git` directory...it's not vital for this course, though, so don't worry about memorizing anything, it's here if you want to dig a little deeper into how Git works under the hood.

Here's a brief synopsis on each of the items in the `.git` directory:

- **config file** - where all *project specific* configuration settings are stored.
  From the **Git Book**:

> Git looks for configuration values in the configuration file in the Git directory (.git/config) of whatever repository you're currently using. These values are specific to that single repository.

For example, let's say you set that the global configuration for Git uses your personal email address. If you want your work email to be used for a specific project rather than your personal email, that change would be added to this file.

- **description file** - this file is only used by the GitWeb program, so we can ignore it
- **hooks directory** - this is where we could place client-side or server-side scripts that we can use to hook into Git's different lifecycle events
- **info directory** - contains the global excludes file
- **objects directory** - this directory will store all of the commits we make
- **refs directory** - this directory holds pointers to commits (basically the "branches" and "tags")

Remember, other than the "hooks" directory, you shouldn't mess with pretty much any of the content in here. The "hooks" directory *can* be used to hook into different parts or events of Git's workflow, but that's a more advanced topic that we won't be getting into in this course.

Clone An Existing Repo

Why Clone?

First, what is cloning?

> to make an identical copy

What's the value of creating an identical copy of something, and how does this relate to Git and version control?

Why would you want to create an identical copy? Well, when I work on a new web project, I do the same set of steps:

- create an `index.html` file
- create a `js` directory
- create a `css` directory
- create an `img` directory
- create `app.css` in the css directory
- create `app.js` in the js directory
- add starter HTML code in `index.html`
- add configuration files for linting (validating code syntax)
  - **HTML linting**
  - **CSS linting**
  - **JavaScript linting**
- **configure my code editor**

...and I do this *every time* I create a new project!...which is a lot of effort I'm putting in for each new project. I didn't want to keep doing these same steps over and over, so I did all of the steps listed above one last time and created a starter project for myself. Now when I create a new project, I just make an identical copy of that starter project!

The way that cloning relates to Git is that the command we'll be running on the terminal is `git clone`. You pass a path (usually a URL) of the Git repository you want to clone to the `git clone` command.

Wanna try cloning an existing project? Let's see how Git's `clone` command works!

Verify Terminal Location

TIP: Now before you clone anything, make sure you are located in the correct directory on the command line. Cloning a project creates a new directory and places the cloned Git repository in it. The problem is that you can't have nested Git repositories. So make sure the terminal's current working directory isn't located in a Git repository. If your current working directory is not in your shell's prompt, type `pwd` to print the working directory.

```
moham@DESKTOP-OAIO3S2 MINGW64 ~/Workplace/udacity-git-course
$ git clone https://github.com/udacity/course-git-blog-project
Cloning into 'course-git-blog-project'...
remote: Enumerating objects: 166, done.
remote: Counting objects: 100% (75/75), done.
remote: Compressing objects: 100% (14/14), done.
remote: Total 166 (delta 69), reused 61 (delta 61), pack-reused 91 (from 1)Receiving
Receiving objects:  50% (83/166), 1.25 MiB | 1.23 MiB/s
Receiving objects: 100% (166/166), 2.05 MiB | 1.38 MiB/s, done.
Resolving deltas: 100% (72/72), done.
```

Git Clone Output Explanation

Let's look briefly at the output that `git clone` displays.

The first line says "Cloning into 'course-git-blog-project'...". Git is creating a directory (with the same name of the project we're cloning) and putting the repository in it...that's pretty cool!

The rest of the output is basically validation - it's counting the remote repository's number of objects, then it compresses and receives them, then it unpacks them.

What if you want to use a different name instead of the default one? Yes, you could just run the command above and manually rename it in Finder/Windows Explorer or use `mv` on the terminal. But that's too many steps for us! Instead, we'd rather clone the project and have it use a different name all in one go! But how do we do that?

Quiz Question

Why don't you check out the documentation for `git clone` and pick the correct way to do it from the options below. The command should clone the blog project repo and store it in a directory named `blog-project`.

- ○ git clone-new-dir https://github.com/udacity/course-git-blog-project blog-project
- ○ git clone https://github.com/udacity/course-git-blog-project --out blog-project
- ○ git clone https://github.com/udacity/course-git-blog-project --rename blog-project
- ◉ git clone https://github.com/udacity/course-git-blog-project blog-project ✓

Not In A Git Repository?

WARNING: Here's a *very important* step that often gets missed when you first start working with Git. When the `git clone` command is used to clone a repository, it creates a new directory for the repository...you already know this. *But, it doesn't change your shell's working directory.* It created the new repo inside the current working directory, which means that the current working directory is still *outside* of this new Git repo! Make sure you `cd` into the new repository.
Remember to use the Terminal's command prompt as an aid - if you're in a directory that is a Git repository, the command prompt will include a name in parentheses.

Determine A Repo's Status

Working with Git on the command line can be a little bit challenging because it's a little bit like a **black box**. I mean, how do you know when you should or shouldn't run certain Git commands? Is Git ready for me to run a command yet? What if I run a command but I think it didn't work...how can I find that out? The answer to all of these questions is the `git status` command!

```
$ git status
```

The `git status` is our key to the mind of Git. It will tell us what Git is thinking and the state of our repository as Git sees it. When you're first starting out, you should be using the `git status` command *all of the time*! Seriously. You should get into the habit of running it after any other command. This will help you learn how Git works and it'll help you from making (possibly) incorrect assumptions about the state of your files/repository.

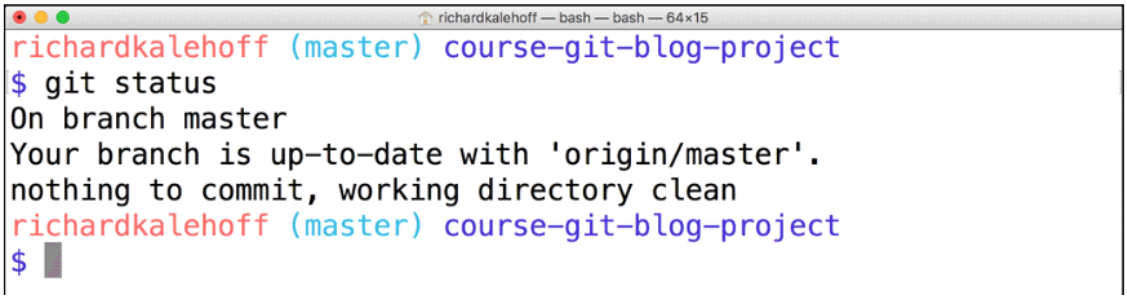
Question 1 of 2

To answer this quiz, make sure you've `cd` ed into the `course-git-blog-project` project. If you've been following along in this lesson and haven't added any files to this project, then what does running `git status` display?

- ○ Status: good
- ○ On branch master
Your branch is up-to-date with 'origin/master'.
Initial commit
nothing to commit (create/copy files and use "git add" to track)
- ○ master branch
Please commit some files
- ◉ On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean ✓

Git Status Output

The `git status` command will display a lot of information depending on the state of your files, the working directory, and the repository. You don't need to worry too much about these, though...just run `git status` and it will display the information you need to know.



Git Status Explanation

As you can see in the GIF above, running `git status` in the `course-git-blog-project` project produces the following output:

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

The output tells us two things:

1. `On branch master` – this tells us that Git is on the `master` branch. You've got a description of a branch on your terms sheet so this is the "master" branch (which is the default branch). We'll be looking more at branches in lesson 5

- `Your branch is up-to-date with 'origin/master'.` – Because `git clone` was used to copy this repository from another computer, this is telling us if our project is in sync with the one we copied from. We won't be dealing with the project on the other computer, so this line can be ignored.
- `nothing to commit, working directory clean` – this is saying that there are no pending changes.

Think of this output as the "resting state" (that's not an official description - it's how I like to describe it!). This is the resting state because there are no new files, no changes have been made in files, nothing is in the staging area about to be committed...no change or action is pending, so that's why I like to call it the resting state.

So this is what it looks like when running `git status` in a repository that already has commits. Let's switch to the `new-git-project` project to see what the `git status` output will produce.

💡 Changes in Git v2.14 💡

In Git version 2.14, running the `git status` command in an empty directory changed the wording of "Inital commit" to the much clearer "No commits yet". So the output would be:

```
On branch master

No commits yet

nothing to commit (create/copy files and use "git add" to track)
```

Question 2 of 2

To answer this quiz, make sure you've `cd`ed into the `new-git-project` project.

If you've been following along in this lesson and haven't added any files to this project, then what does running `git status` display?

- ○ Status: good
- ◉ On branch master
Initial commit
nothing to commit (create/copy files and use "git add" to track) ✓
- ○ master branch
Please commit some files
- ○ On branch master
nothing to commit, working directory clean

Explanation Of Git Status In A New Repo

This is the output of running `git status` in the `new-git-project` project:

```
$ git status
On branch master

Initial commit

nothing to commit (create/copy files and use "git add" to track)
```

To be completely clear, I haven't made any commits in my project yet. If you have made a commit, then your output should look exactly like that of the course-git-blog-project project.

If you compare this to the `git status` output from the course-git-blog-project project, then you'll see that they're pretty similar. The thing to note that's different is that this output includes the line `Initial commit`. This is the tiniest bit confusing because there actually aren't any commits in this repository yet! We haven't discussed making a commit yet, but when we do, we will be able to make an initial commit.

Wanna have a sneak peak of the next lesson and at the same time prove that there aren't any commits in this repo yet? Great, I knew you did! Try running the command `git log` and check out its response:

```
$ git log
fatal: your current branch 'master' does not have any commits yet
```

Well, that's kind of scary looking. "Fatal"? Fortunately, it turns out that just means that the Git program is exiting because it can't find any work to do. Git tells us this as if it were an error, but it's really not a problem. We know we haven't put any commits into this repo yet.

It's pretty clear from the response that there aren't any commits!

We've just taken a very brief look at the `git status` command. Remember that the output of `git status` will change depending on if files have been added/deleted/modified, what's on the staging index, and the state of the repository. We'll be using the `git status` command throughout this entire course, so get comfortable running it!

Git Status Recap

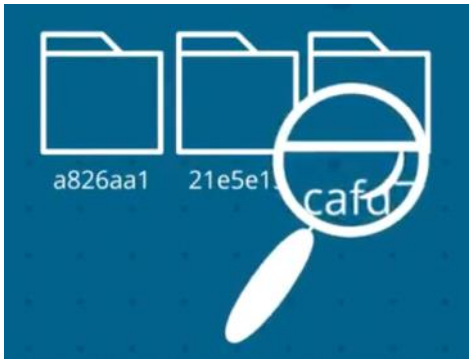
The `git status` command will display the current status of the repository.

```
$ git status
```

I can't stress enough how important it is to use this command *all the time* as you're first learning Git. This command will:

- tell us about new files that have been created in the Working Directory that Git hasn't started tracking, yet
- files that Git *is* tracking that have been modified
- a whole bunch of other things that we'll be learning about throughout the rest of the course ;-)

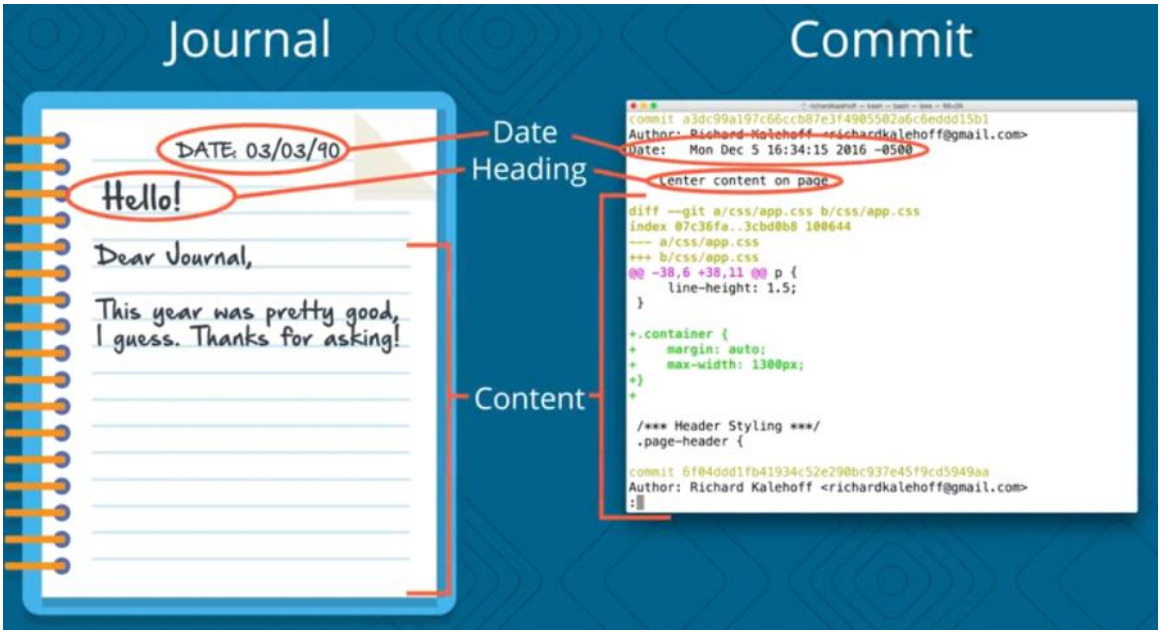
In this lesson, we use git init to create our own repository. We used git clone to copy an existing repository, and we used git status to determine the status of a repository. Now, you might want to just jump right into making commits because a repository isn't very useful without them. But before we start, being able to review commits will help immensely when we start making our own. So, let's learn how to review existing commits.



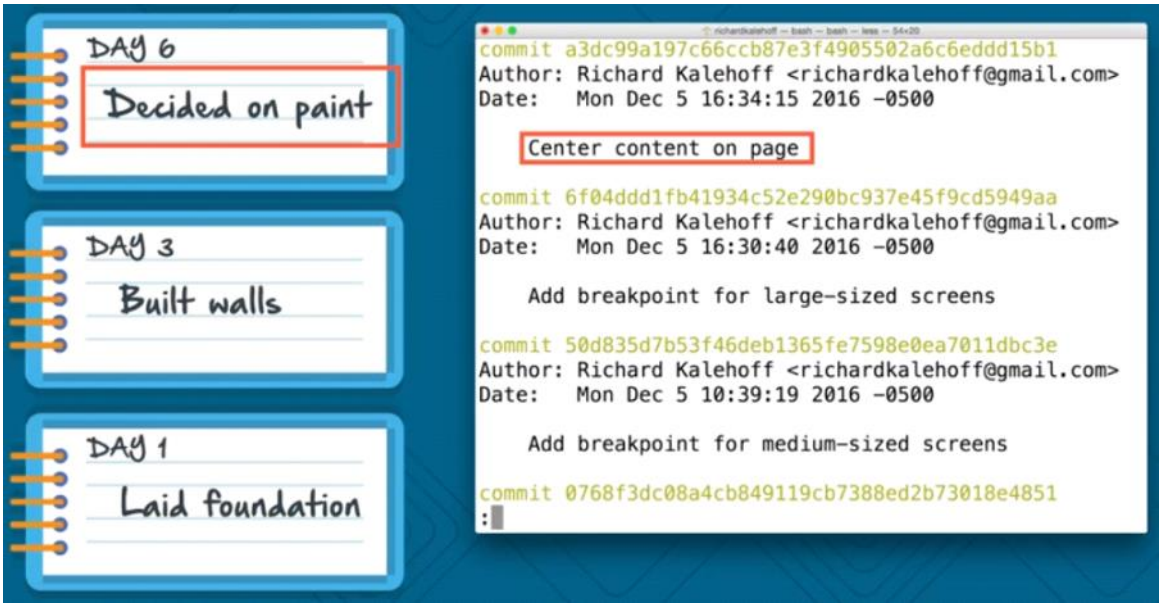
Review a Repo's History

Let me ask you a question. What did you do yesterday? Did you go to the beach? Take a walk around the block? Cook yourself some sweet and sour chicken because it's amazing? What about the day before yesterday? How about two weeks ago? When I was young, my parents wanted me to keep a daily journal so that I could look back on it sometime in the future and remember places I'd gone and experiences I'd encountered. Unfortunately, I wrote in it about once a year, which is terrible because the entries weren't detailed at all and just touched on high-level events.

A good journal entry should have a date so you know when it occurred, a heading so you can give it a short description of that entry, and then the entry itself that has all of the information. Each time a new entry is made in the journal, it's like making a commit in the repository.



And just like your journal entries, you want your commits to be frequent and descriptive so you can see the progress of your project over time. Git automatically records the date and the content changes that have been made when a commit is created, so all you need to do is provide the descriptive heading or message for the commit. You want journal commits to be frequent so that you can see the ups and downs in life over time. Similarly, you want to make frequent commits with descriptive messages.



This will make it easy to look back at the project and see how it has evolved. We're

not making commits just yet, but start getting used to the idea that you should make commits often with descriptive commit messages. But how often is often, and how can you know what makes a good commit? How many files or lines of code should be included in the commit? One of the best ways to know what makes a good commit is to look at existing commits.

The tools we'll be using to review a repository's history are **git log** and **git show**.

- With git log, you'll be able to display information about the existing commits. Git log is extremely powerful and you'll be using it all of the time.
- The git show command displays info about the given commit. So, with this one, you provide it the commit ID, also known as the SHA, and the command displays info about just that one commit.

Just two commands, git log and git show. Seem small, right? But this might actually be the most important lesson in the course. Learning how to display a repository's history and reviewing specific commits is useful for so many things. Are you ready? You're focused? Let's look at the all-powerful git log.

Displaying A Repository's Commits

TIP: In lesson 2 you used `git clone` to clone the blog project. This is the project we'll be using in this lesson. If you skipped cloning the project in the previous lesson, then run the following command to get the project:

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

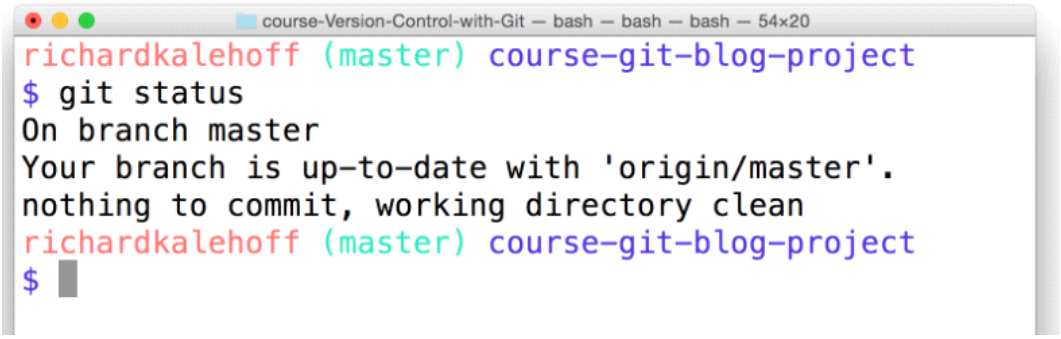
Don't forget to `cd` into the project after you've cloned it.

If you have questions about this, review how to **Clone An Existing Repo** or ask in **Knowledge**.

Question 1 of 5

After you've cloned the blog project repository, navigate to the project's directory using the command line. Once you're located inside the blog project, what is the very first thing you should do in a Git repository?

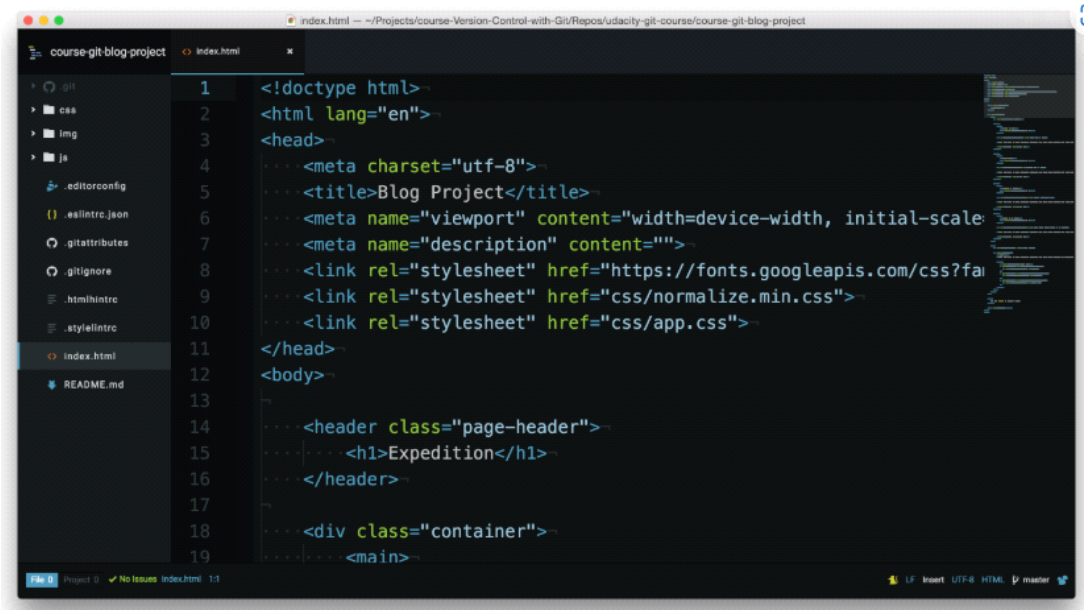
- (●) run the `git status` command ✓
- ( ) open the project in a code editor
- ( ) decide what new feature to work on



Git Status & Opening The Project

You can see that `git status` tells us that there's "nothing to commit, working directory clean". That means we're good to go ahead and check out the project!

So open the project in your favorite code editor. If you haven't yet, take a minute or two to look at the project – look over the CSS and the JavaScript files, but look particularly at the HTML file.



Question 2 of 5

In the `index.html` file, take a look at the `<h1>Expedition</h1>` heading around line 15.

Based on what you can see here when was that heading added?

- ◯ It was added on a Tuesday. Yeah, a Tuesday.
- ◯ 3 weeks ago
- ◉ ¯\(ツ)/¯ I can't tell that by looking at the code. ✓

Question 3 of 5

Ok, so we're not quite sure *when* the heading was added. How about an easier question - *who* added this heading? Again, what can you tell from just looking at the code?

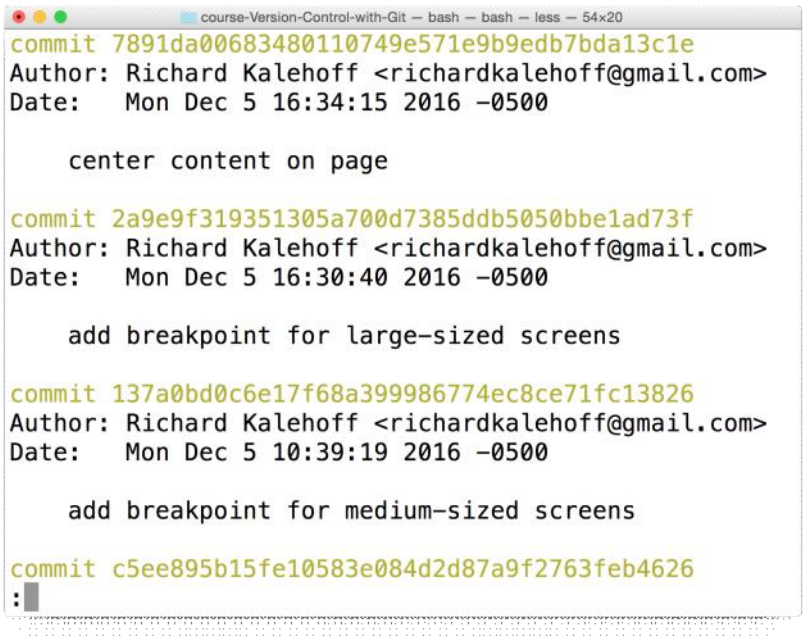
- ◯ Richard did!
- ◉ No clue ✓

The Git Log Command

Finding the answers to these questions is exactly what `git log` can do for us! Instead of explaining everything that it can do for us, let's experience it! Go ahead and run the `git log` command in the terminal:

```
$ git log
```

The terminal should display the following screen.



We're looking at the log of commits that have been made in this repository. This is one commit, here is another. Look at all of the information that the log is displaying.
It's showing the commits SHA, the person who made the commit, the date the commit was made, and the message for the commit. This is just a short description of what changes were made in this commit. Now these are just the first three or so commits that can fit on the screen. There are plenty more. How do I know that? The colon here means that there are more lines of output to be displayed.

Navigating The Log

If you're not used to a pager on the command line, navigating in **Less** can be a bit odd. Here are some helpful keys:

- to scroll **down**, press
  - `j` or `↓` to move *down* one line at a time
  - `d` to move by half the page screen
  - `f` to move by a whole page screen
- to scroll **up**, press
  - `k` or `↑` to move *up* one line at a time
  - `u` to move by half the page screen
  - `b` to move by a whole page screen
- press `q` to **quit** out of the log (returns to the regular command prompt)

Question 4 of 5

Use `git log` to find the commit that has a SHA that starts with `f9720a`. Who made the commit?

- ○ James Parkes
- ◉ Richard Kalehoff ✓
- ○ Colt Steele
- ○ Julia Van Cleve
- ○ Godzilla McFiddlebrunches

Use / to search commit response

What Is The Message? ✓

Use `git log` to find the commit with the SHA that starts with `8aa6668`. What is the message for that commit?

Convert social links from text to images

Question 5 of 5

Use `git log` to find the commit with the SHA that starts with `f9720a9`. When was that commit made?

- ○ Mon Dec 5 10:25:22 2016
- ◉ Mon Dec 5 10:11:51 2016 ✓
- ○ Sat Dec 3 16:09:00 2016
- ○ Fri Dec 2 16:58:27 2016

Changing How Git Log Displays Information

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**.

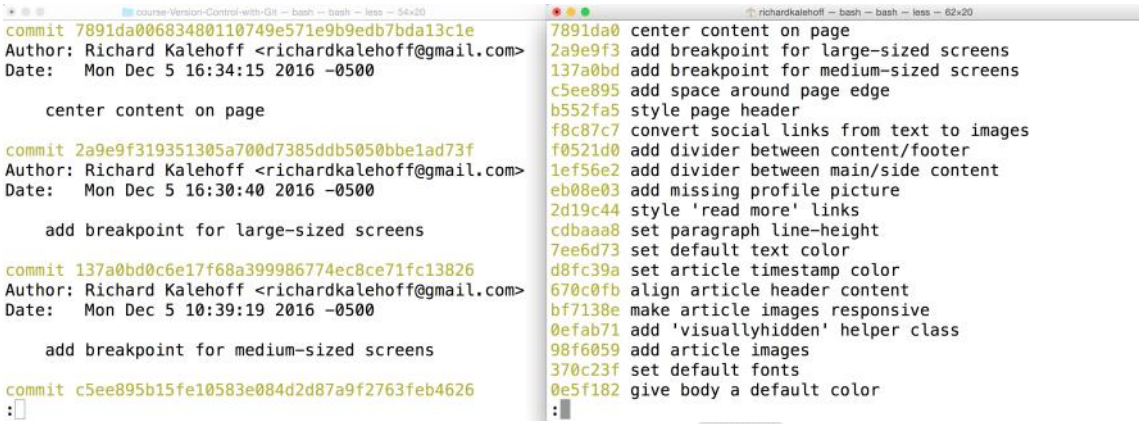
> TIP: This isn't a course on the command line, but a flag is used to alter how a program functions. For example, the `ls` command will list all of the files in the current directory. The `ls` command has a `-l` flag (i.e. `ls -l`) that runs the same `ls` command but alters how it works; it now displays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

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
```

Check out how different the output is!



Two Terminal applications side-by-side. The left one shows the result of the `git log` command with all of the information while the right one shows the result of the `git log --oneline` command with just the short SHA and the commit message.

Quiz Question

You're deep in the weeds of the `git log --oneline` command and want to get out of the `git log --oneline` output and return to the regular command prompt. What do you press on the keyboard to return to the regular command prompt?

- ○ the `esc` key
- ◉ the `q` key ✓
- ○ the `ctrl` + `c` keys
- ○ the `ctrl` + `d` keys

git log --oneline Recap

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

```
$ git log --oneline
```

This command:

- lists one commit per line
- shows the first 7 characters of the commit's SHA
- shows the commit's message

Viewing Modified Files

We just looked at the `--oneline` flag to show one commit per line. That's great for getting an overview of the repository. But what if we want to dig in a little to see what file or files were changed by a commit?

Question 1 of 4

If you look in the repository at commit `a3dc99a`, it has the message "Center content on page".

What file or files were changed in this commit?

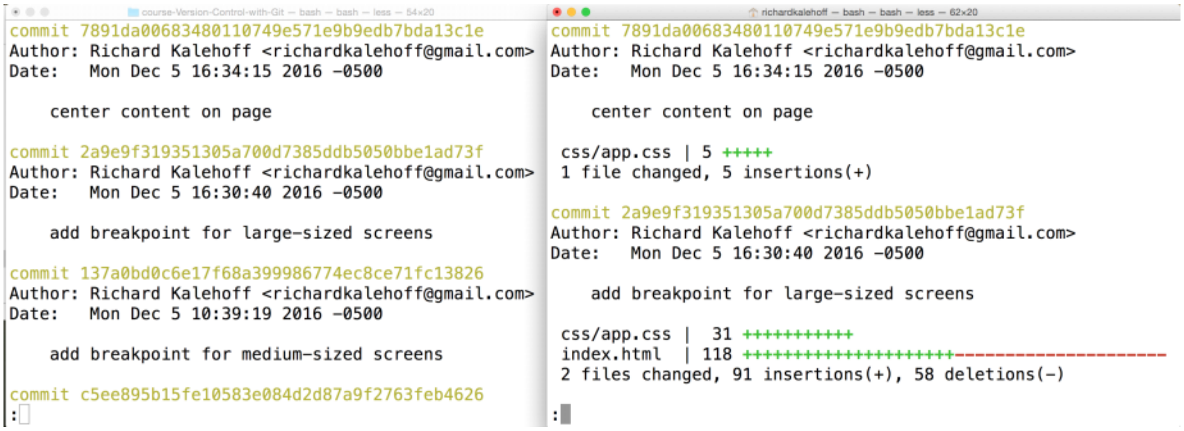
- ○ An HTML file
- ○ A CSS file
- ○ A JavaScript file
- ○ An HTML and CSS file
- ○ An HTML and JS file
- ◉ There's no way to know for sure ✓

`git log --stat` Intro

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

```
$ git log --stat
```

Run this command and check out what it displays.



Two Terminal applications side-by-side. The left one shows the result of the `git log` command with all of the information while the right one shows the result of the `git log --stat` command which lists the files that were changed as well as the number of added/removed lines.

Question 2 of 4

Using what you've learned so far about `git log` and its flags, how many files were modified in the commit with the SHA `6f04ddd`?

- ○ 1 file
- ◉ 2 files ✓
- ○ 9 files
- ○ 10 files

Question 3 of 4

You did so well with the first one, so here's another! How many files were modified in the commit with the SHA `8d3ea36`?

- ○ 1 file
- ◉ 2 files ✓
- ○ 3 files
- ○ 5 files

Question 4 of 4

Now it's time to look at the other info the `--stat` flag displays. How many lines of code were *deleted* in `index.html` in the commit with the SHA `8d3ea36`?

- ◯ 2 lines
- ◉ 4 lines ✓
- ◯ 9 lines
- ◯ 13 lines
- ◯ 28 lines

`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

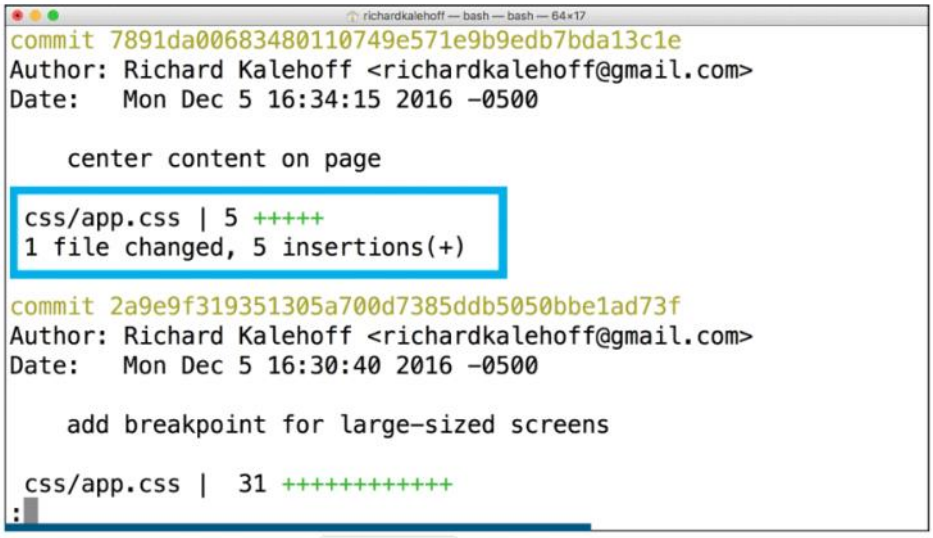
Viewing File Changes

Viewing Changes

We know that `git log` will show us the commits in a repository, and if we add the `--stat` flag, we can see what files were modified and how many lines of code were added or removed. Wouldn't it be awesome if we could see exactly *what those changes were*?

If this isn't the best part of a version control system, I don't know what is! Being able to see the exact changes that were made to a file is incredibly important! Being able to say, "oh, ok, so this commit adds 5 pixels of border-radius to the button!".

For example, in the blog project, the commit `a3dc99a` has the message "center content on page" and modifies the CSS file by adding 5 lines. What are those five lines that were added? How can we figure out what those 5 lines are?

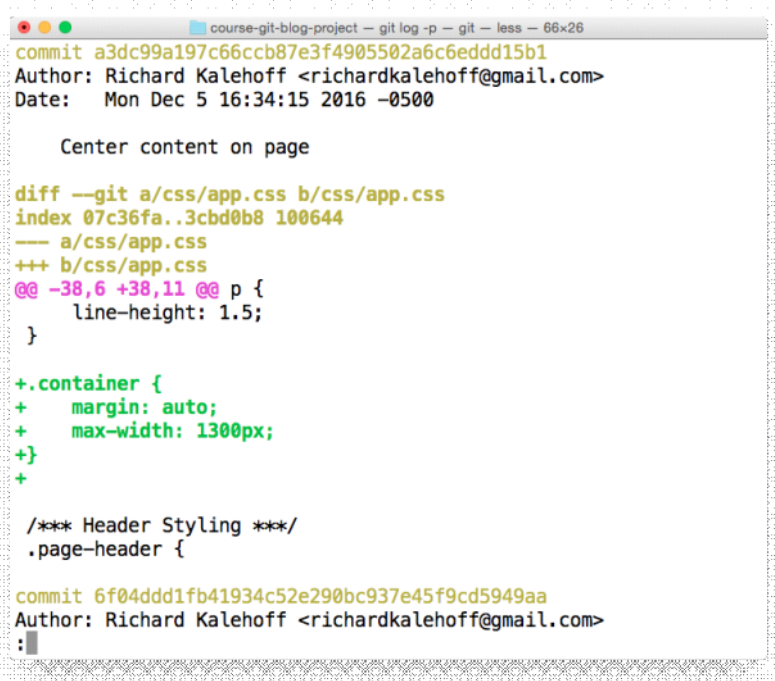


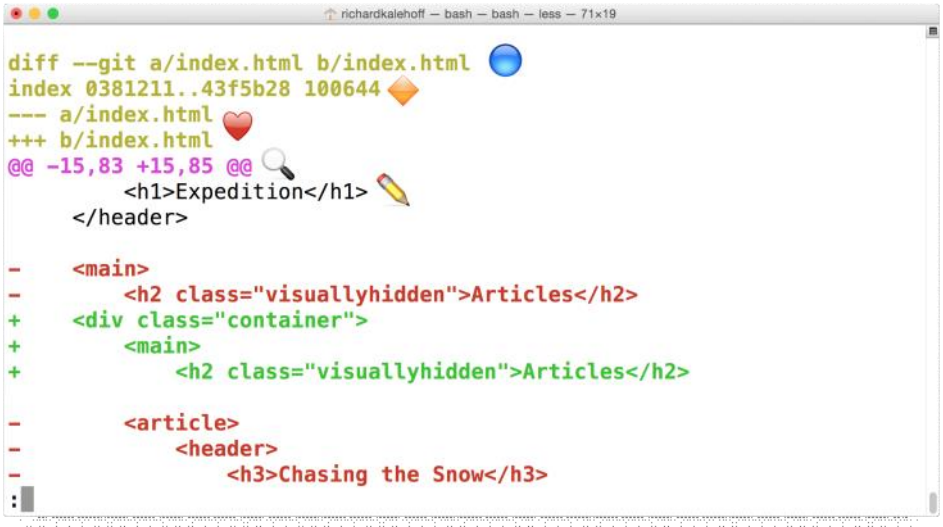
`git log -p`

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
```

Run this command and check out what it displays.


course-git-blog-project — git log -p — git — less — 66×26
commit a3dc99a197c66ccb87e3f4905502a6c6eddd15b1
Author: Richard Kalehoff <richardkalehoff@gmail.com>
Date: Mon Dec 5 16:34:15 2016 -0500
Center content on page
diff --git a/css/app.css b/css/app.css
index 07c36fa..3cbd0b8 100644
--- a/css/app.css
+++ b/css/app.css
@@ -38,6 +38,11 @@ p {
line-height: 1.5;
}
+.container {
+ margin: auto;
+ max-width: 1300px;
+}
+
/*** Header Styling ***/
.page-header {
commit 6f04ddd1fb41934c52e290bc937e45f9cd5949aa
Author: Richard Kalehoff <richardkalehoff@gmail.com>
:


richardkalehoff — bash — bash — less — 71×19
diff --git a/index.html b/index.html
index 0381211..43f5b28 100644
--- a/index.html
+++ b/index.html
@@ -15,83 +15,85 @@
<h1>Expedition</h1>
</header>
- <main>
- <h2 class="visuallyhidden">Articles</h2>
+
+ <main>
+ <h2 class="visuallyhidden">Articles</h2>
- <article>
- <header>
- <h3>Chasing the Snow</h3>
:

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

Question 1 of 4

Using what you've learned so far about `git log`'s `-p` flag, look at the commit with the SHA `50d835d`. What line number in `app.css` should you start looking at to see what has been changed?

Tip - don't forget that while looking at the `git log` output, the `d` key will scroll *down* by half a page while the `u` key will scroll *up* half a page.

- ○ line 63
- ○ line 89
- ◉ line 127 ✓
- ○ line 155

Question 2 of 4

Using `git log` and any of its flags, what code was added in by commit `4a60beb`?

- ○ color: #352d2d;
- ○ color: #250808;
- ○ color: #333333;
- ◉ color: #2e3d49; ✓

Question 3 of 4

`git log --stat` and `git log -p` are both really helpful commands. Wouldn't it be great if we could have both of their output at the same time? Hmmm...

What happens when `git log -p --stat` is run?

- ○ it displays only the patch information
- ○ it displays only the stats
- ○ it displays both with the patch info above the stats info
- ◉ it displays both with the stats info above the patch info ✓

In the video above, we looked at a commit that indents a lot of code. The patch output shows all of those lines as having been removed and then added again at their new level of indentation. Showing all of the indent changes makes it hard to tell what was actually added, though.

Question 4 of 4

What does the `-w` flag do to the patch information? For help, check this Git docs page.

- ○ it displays non-whitespace characters in blinking text
- ○ it displays non-whitespace changes in bold
- ◉ it ignores whitespace changes ✓
- ○ it shows a separate patch area with just new/removed content

That's right! `git log -p -w` will show the patch information, but will not highlight lines where *only* whitespace changes have occurred.

`git log -p` Recap

To recap, the `-p` flag (which is the same as the `--patch` flag) is used to alter how `git log` displays information:

```
$ git log -p
```

This command adds the following to the default output:

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

Viewing A Specific Commit

Too Much Scrolling

The last few quizzes in the previous section had you scrolling and scrolling through the patch output just to get to the right commit so you could see *its* info. Wouldn't it be super handy if you could just display a specific commit's details without worrying about all of the others in the repo?

There are actually two ways to do this!

- providing the SHA of the commit you want to see to `git log`
- use a new command `git show`

They're both pretty simple, but let's look at the `git log` way and then we'll look at `git show`.

You already know how to "log" information with:

- `git log`
- `git log --oneline`
- `git log --stat`
- `git log -p`

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
```

By supplying a SHA, the `git log -p` command will *start at that commit*! No need to scroll through everything! Keep in mind that it will *also* show all of the commits that were made *prior* to the supplied SHA.

New Command: `git show`

The other command that shows a specific commit is `git show`:

```
$ git show
```

Running it like the example above will only display the most recent commit. Typically, a SHA is provided as a final argument:

```
$ git show 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

You are now among the `git log`ging elite! Try your hand at a few quizzes.

Question 1 of 3

How many rulesets are added to the CSS by commit `8d3ea36`?

- ○ 1
- ◉ 2 ✓
- ○ 3
- ○ 4

Question 2 of 3

There's a commit with the message "Convert social links from text to images". How many files were changed by this commit?

- ○ 2 files
- ○ 4 files
- ◉ 5 files ✓
- ○ 9 files

Question 3 of 3

Look at commit `fdf5493`. What's the first HTML heading element that's added by this commit?

- ○ an `<h1>`
- ◉ an `<h2>` ✓
- ○ an `<h3>`
- ○ an `<h4>`

Add Commits To A Repo

This is the lesson where we finally learn how to make commits of our very own. Are you excited? Because I am!

So far, we've laid the groundwork by learning:

- The git init command to create a new repository.
- The git clone command to copy an existing repository.
- The git log command to review existing commits.
- The all-important git status command to check the status of the repository.

As a side note, remember to keep a cheat sheet with all the terms handy during this lesson, as we'll be using almost every term on it.

Now, we'll build on this foundation by introducing three new commands: **git add**, **git commit**, and **git diff**.

- With **git add**, you'll move files from the working directory to the staging index.
- With **git commit**, you'll take files from the staging index and save them in the repository, which is what actually creates a commit.
- The **git diff** command is really cool and will feel familiar because you've already seen its output before. It displays the differences between two versions of a file, and its output is identical to the output of the **git log -p** command we used in the previous lesson.

So, we've got a couple of new commands and one that's going to feel very familiar. But before we can use the commit or diff commands, we need to learn how to use git add. So, let's get cracking!

Git Add

Move To Correct Project

If you've been following along, you should have two different directories with Git projects on your computer:

- new-git-project - an empty directory that you converted to a Git repository using `git init`
- course-git-blog-project - an existing blog project you retrieved using `git clone`

To avoid any confusion with existing commits, we'll be making our commits to the new-git-project Git repository.

On the Terminal, make sure you `cd` into the `new-git-project` directory. If you don't have a `new-git-project` directory, create it now. Once you're inside the directory, run the `git init` command. If you've already run `git init` before it's ok – running `git init` multiple times doesn't cause any problems since it just re-initializes the Git directory.

Status Status Status

I've said it a number of times already, but the `git status` command will be *extremely helpful* in this lesson. You should have it as your goal to run the `git status` command both *before* and *after* any other Git command.

Git Status Output Review

This is the output:

```
On branch master

Initial commit

nothing to commit (create/copy files and use "git add" to track)
```

Notice that last line – `nothing to commit (create/copy files and use "git add" to track)`. See how it's recommending the `git add` command? that's super helpful! The `git status` output will give you advice or hints as to what you should do next.

Let's do what the feedback says and create some files.

Create An HTML File

First, create a file named `index.html`, and fill it with some starter code:

```
<!doctype html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <title>Blog Project</title>
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <meta name="description" content="">
    <link rel="stylesheet" href="css/app.css">
</head>
<body>

    <script src="js/app.js"></script>
</body>
</html>
```

Things to note, the code references a CSS file and a JavaScript file.

Now create the CSS and JavaScript files. You can leave both of these files empty. We'll add content to them in a bit.

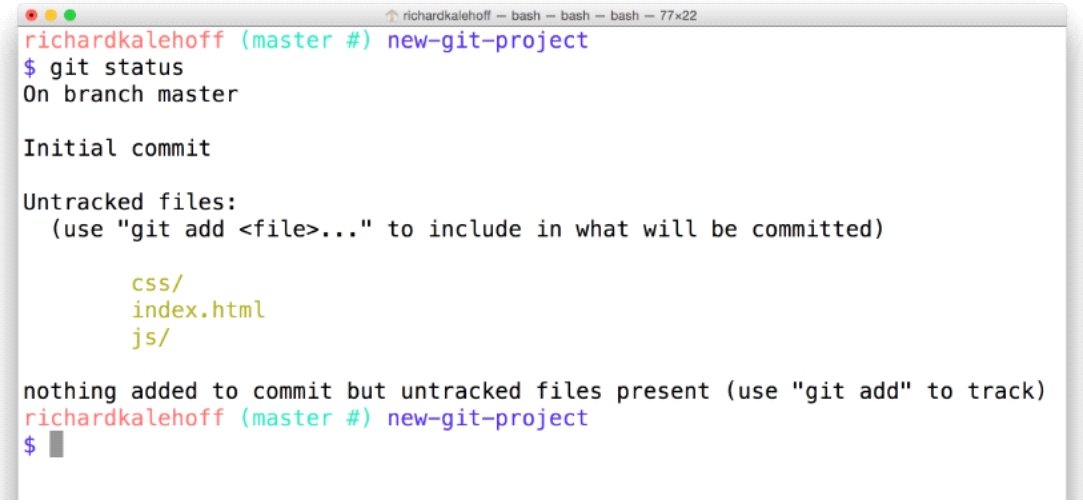
Quick Git Status Check

We just made a number of changes to the repository by adding files and content. It's time to do a quick check-in with Git:

```
$ git status
```

Here's what my Terminal displays:

Here's what my Terminal displays:



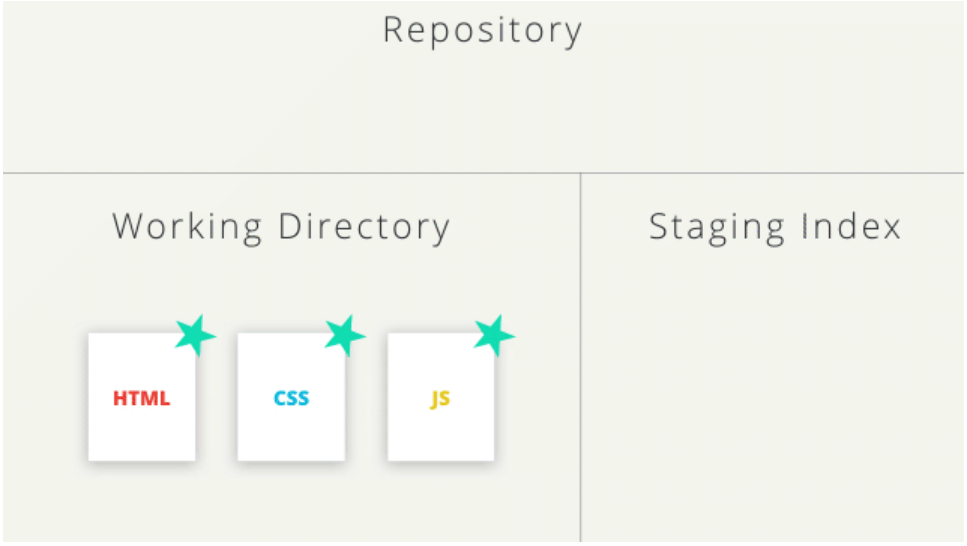
Big Picture Review

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.



Staging Files

Alrighty, it's go time! Run the following command on the Terminal which uses `git add` to add `index.html` to the Staging Index:

```
$ git add index.html
```

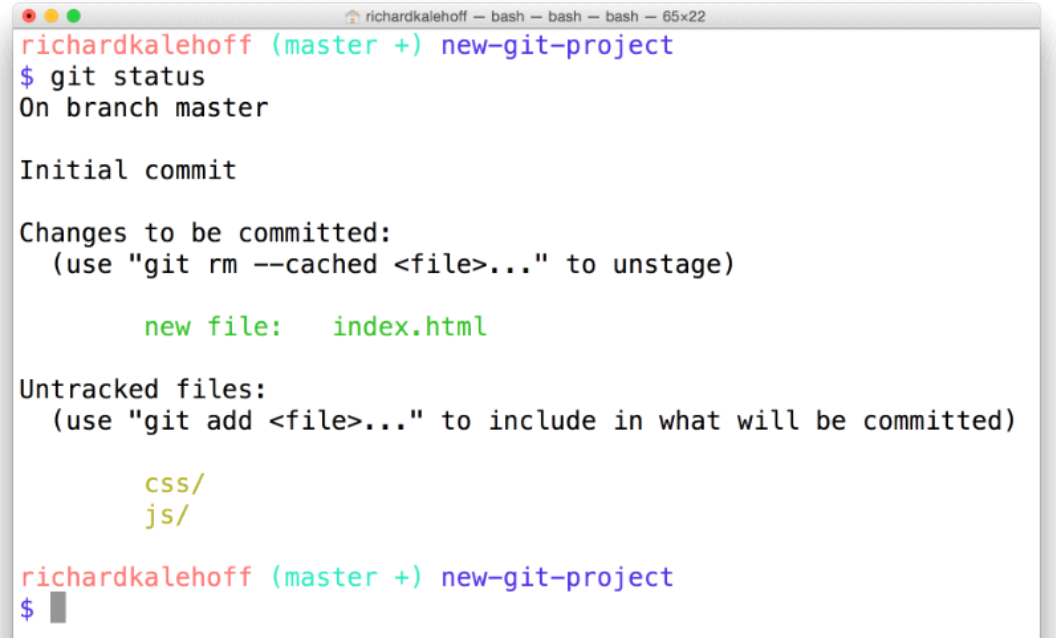
Note - we are *only* adding the `index.html` file. We'll add the CSS and JavaScript files in just a second.

Running the `git add` command produces no output (as long as there wasn't an error). So how do we have Git tell us what it did and has happened to the `index.html` file that was added? That's what `git status` does. You're probably sick of me stressing the importance of the `git status` command, but it's an extremely helpful command, especially if you're new to version control and/or the command line.

Let's check out the status of the project:

```
$ git status
```

This is the output I get:



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.

Stage Remaining Files

The `index.html` file has been staged. Let's stage the other two files. Now we *could* run the following:

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

...but that's a lot of extra typing. We could use a special command line character to help:

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 .
```

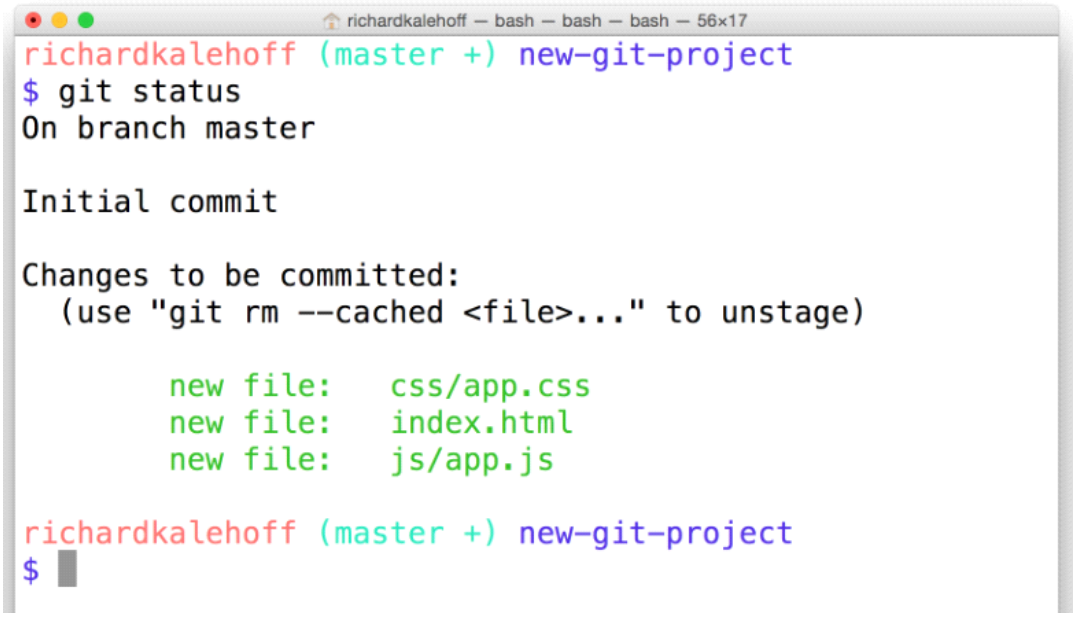
The only thing to be careful of is that you might accidentally include more files than you meant to. Right now we *want* both `css/app.css` and `js/app.js` to be staged, so running this command is fine right now. But let's say you added some images to an `img` directory but didn't want to stage them just yet. Running `git add .` *will* stage them. If you do stage files that you didn't mean to, remember that `git status` will tell you the command to use to "unstage" files.

Stage The Remaining Files

Let's use the shortcut to stage the remaining files:

```
$ git add .
```

And then a quick `git status`:



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>
```

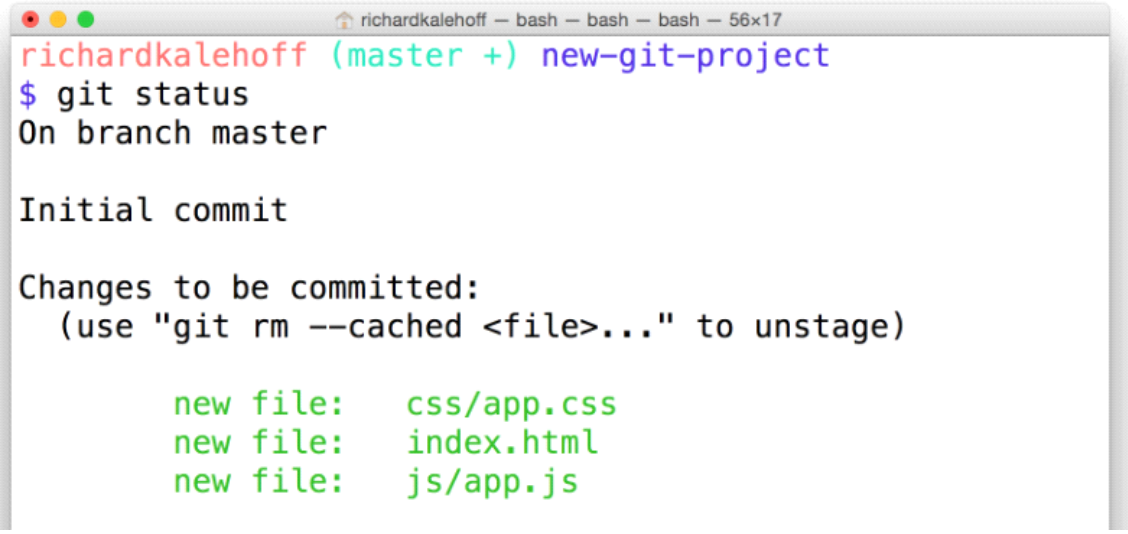
This command:

- 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)

Git Commit

One Last Git Status Check

If you haven't added any new files to the Working Directory or modified any of the existing files, nothing will have changed, but to make sure, let's run a quick `git status` again right before we make the commit just to make *absolutely sure* the project is how we left it.



Make A Commit

Ok, let's do it!

To make a commit in Git you use the `git commit` command, but don't run it just yet. Running this command will open the code editor that you configured way back in the first lesson. If you haven't run this command yet:

```
$ git config --global core.editor <your-editor's-config-went-here>
```

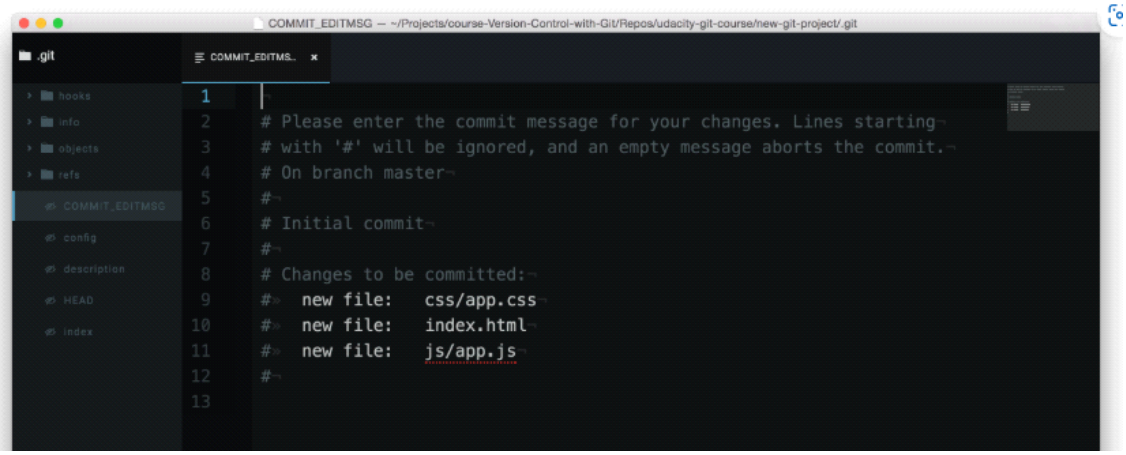
...go back to the Git configuration step and configure Git to use your chosen editor.

If you didn't do this step and you *already* ran `git commit`, then Git *probably* defaulted to using the "Vim" editor. Vim is a popular editor for people who have been using Unix or Linux systems forever, but it's not the friendliest for new users. It's definitely not in the scope of this course. Check out **this Stack Overflow post on how to get out of Vim** and return to the regular command prompt.

If you *did* configure your editor, then go ahead and make a commit using the `git commit` command:

```
$ git commit
```

Remember, your editor should pop open and you should see something like this:



Terminal Hangs

If you switch back to the Terminal for a quick second, you'll see that the Terminal is chillin' out just waiting for you to finish with the code editor that popped up. You don't need to worry about this, though. Once we add the necessary content to the code editor and finally *close the code editor window*, the Terminal will unfreeze and return to normal.

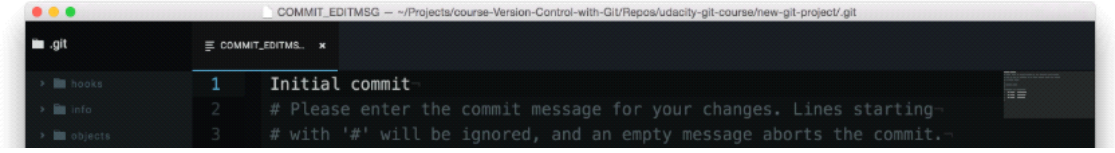
Code Editor Commit Message Explanation

Ok, switch back to the code editor. Here's what's showing in my editor:

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   new file:   css/app.css
#   new file:   index.html
#   new file:   js/app.js
#
```

The first paragraph is telling us exactly what we need to do - we need to supply a message for this commit. Also, any line that begins with the `#` character will be ignored. Farther down it says that this will be the initial commit. Lastly, it's giving us a list of the files that will be committed.

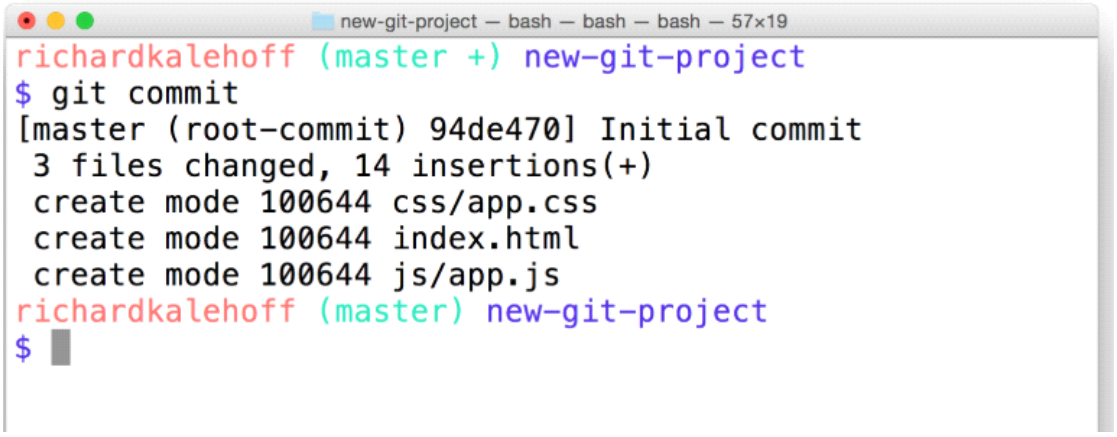
> Since this is the very first commit of the repository, we'll use the commit message "Initial commit". The text "Initial commit" isn't special, but it's the de facto commit message for the *very first* commit. If you want to use something else, feel free!
>
> Type out your commit message on the first line of the code editor:



Finish Committing

Now save the file and close the editor window (closing just the pane/tab isn't enough, you need to close the code editor window that the `git commit` command opened).

Awesome, now switch back to the Terminal and you should see something like the following:



The Terminal application after closing the code editor. It displays the SHA for the new commit as well as information about the commit like the files that were added and how many lines of code were added.

First Commit, Congrats!

You just made your first commit - woohoo! 🙌 How does it feel? Was it more towards the awe-inspiring side or the anticlimactic? Honestly, when I made my first commit, I was a bit like:

> "Wait...is that it? You just add the files you want to have committed to the Staging Area, and then you run 'git commit'?"

...and the answer to my questions are "Yes" and "Yes". That's all there is to it. At first, version control seems like this overwhelming obstacle that one must overcome to become a true programmer/developer/designer/etc. But once you get a handle on the terminology (which I think is the most challenging part), then actually using version control isn't all that challenging.

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"
```

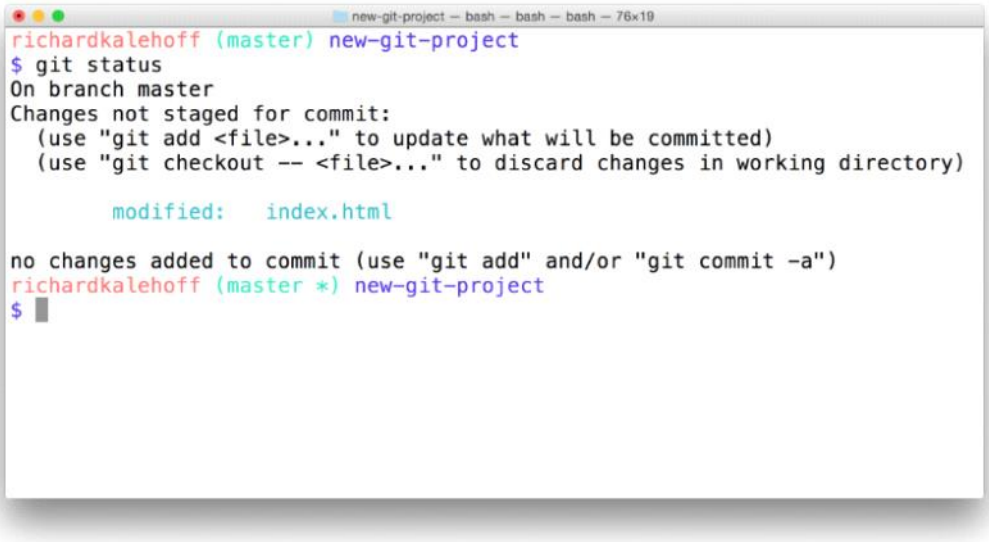
In the example above, the text `"Initial commit"` is used as the commit message. Be aware that you can't provide a description for the commit, only the message part.

2nd Commit - Add Changes

We've had a short breather, so let's make a second commit! Here, add this just inside the `body` tag in `index.html`:

```
<header>
    <h1>Expedition</h1>
</header>
```

Ok, now what do you do next? That's right, it's our good old friend `git status`!



The Terminal application showing the result of the `git status` command. It displays a "Changes not staged for commit" section that includes the modified "index.html" file.

TIP: If you run `git status` but don't see that `index.html` has changed, make sure to save the file. I modify a file and then forget to save it - *all* - *the* - *time*! I like to think that forgetting to save a file after editing it is the mark of a true professional.

Multipurpose Git Add

So we've modified our file. Git sees that it's been modified. So we're doing well so far. Now remember, to make a commit, the file or files we want committed need to be on the Staging Index. What command do we use to move files from the Working Directory to the Staging Index? You got it - `git add`!

Even though we used `git add` to add *newly created files* to the Staging Index, we use the same command to move *modified files* to the Staging Index.

Use the `git add` command to move the file over to the Staging Index, now. Verify that it's there with `git status`.

Second Commit

Now that we have a file with changes we can commit, let's make our second commit! Use the `git commit` command to make a commit. Use the commit message `Add header to blog`.

Now you might be asking yourself, "Why did Richard pick that as the commit message to use?" or "What makes a good commit message?". These are fantastic questions that we'll be looking at in the next concept!

Quiz Question

Let's say you have a personal blog and want to change the color of the article headings. You

- **edit the HTML files to give each heading a class**
- **edit the CSS file to add the new class and give it a color**
- **you save all files**
- **you run `git commit` on the terminal**

Will your code editor open up to let you type out the commit message?

- ○ Yes
- ◉ No ✓

That's right! Running `git commit` will *instead* display the output of `git status` and "no changes added to commit". It did this because you did not use `git add` to move the files from the Working Directory to the Staging Index.

What To Include In A Commit

I've been telling you what files to create, giving you the content to include, and telling you when you should make commits. But when you're on your own, how do you know what you should include in a commit and when/how often you should make commits?

The goal is that each commit has a single focus. Each commit should record a single-unit change. Now this can be a bit subjective (which is totally fine), but each commit should make a change to just one aspect of the project.

Now this isn't limiting the number of lines of code that are added/removed or the number of files that are added/removed/modified. Let's say you want to change your sidebar to add a new image. You'll probably:

- add a new image to the project files
- alter the HTML
- add/modify CSS to incorporate the new image

A commit that records all of these changes would be totally fine!

Conversely, a commit shouldn't include unrelated changes - changes to the sidebar *and* rewording content in the footer. These two aren't related to each other and shouldn't be included in the same commit. Work on one change first, commit that, and then change the second one. That way, if it turns out that one change had a bug and you have to undo it, you don't have to undo the other change too.

The best way that I've found to think about what should be in a commit is to think, "What if all changes introduced in this commit were erased?". If a commit were erased, it should only remove one thing.

> Don't worry, commits don't get randomly erased.

> In a later lesson, we'll look at using Git to undo changes made in commits and how to manually, carefully remove the last commit that was made.

Git Commit Recap

The `git commit` command takes files from the Staging Index and saves them in the repository.

```
$ git commit
```

This command:

- will open the code editor that is specified in your configuration
  - (check out the Git configuration step from the first lesson to configure your editor)

Inside the code editor:

- a commit message must be supplied
- lines that start with a `#` are comments and will not be recorded
- save the file after adding a commit message
- close the editor to make the commit

Then, use `git log` to review the commit you just made!

Commit Messages

Good Commit Messages

Let's take a quick stroll down Stickler Lane and ask the question:

> How do I write a *good* commit message? And why should I care?

These are *fantastic* questions! I can't stress enough how important it is to spend some time writing a *good* commit message.

Now, what makes a "good" commit message? That's a great question and has been **written about a number of times**. Here are some important things to think about when crafting a good commit message:

Do

- do keep the message short (less than 60-ish characters)
- do explain *what* the commit does (not *how* or *why*!)

Do not

- do not explain *why* the changes are made (more on this below)
- do not explain *how* the changes are made (that's what `git log -p` is for!)
- do not use the word "and"
  - if you have to use "and", your commit message is probably doing too many changes - break the changes into separate commits
  - e.g. "make the background color pink *and* increase the size of the sidebar"

The best way that I've found to come up with a commit message is to finish this phrase, "This commit will...". However, you finish that phrase, use *that* as your commit message.

Above all, ***be consistent*** in how you write your commit messages!

Question 1 of 3

Reviewing the guidelines on what makes a good commit message, is the following commit message good?

"Update the footer to copyright information"

- (●) Yes ✓
- ( ) No

Question 2 of 3

Is the following a good commit message?

"Add a

tag to the body"

- ( ) Yes
- (●) No ✓

Question 3 of 3

Is the following a good commit message?

"Add changes to app.js"

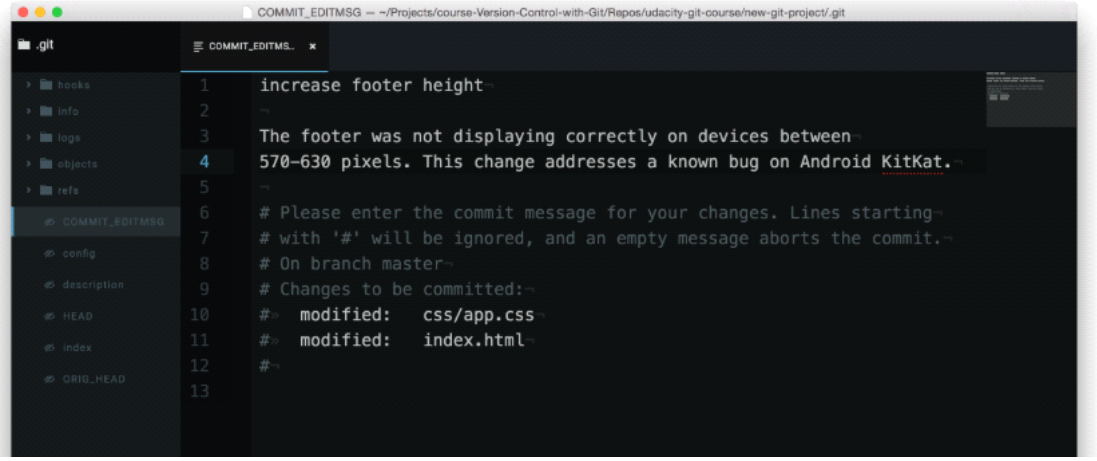
- ○ Yes
- ◉ No ✓

Explain the *Why*

If you need to explain *why* a commit needs to be made, you can!

When you're writing the commit message, the first line is the message itself. After the message, leave a blank line, and then type out the body or explanation including details about why the commit is needed (e.g. URL links).

Here's what a commit message edit screen might look like:



This details section of a commit message *is* included in the `git log`. To see a commit message with a body, check out the Blog project repo and look at commit `8a11b3f`.

Only the message (the first line) is included in `git log --oneline`, though!

Udacity's Commit Style Requirements

As I've mentioned, there are a number of ways to write commit messages. If you're working on a team, they might already have a predetermined way of writing commit messages. Here at Udacity, we have our own standard for commit messages. You can check it out on our Git Commit Message Style Guide(opens in a

new tab).
If you haven't chosen a commit message style, feel free to use ours. But if you're working on an existing project, use their existing style; it's much more important to be consistent with your actual team than to be consistent with us!

Git Diff Up Next!

In the next section, we'll look at a new tool (with a familiar output!). This tool will tell us what changes we've made to files *before* the files have been committed!

Git Diff

Why Do We Need This

You might be like me where I start work on the next feature to my project at night, but then go to bed before I actually finish. Which means that, when I start working the next day, there are uncommitted changes. This is fine because I haven't finished the new feature, but I can't remember exactly what I've done since my last commit. `git status` will tell us what files have been changed, but not what those changes actually were.

The `git diff` command is used to find out this information!

`git diff`

The `git diff` command can be used to see changes that have been made but haven't been committed, yet.

```
$ git diff
```

To see `git diff` in action, we need some uncommitted changes! In `index.html`, let's reword the heading. Change the heading from "Expedition" to "Adventure". Save the file and run `git diff` on the Terminal.

You should see the following:

Wow, doesn't that look familiar! It's the same output that we say with `git log -p`! Wanna know a secret? `git log -p` uses `git diff` under the hood. So you've actually already learned how to read the output of `git diff`!

If you don't remember what the different sections are, check out the Annotated "git log -p" Output from the previous lesson.

💡 These Changes Were Not Committed 💡

The changes in this section were used to demo the output of `git diff`. They were not committed to the repository. If you'd like, you can definitely commit the changes to the repository, just know that your `git log` will look slightly different from mine because it includes this extra 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

Having Git Ignore Files

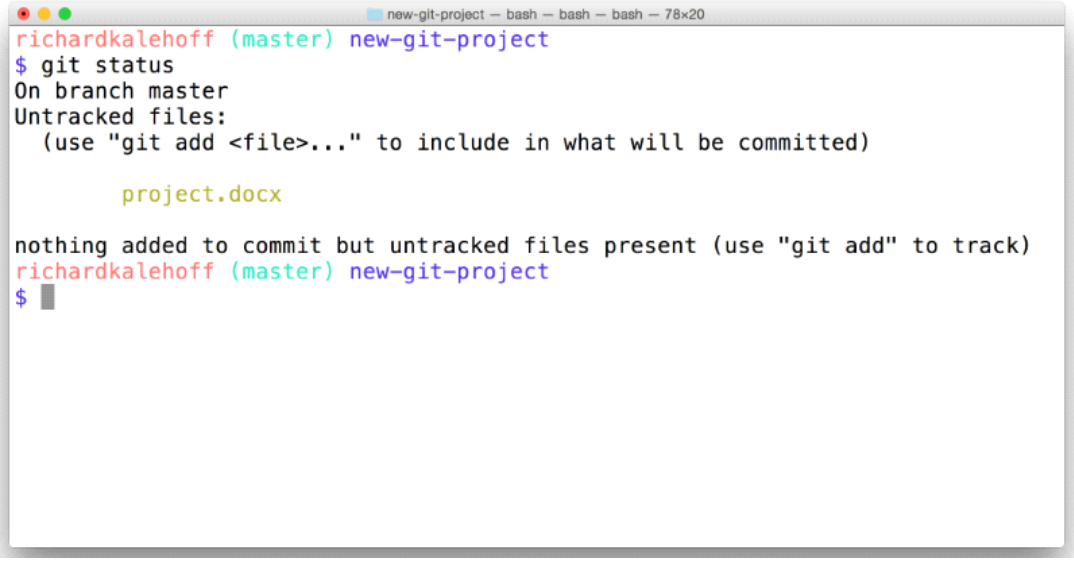
Why Should Files Be Ignored

Remember a couple sections back when we were learning about `git add`? Instead of adding the files one by one, there was a special character that we could use to indicate the current directory and all subdirectories. Do you remember what that character is?

That's right, the period (`.`)!

The Problem

Let's say you add a file like a Word document to the directory where your project is stored *but don't want it added to the repository*. (You can simulate adding a Word document by running `touch project.docx`) Git will see this new file, so if you run `git status` it'll show up in the list of files.



The Terminal application showing the output of the `git status` command. The output shows a new Word document that is in Git's "Untracked files" section.

> The potential problem with having this file in your project is that, because `git add .` adds *all* files, the Word document might get accidentally committed to the repository.

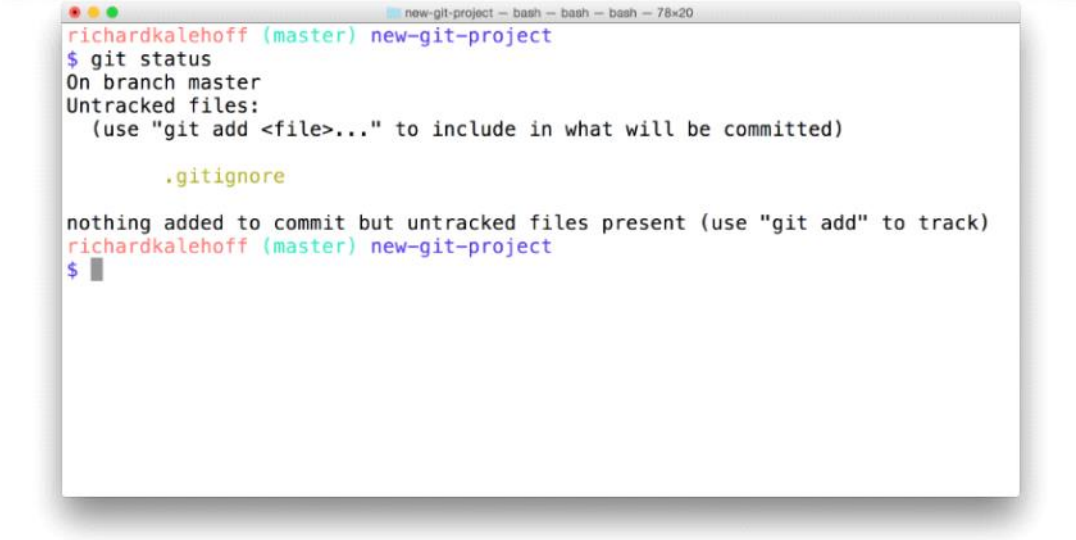
Git Ignore

If you want to keep a file in your project's directory structure but make sure it isn't accidentally committed to the project, you can use the specially named file, `.gitignore` (note the dot at the front, it's important!). Add this file to your project in the same directory that the hidden `.git` directory is located. All you have to do is list the *names* of files that you want Git to ignore (not track) and it will ignore them.

Let's try it with the "project.docx" file. Add the following line inside the `.gitignore` file:

```
project.docx
```

Now run `git status` and check its output:



The Terminal application showing the output of `git status`. The Word document is no longer listed as an untracked file. The new ".gitignore" file is listed, though.

Git knows to look at the contents of a file with the name `.gitignore`. Since it saw "project.docx" in it, it ignored that file and doesn't show it in the output of `git status`.

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).

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
```

Question 1 of 2

Which of the following files will be ignored if `*.png` is entered into the `.gitignore` file?

- [ ] ocean.jpg
- [x] trees.png ✓
- [ ] png-format.pdf
- [ ] not-a-png.jpeg
- [x] bg-pattern.png ✓
- [ ] logo.gif
- [x] LOUDFILE.PNG ✓

Question 2 of 2

If you ask Git to ignore "be?rs", which of the following filenames will be ignored?

- [x] bears ✓
- [ ] beavers
- [ ] BeArS
- [x] beers ✓
- [ ] boars

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.

Outro

We just covered a ton of information in this lesson! Here's a quick recap:

- We explored using **git add** to stage content.
- We used **git commit** to save changes to the repository and discussed what

makes a good commit message.
- We learned how **git diff** can help us review changes that haven't been committed yet.
- Finally, we looked at the **.gitignore** file, which tells Git which files to avoid tracking.

With this knowledge, you now know how to:
- Create or clone a repository.
- Make professional commits with clear, descriptive commit messages.
- Review existing commits.

These skills allow you to jump in and start working on any Git repository with confidence.
In the next lesson, we'll dive into **branching**, a powerful concept that lets you work on a project in multiple states simultaneously. Branching will take your Git skills to the next level—let's go check it out!

Tagging, Branching, and Merging

Now that you're a pro at making commits, it's time to level up again! This next step will give you true version control superpowers. In this lesson, we'll explore four powerful Git commands: **git tag**, **git branch**, **git checkout**, and **git merge**.
What You'll Learn:

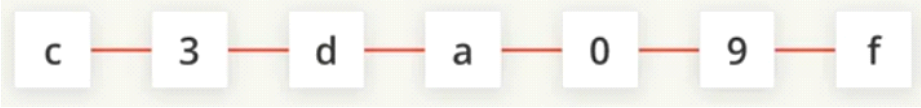
1. **git tag**:
    - Add tags to specific commits.
    - Tags are extra labels that provide useful information, such as marking a commit as the "beta release."
2. **git branch**:
    - Create branches to develop different features of your project in parallel.
    - Branches help you avoid confusion about which commits belong to which feature.
3. **git checkout**:
    - Switch between different branches and tags.
4. **git merge**:
    - Combine changes from different branches automatically.
    - This is an incredibly powerful feature that streamlines development.

These commands will help keep your project organized and speed up development. Let's start with the easiest command: **git tag**.

Tagging

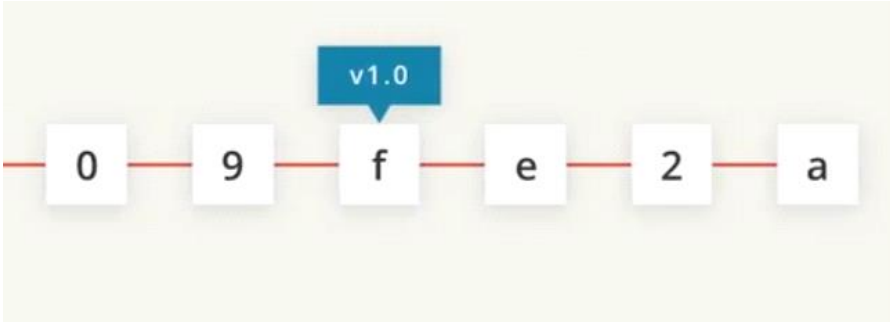
Let's start at the beginning—because it's a very good place to start!
We have a project with some files. We add these files together and make a commit, which has a unique SHA identifier. To simplify, let's shrink this down and display just the first letter of the SHA. Great! Now we have this one commit.

Next, let's make a few more changes to the project and commit them. Awesome! Now, let's say the most recent commit (starting with the letter **f**) represents version 1.0 of our project. How can we indicate this in Git?



We don't want to write this in the commit message, and listing it in the body of the commit message isn't ideal either. Instead, we can use a **tag** to mark this commit. Tags allow you to highlight specific commits and label them with meaningful information, such as **v1.0**.

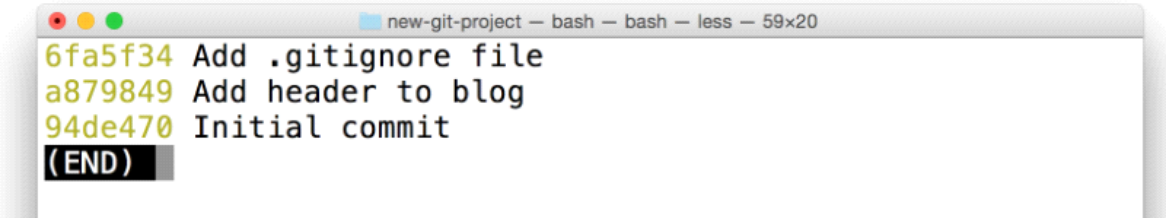
Let's add another commit. Notice how the tag stays locked to the original commit, even as more commits are added to the repository. Tags are a great way to mark important milestones in your project's history.



Where Are We?

You can do these steps in either project, but I'll be doing them in the `new-git-project` project.

Let's take a look at the log of the project so far:



The Terminal application showing the output from running git log --oneline*.*

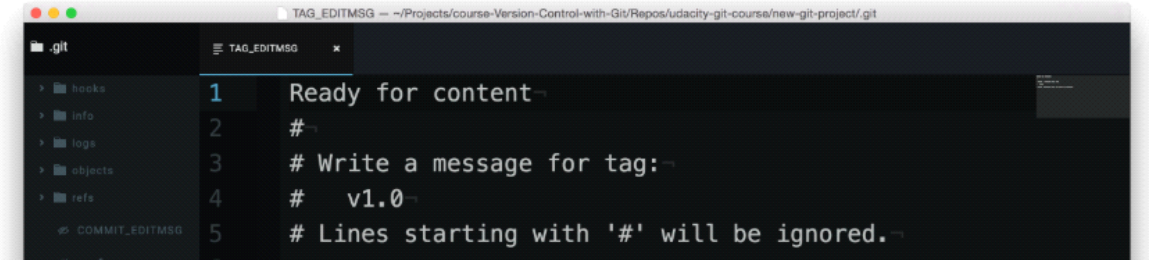
Git Tag Command

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
```

This will open your code editor and wait for you to supply a message for the tag. How about the message "Ready for content"?



Code editor waiting for the tag's message to be supplied.

CAREFUL: In the command above (`git tag -a v1.0`) the `-a` flag is used. This flag tells Git to create an *annotated* flag. If you don't provide the flag (i.e. `git tag v1.0`) then it'll create what's called a *lightweight* tag.

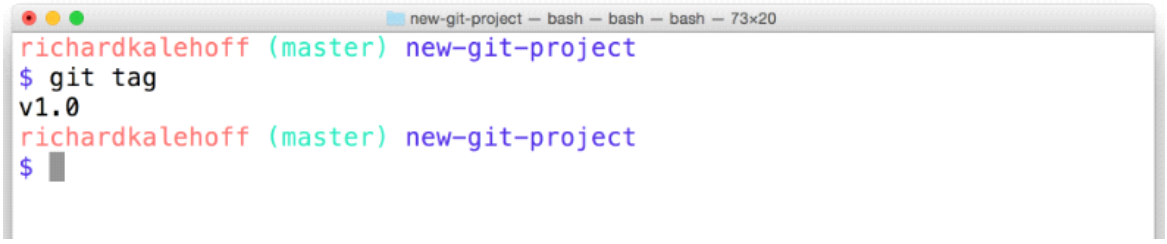
Annotated tags are recommended because they include a lot of extra information such as:

- the person who made the tag
- the date the tag was made
- a message for the tag

Because of this, you should always use annotated tags.

Verify Tag

After saving and quitting the editor, nothing is displayed on the command line. So how do we know that a tag was actually added to the project? If you type out just `git tag`, it will display all tags that are in the repository.



The Terminal application showing the output of the git tag *command. The tag* v1.0 *is listed.*

So we've verified that it's in the repository, but let's actually see *where* it is inside the repository. To do that, we'll go back to our good old friend, `git log`!

Git Log's --decorate Flag

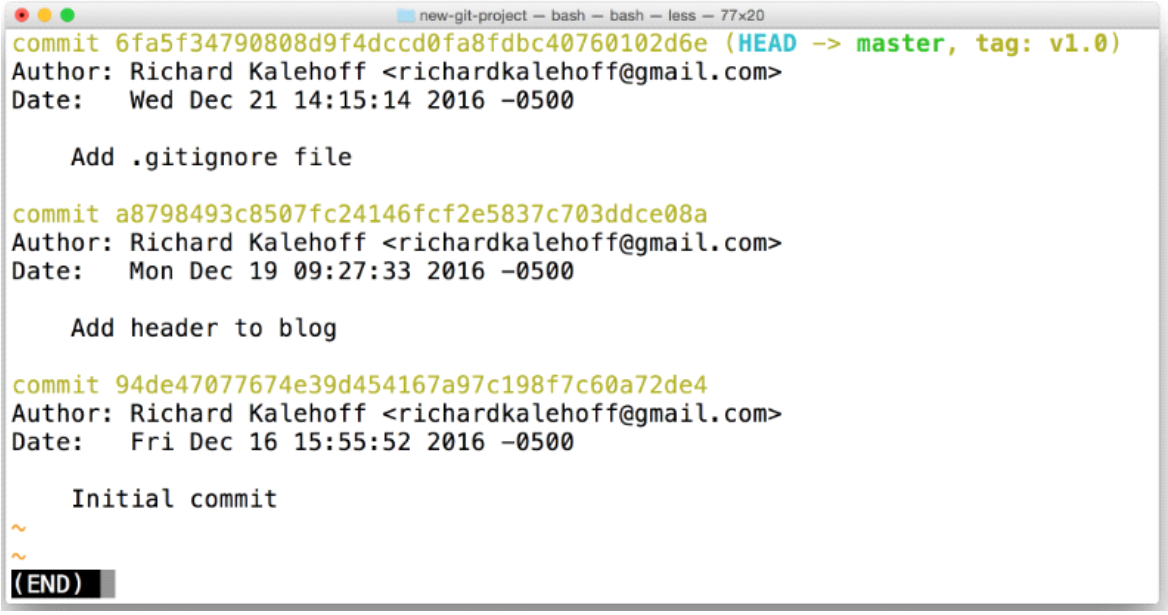
As you've learned, `git log` is a pretty powerful tool for letting us check out a repository's commits. We've already looked at a couple of its flags, but it's time to add a new one to our toolbelt. The `--decorate` flag will show us some details that are hidden from the default view.

Try running `git log --decorate` now!

💡 `--decorate` Flag Changes in Git 2.13 💡

In the 2.13 update to Git, the `log` command has changed to automatically enable the `--decorate` flag. This means that you do not need to include the `--decorate` flag in your command, since it is automatically included, anyway! So the following commands result in the exact same output:

```
$ git log --decorate
$ git log
```



The Terminal application showing the output of the git log --decorate *command. The log output now displays the newly created tag.*

The tag information is at the very end of the first line:

```
commit 6fa5f34790808d9f4dccd0fa8fdbc40760102d6e (HEAD -> master, tag: v1.0)
```

See how it says `tag: v1.0`? That's the tag! Remember that tags are associated with a specific commit. This is why the tag is on the same line as the commit's SHA.

HEAD -> master?

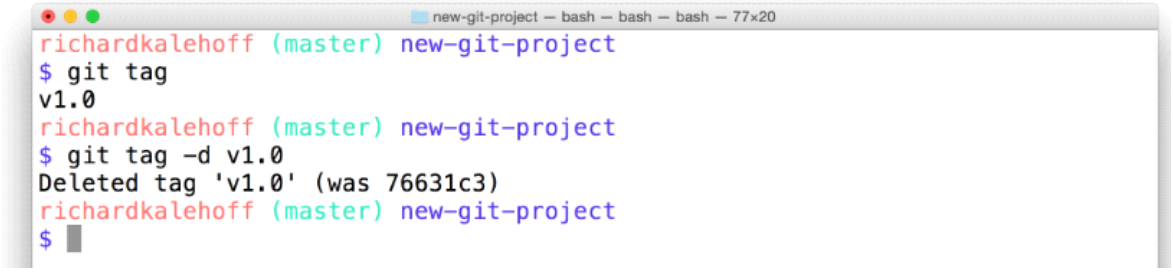
Did you notice that, in addition to the tag information being displayed in the log, the `--decorate` also revealed `HEAD -> master`? That's information about a *branch*! We'll be looking at branches in Git, next.

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
```



The Terminal application showing the removal of a tag by using the `-d` *flag. The command that is run is* `git tag -d v1.0`.

Question 1 of 2

By default, a Git tag will not appear in a log. What flag must be used to display the tag information in the output of `git log`?

- ○ --show-tags
- ○ --tags
- ○ --display-all
- ◉ --decorate ✓

Question 2 of 2

Which of the following will delete the tag `v-1`?

- [x] git tag --delete v-1 ✓
- [ ] git remove v-1
- [x] git tag -d v-1 ✓
- [ ] git delete v-1

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 -a v1.0 a87984
```

(after popping open a code editor to let you supply the tag's message) this command will tag the commit with the SHA `a87084` with the tag `v1.0`. Using this technique, you can tag any commit in the entire git repository! Pretty neat, right?...and it's just a simple addition to add the SHA of a commit to the Git tagging command you already know.

Tag Older Commit?

Using the following `git log --oneline` information, what command would you run to give the commit with the message "style page header" a tag of `beta` ?

```
2a9e9f3 add breakpoint for large-sized screens
137a0bd add breakpoint for medium-sized screens
c5ee895 add space around page edge
b552fa5 style page header
f8c87c7 convert social links from text to images
```

---

git tag -a beta b552fa5

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*

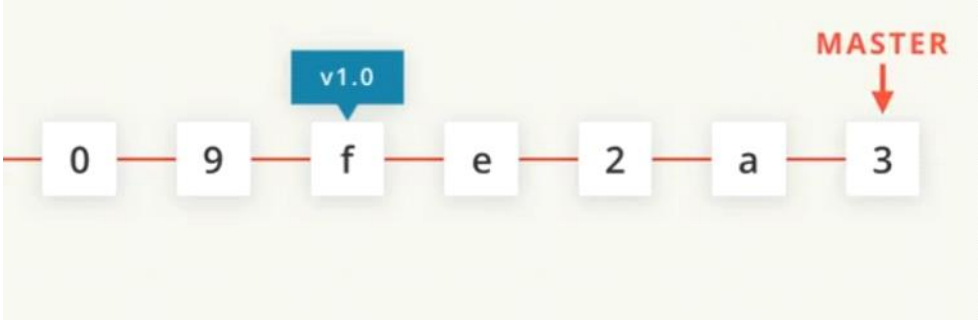
Branching

This is the sample Git repository we built up in the previous lesson. It contains several commits, a tag on one of the commits, and a branch that has been hidden until now. Let's explore how branches and tags work in Git.

Branches in Git

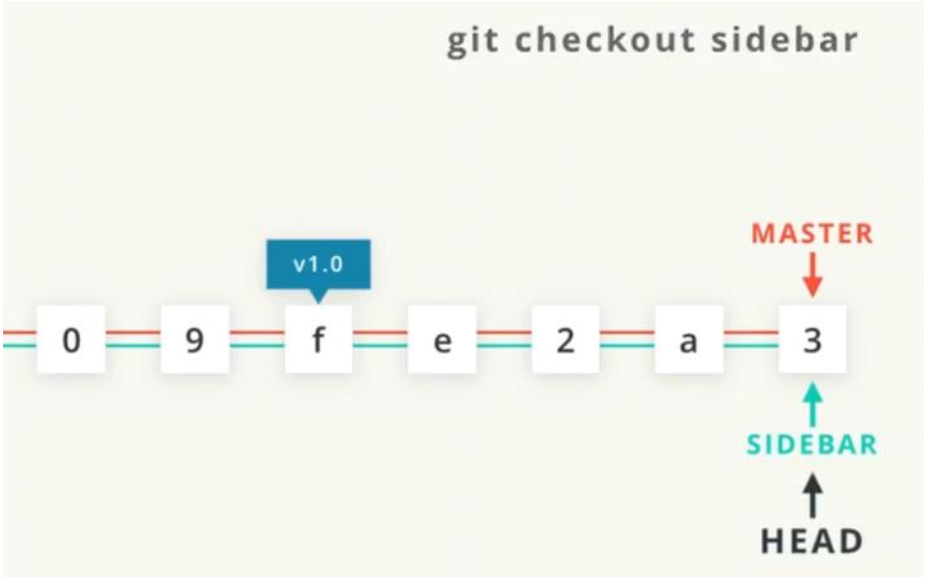
- By default, the first branch in a Git repository is named **master**, but this is not a special name—it's just the default provided by Git.
- When a commit is made, it is added to the active branch, and the branch pointer moves to point to the new commit.

- Unlike tags, which are permanent pointers to specific commits, branches move as new commits are added.



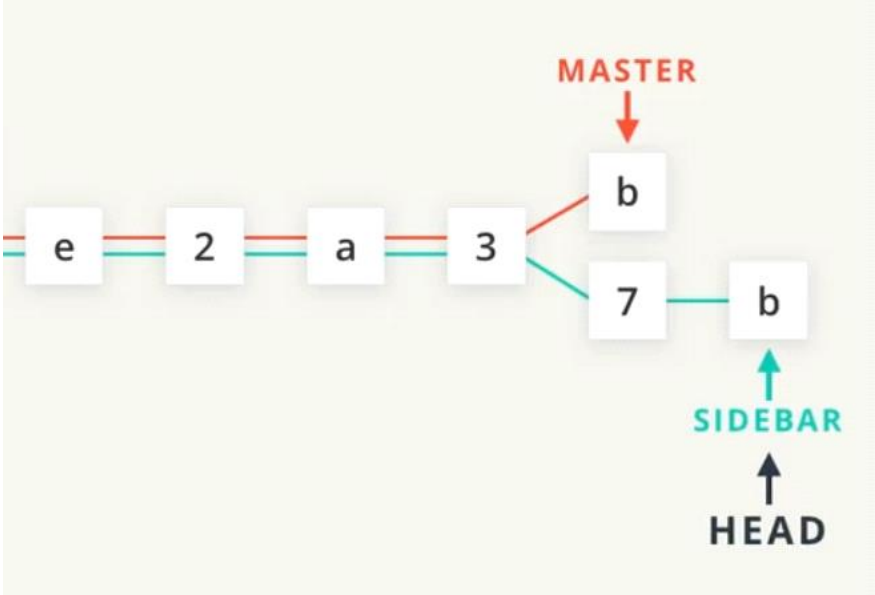
Creating and Switching Branches

- Let's create a new branch called **sidebar**.
- Branches are powerful because they allow us to work on the same project in isolated environments.
- To switch between branches, we use the **git checkout** command. For example:
  - git checkout sidebar switches to the **sidebar** branch.
  - git checkout master switches back to the **master** branch.
- The **HEAD** pointer indicates the active branch. Commits are added to the branch that **HEAD** points to.

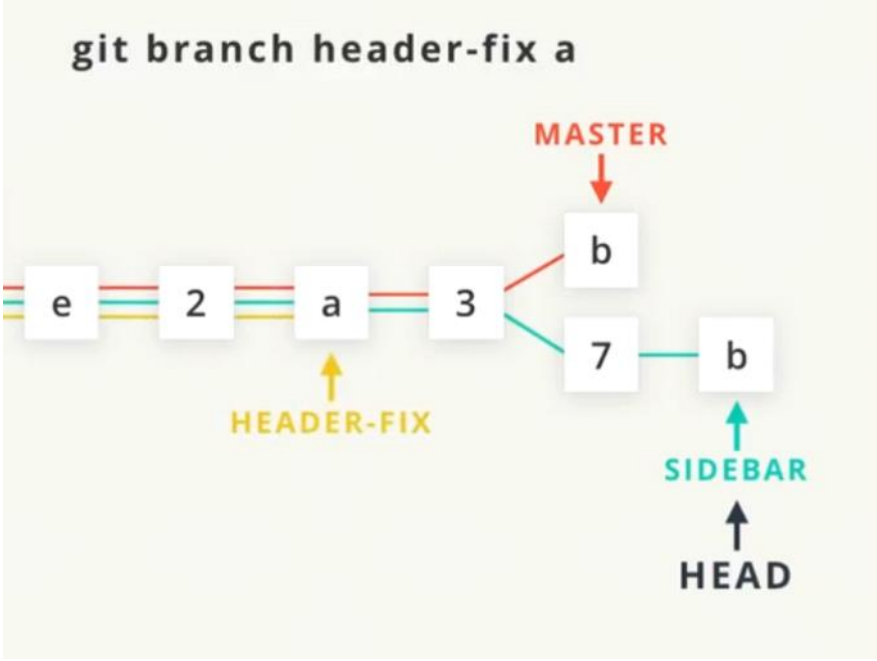


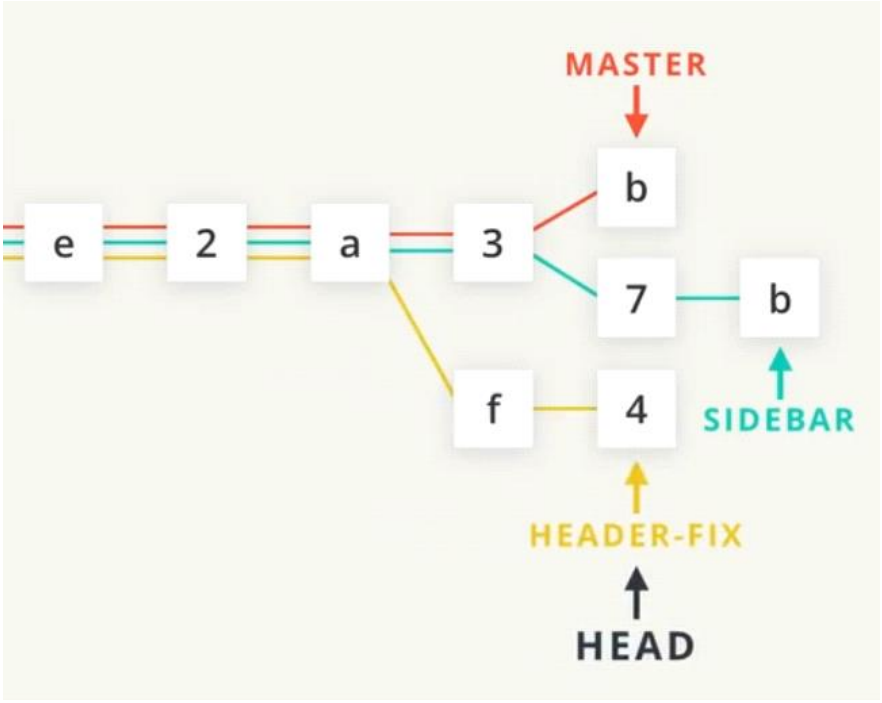
Working with Multiple Branches

- Let's switch to the **sidebar** branch and add a commit there. Now, **HEAD** points to **sidebar**, and new commits will be added to this branch.



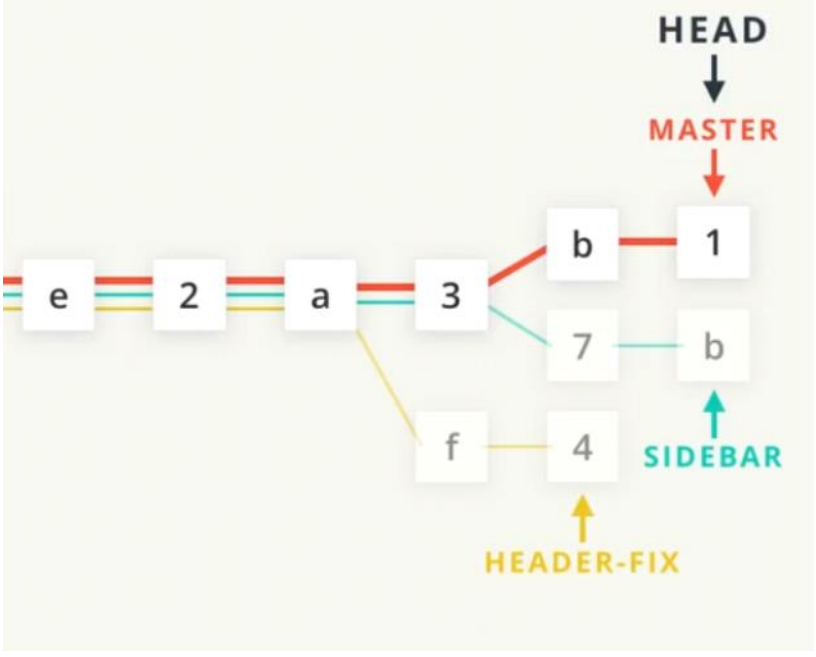
- We can create as many branches as we want and choose where they start. For example, if there's a problem with the header introduced in commit **a**, we can:
    1. Create a new branch at commit **a** using the **git branch** command.
    2. Switch to the new branch using **git checkout**.
    3. Make the necessary changes to fix the problem.



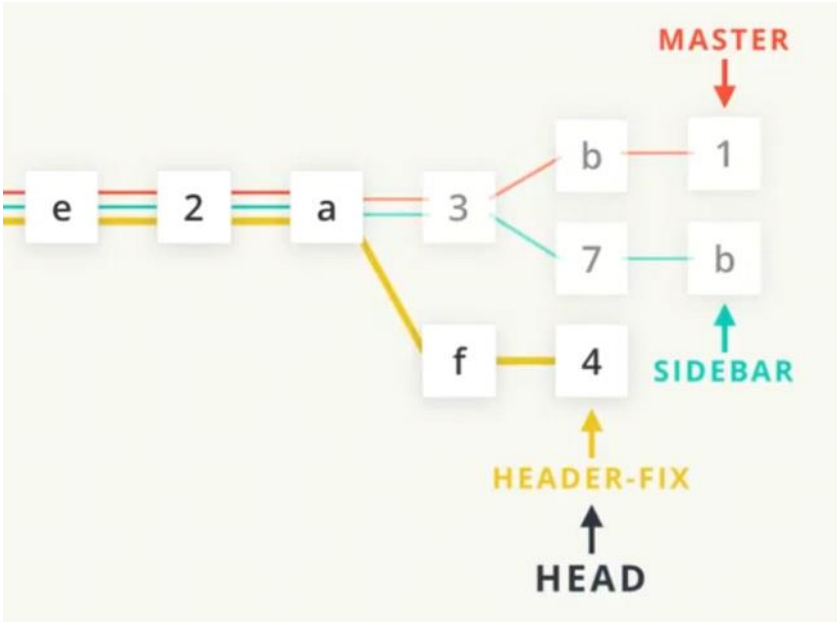


Branch Isolation

- When you switch to a branch, Git only shows the changes that are part of that branch. For example:
  - If you're on the **master** branch, commits from other branches (e.g., **f**, **4**, **7**, and **b**) will not appear in your files.
  - These commits are safely stored in the repository and can be accessed by switching to the appropriate branch.



- For instance, switching to the **header-fix** branch will make the commits on that branch appear in your file system and code editor.



So that's the big picture of how branches work and how to switch between branches. Did you know that you've already seen the master branch on the command line? Because of the setup files you added in the first lesson, the current branch is displayed right in the command prompt.

The Terminal application showing the current branch in the command prompt. The current branch is the "master" branch.

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

If we type out just `git branch` it will list out the branches in a repository:

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
```

Question 1 of 2

Remember that there are a number of branches in the repository, but that the command prompt displays the *current branch*.

Now that we just created a new "sidebar" branch, does the command prompt display `sidebar` or `master`?

- ○ sidebar
- ◉ master ✓

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
```

It's important to understand how this command works. Running this command will:

- remove all files and directories from the Working Directory that Git is tracking
  - (files that Git tracks are stored in the repository, so nothing is lost)
- go into the repository and pull out all of the files and directories of the commit that the branch points to

So this will remove all of the files that are referenced by commits in the master branch. It will replace them with the files that are referenced by the commits in the sidebar branch. This is very important to understand, so go back and read these last two sentences.

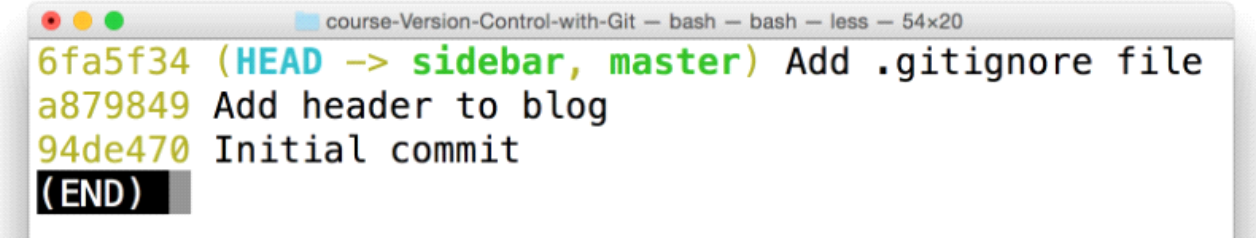
The funny thing, though, is that both `sidebar` and `master` are pointing *at the same commit*, so it will look like nothing changes when you switch between them. But the command prompt will show "sidebar", now:

Branches In The Log

The branch information in the command prompt is helpful, but the clearest way to see it is by looking at the output of `git log`. But just like we had to use the `--decorate` flag to display Git tags, we need it to display branches.

```
$ git log --oneline --decorate
```

This is what my log output displays (yours might look different depending on what commits you've made):



In the output above, notice how the special "HEAD" indicator we saw earlier has an arrow pointing to the sidebar branch. It's pointing to sidebar because the sidebar branch is the current branch, and any commits made right now will be added to the sidebar branch.

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.

Question 2 of 2

From what you know about both the `git branch` and `git tag` commands, what do you think the following command will do?

```
$ git branch alt-sidebar-loc 42a69f
```

- ○ will create a branch `alt` at the same commit as the `master` branch
- ○ will create the 3 branches `alt`, `sidebar`, `loc`
- ○ will move the master branch to the commit with SHA `42a69f`
- ◉ will create the `alt-sidebar-loc` branch and have it point to the commit with SHA `42a69f` ✓

Which Branch Is Active?

Given the following output from `git branch`:

```
$ git branch
  barbara
* footer-fix
  master
  richard
  sidebar
  social-icons
```

Which branch is the active branch?

footer-fix

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 Branch Recap

To recap, the `git branch` command is used to manage branches in Git:

```
# to list all branches
$ git branch

# to create a new "footer-fix" branch
$ git branch footer-fix

# to delete the "footer-fix" branch
$ git branch -d footer-fix
```

This command is used to:

- list out local branches
- create new branches
- remove branches

Branching Effectively

The Game Plan

Right now we have all of our code on the `master` branch (which is the default branch). We're about to work with branches, by:

- adding content to them
- creating new branches
- switching back and forth between them

Let's use branches to make the following changes:

1. on the master branch - add a default color to the page

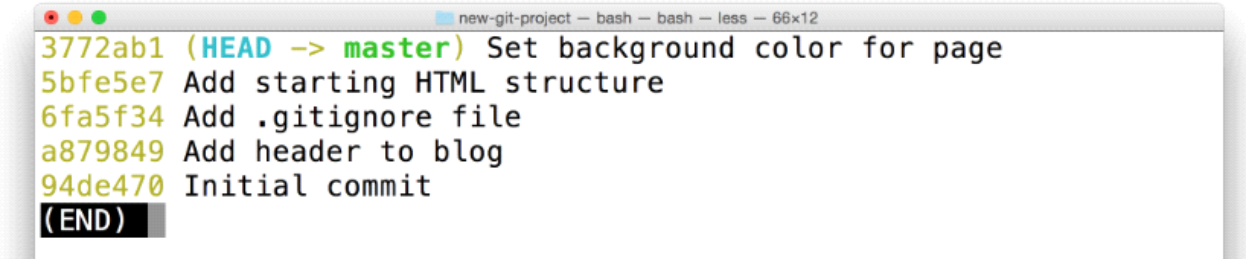
- create a sidebar branch - create a sidebar for the page
- on the master branch - change the heading of the page
- on the sidebar branch - add more content to the sidebar
- create a footer branch - add social links to the footer

Change 1 - Add Page Color

Make sure you're on the `master` branch and add the following content to `css/app.css`:

```
body {
    background-color: #00cae4;
}
```

Save the file. Then add the file to the staging index and commit it to the repository.

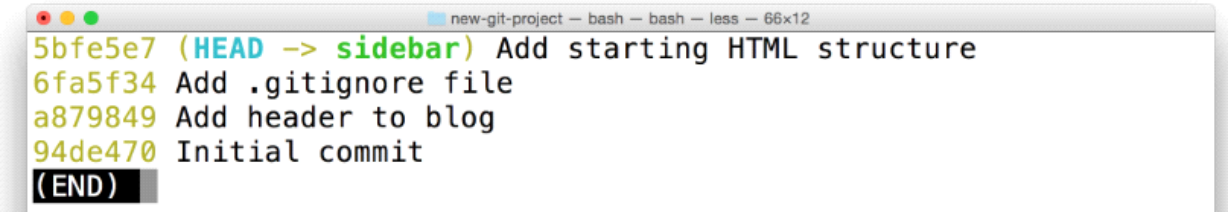


Change 2 - Add Sidebar

Let's add a sidebar to the page. But let's say that we're not really sure if we like the new background color. So we'll place the sidebar branch on the commit *before* the one that sets the page's color. Your SHAs will be different, but, for me, the commit that's before the one that adds the color has a SHA of `5bfe5e7`. So adding the branch to that commit would look like:

```
$ git branch sidebar 5bfe5e7
```

Now use the `git checkout` command to switch to the new `sidebar` branch. Running a `git log --oneline --decorate` shows me:



Did you notice that the `master` branch does not display in the output? Where did it go!?! Is it lost? Don't worry, it's still there, we'll see how to get it to display in just a second.

But first, in your code editor, switch to the `app.css` file. Notice that it does not have the CSS we previously entered! Because of this, if you load the project up in the browser, the page won't have a colored background. This makes sense since the CSS file is empty, but do you know why?

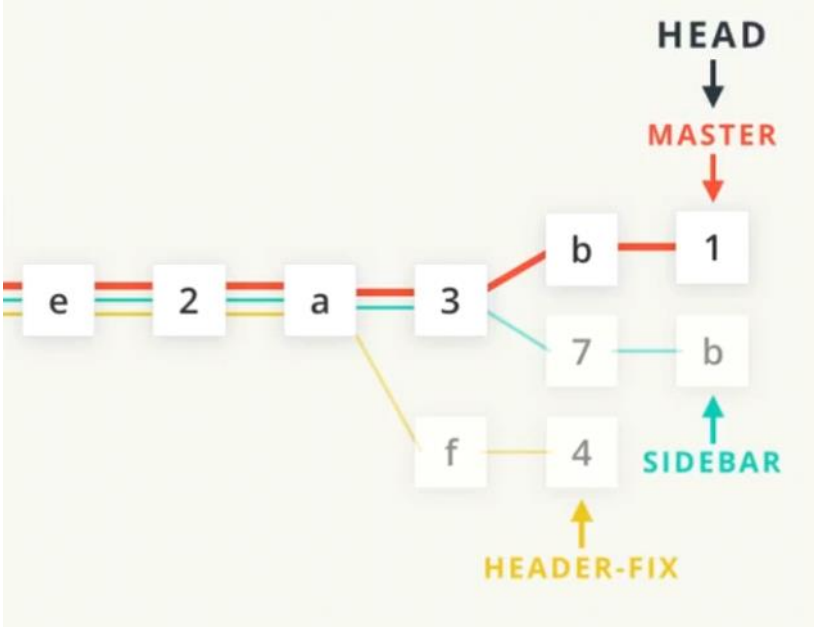
Question 1 of 2

Thinking back to the branching repository video from the previous lesson, why would the CSS file be empty?

- ○ The content has been erased.
- ◉ The content is stored safely on another branch. ✓
- ○ The content is in a temporary file that needs to be saved.
- ○ A bear ate it.

This is a very important thing to understand when starting to work with branches. If

content is stored on one branch and their commits on another branch, those other commits are invisible until we switch to that other branch. When we switch, then we'll see those commits. But the original commits will no longer appear in our code editor. They aren't lost. They're just not displaying since we've checked out a different branch.



Create a sidebar by adding the following `<aside>` code to the HTML file:

```
<div class="container">
    <main>

    </main>
</div>

<!-- start of new content -->
<aside>
    <h2>About Me</h2>

    <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit. Eos, debit
</aside>
<!-- end of new content -->

<footer>
    Made with ♥ @ Udacity
</footer>
```

I added my `<aside>` content next to the `<main>` element as a fellow child of the `<div class="container">` element.

Feel free to add any content inside the `<aside>` element!

⚠ Don't Change the CSS

WARNING: It's very important that you *do not change the CSS file*.

We'll change it later, but if you make a change right now, we'll end up having what's known as a "merge conflict". We'll manually cause a merge conflict in a bit, but we don't want to have one right now, so just don't make any changes to the CSS file, yet.

Change 3 - Change Heading On Master

Let's switch back to the master branch and update the page heading.

Use the `git checkout` command to switch back to the `master` branch. (Notice that the HTML for the new sidebar is no longer there(!) because all that code is stored safely on the `sidebar` branch.)

Now change the `<h1>` heading of the page from "Expedition" to something else. How about something exciting like the word "Adventure"!?!

Question 2 of 2

Pop quiz time! How do you have Git show you the changes you've saved, but not yet committed?

- ○ git show --diff
- ○ git log -p
- ◉ git diff ✓
- ○ git log --stat

Change 4 - Add More Content To Sidebar

Switch back to the `sidebar` branch (notice, again, that content we've added to the `master` branch isn't visible on the `sidebar` branch).

Now just add some content inside the `<aside>` element. Add something about yourself - your favorite movie or book (my favorite is LOTR!). Anything will work, you just need to add some content.

Again, make sure that you do not make changes to the CSS file.

Now save the `index.html` file and make a commit.

Change 5 - Add Social Links To Footer

We've made a number of changes, and we're about to make our last one. Let's add some social icons to the page's footer. For grins and giggles, let's make this change on a new footer branch that's based off the `master` branch. So we need to create a new `footer` branch, first.

💡 Switch and Create Branch In One Command 💡

The way we currently work with branches is to create a branch with the `git branch` command and then switch to that newly created branch with the `git checkout` command.

But did you know that the `git checkout` command can actually create a new branch, too? If you provide the `-b` flag, you can create a branch *and* switch to it all in one command.

```
$ git checkout -b richards-branch-for-awesome-changes
```

It's a pretty useful command, and I use it often.

Let's use this new feature of the `git checkout` command to create our new `footer` branch and have this footer branch start at the same location as the master branch:

```
$ git checkout -b footer master
```

Now if we run a quick `git log --oneline --decorate`, we should see (your commit messages might be different):

Add Social Links

Now that we're on a new branch, let's add some social links to the page's footer. I've added the following content:

```
<footer>
    <!-- start of new content -->
    <section>
        <h3 class="visuallyhidden">Social Links</h3>
        <a class="social-link" href="https://twitter.com/udacity">
            <img src="img/social-twitter.png" alt="Twitter">
        </a>
        <a class="social-link" href="https://www.instagram.com/udacity/">
            <img src="img/social-instagram.png" alt="Instagram">
        </a>
        <a class="social-link" href="https://plus.google.com/+Udacity">
            <img src="img/social-google.png" alt="Google Plus">
        </a>
    </section>
    <!-- end of new content -->
</footer>
```

See All Branches At Once

We've made it to the end of all the changes we needed to make! Awesome job!

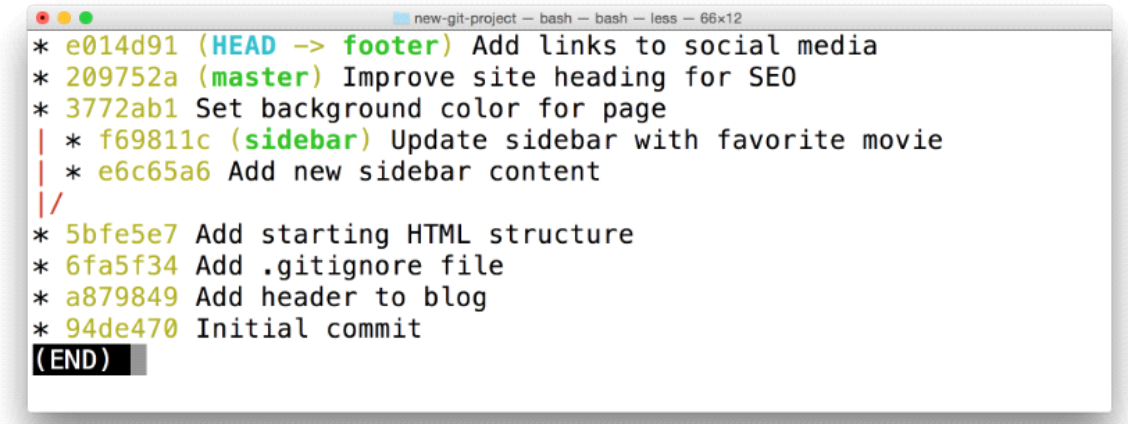
Now we have multiple sets of changes on three different branches. We can't see other branches in the `git log` output unless we switch to a branch. Wouldn't it be nice if we could see *all* branches at once in the `git log` output.

As you've hopefully learned by now, the `git log` command is pretty powerful and *can* show us this information. We'll use the new `--graph` and `--all` flags:

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

The `--graph` flag adds the bullets and lines to the leftmost part of the output. This shows the actual *branching* that's happening. The `--all` flag is what displays *all* of the branches in the repository.

Running this command will show all branches and commits in the repository:



Recap Of Changes

We've made the following changes:

1. on the master branch, we added a default color to the page

- we created a sidebar branch and added code for a sidebar
- on the master branch, we changed the heading of the page
- on the sidebar branch, we added more content to the sidebar
- we created a footer branch and added social links to the footer

These changes are all on their own, separate branches. Let's have Git combine these changes together. Combining branches together is called **merging**.

Merging

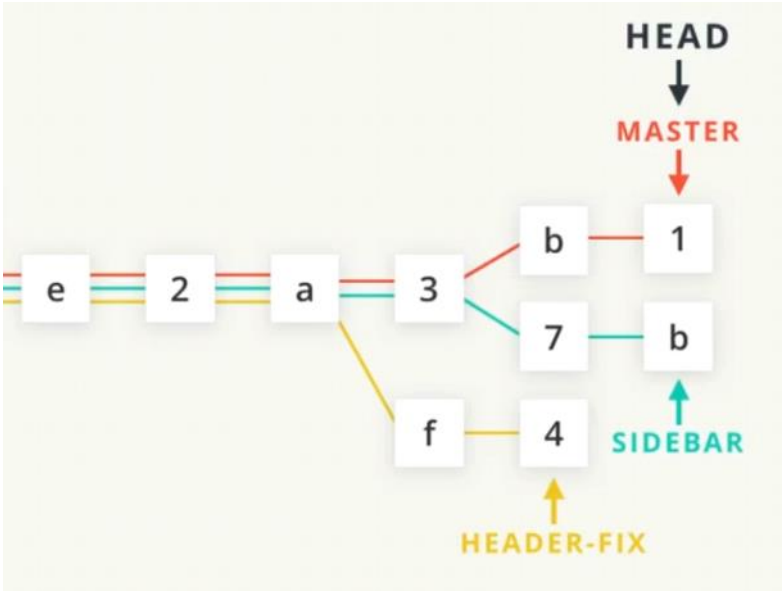
Remember that the purpose of a topic branch (like `sidebar`) is that it lets you make changes that do not affect the `master` branch. Once you make changes on the topic branch, you can either decide that you don't like the changes on the branch and you can just delete that branch, or you can decide that you want to keep the changes on the topic branch and combine those changes in with those on another branch.

Combining branches together is called **merging**.

Git can automatically merge the changes on different branches together. This branching and merging ability is what makes Git *incredibly powerful*! You can make small or extensive changes on branches, and then just use Git to combine those changes together.

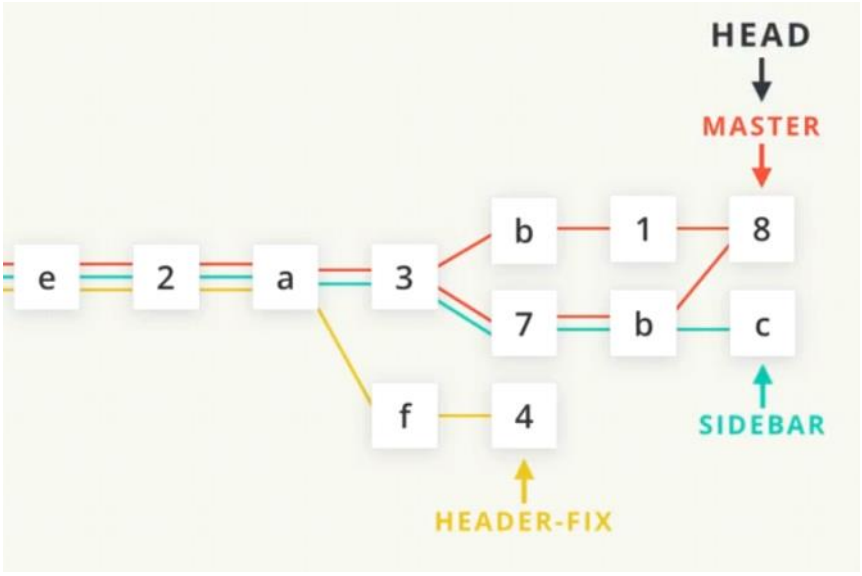
Let's see how this works, in theory. Pay attention to the two main types of merges in Git, a regular **merge** and a **Fast-forward merge**.

Let's explore how merging works in Git using our imaginary project with different branches.



Merging master and sidebar Branches

1. Suppose we want to merge the **master** and **sidebar** branches.
2. When a merge occurs, it creates a **merge commit**.
3. Since **HEAD** is pointing to the **master** branch, the merge commit will be placed on the **master** branch, and the branch pointer will move forward.
4. Let's merge **sidebar** into **master**. Notice that the merge commit links to two different earlier commits.
5. This merge does not affect the **sidebar** branch. We can switch back to it and continue making commits without any issues.

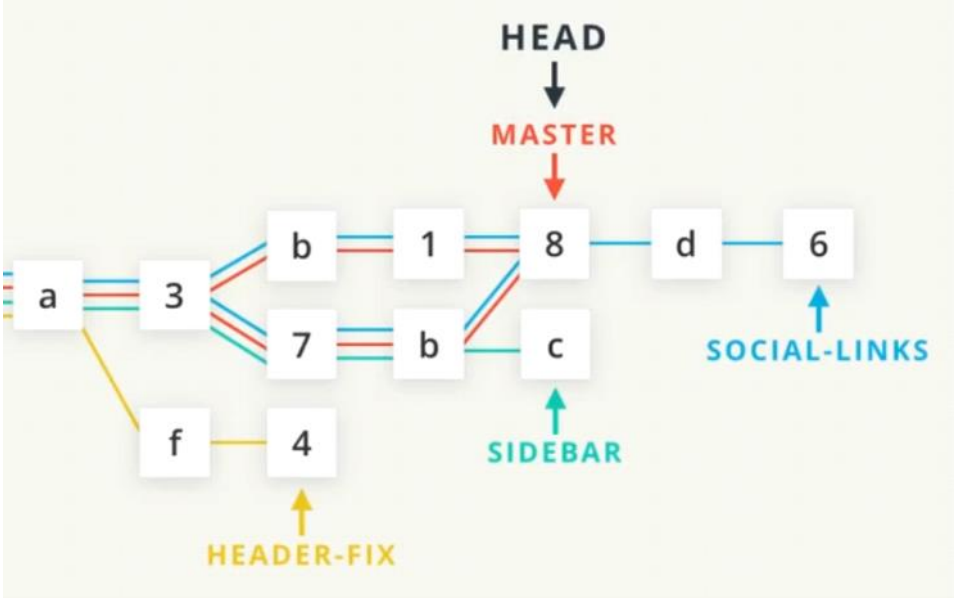


Merging social-links into master

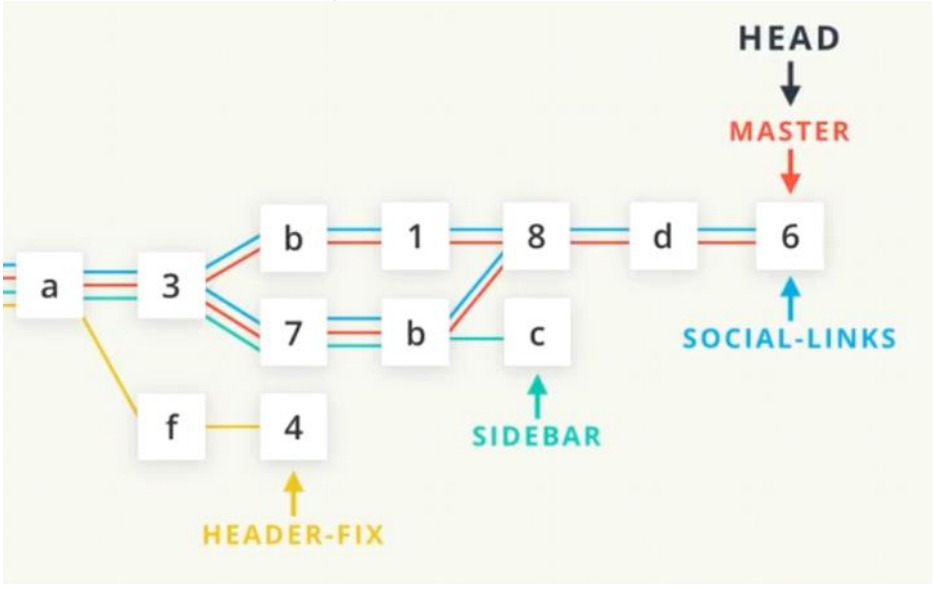
- Imagine we have a **social-links** branch that is a few commits ahead of

the **master** branch.

- The **master** branch does not yet include the commits from the **social-links** branch.
- To include those commits in **master**, we would merge the **social-links** branch into **master**.



- Since **HEAD** is pointing to **master**, the **master** branch will move forward after the merge. Because the **social-links** branch is directly ahead of **master**, Git will perform a **fast-forward merge**.



The colored lines in the diagram represent connections between commits. The order of these lines does not matter. For example, moving the red line to the top does not change anything—they simply show how commits are linked.

⚠ Know The Branch ⚠

It's very important to know which branch you're on when you're about to merge branches together. Remember that making a merge makes a commit.

As of right now, we do not know how to *undo* changes. We'll go over it in the next lesson, but if you make a merge on the wrong branch, use this command to undo the merge:

```
git reset --hard HEAD^
```

(Make sure to include the `^` character! It's a known as a "Relative Commit Reference" and indicates "the parent commit". We'll look at Relative Commit References in the next lesson.)

The Merge Command

The `git merge` command is used to combine Git branches:

```
$ 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

Fast-forward Merge

In our project, I've checked out the `master` branch and I want *it* to have the changes that are on the `footer` branch. If I wanted to verbalize this, I could say this is - "I want to merge in the `footer` branch". That "merge in" is important; when a merge is performed, the *other* branch's changes are brought into the branch that's currently checked out.

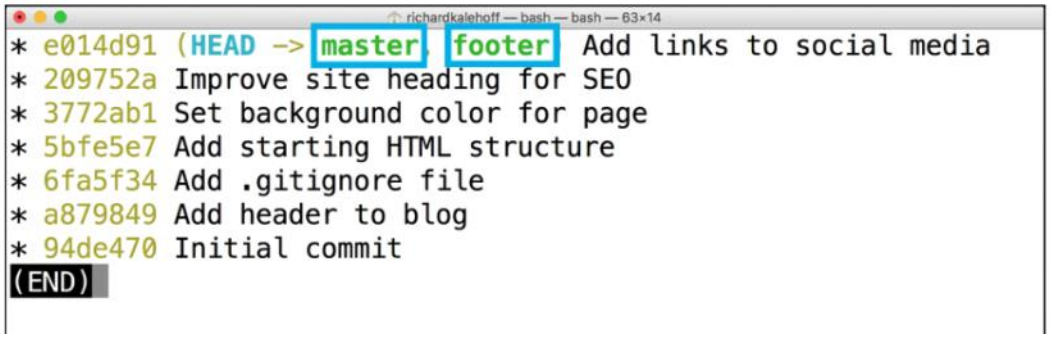
Let me stress that again - When we merge, we're merging some other branch into the current (checked-out) branch. We're not merging two branches into a new branch. We're not merging the current branch into the other branch.

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.

To merge in the `footer` branch, run:

```
$ git merge footer
```

This is what my Terminal displays after running the command:



Perform A Regular Merge

Fantastic work doing a Fast-forward merge! That wasn't too hard, was it?

But you might say - "Of course that was easy, all of the commits are already there and the branch pointer just moved forward!"...and you'd be right. It's the simplest of merges.

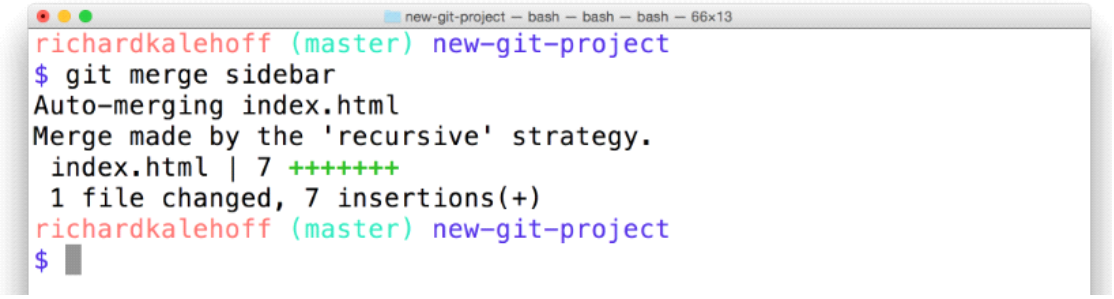
So let's do the more common kind of merge where two *divergent* branches are combined. You'll be surprised that to merge in a divergent branch like `sidebar` is actually no different!

To merge in the `sidebar` branch, make sure you're on the `master` branch and run:

```
$ git merge sidebar
```

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.

> This is what my Terminal shows after using the default commit message:



Aaaand that's all there is to merging! It's pretty simple, isn't it? You might read a lot of material that makes branching and merging seem overly complicated, but it's really not too bad at all.

How about a quick quiz to see if you're following along with how merging works.

Let's say a repository has 4 branches in it:

- **master**
- **allisons-mobile-footer-fix**
- **nav-updates**
- **jonathans-seo-changes**

The changes on `master` and `allisons-mobile-footer-fix` need to be merged together. If HEAD points to `allisons-mobile-footer-fix`, which branch will move when the merge is performed?

- ○ master
- ◉ allisons-mobile-footer-fix ✓
- ○ nav-updates
- ○ jonathans-seo-changes

What If A Merge Fails?

The merges we just did were able to merge successfully. Git is able to intelligently combine lots of work on different branches. However, there are times when it can't combine branches together. When a merge is performed and fails, that is called a **merge conflict**. We'll look at merge conflicts, what causes them, and how to resolve them in the next lesson.

Merge Recap

To recap, the `git merge` command is used to combine branches in Git:

```
$ git merge <other-branch>
```

There are two types of merges:

- Fast-forward merge – the branch being merged in must be *ahead* of the checked out branch. The checked out branch's pointer will just be moved forward to point to the same commit as the other branch.
- the regular type of merge
  - two divergent branches are combined
  - a merge commit is created

Merge Conflicts

Sometimes Merges Fail

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**.

If a merge conflict does occur, Git will try to combine as much as it can, but then it will leave special markers (e.g. `>>>` and `<<<`) that tell you where you (yep, you the programmer!) needs to manually fix.

What Causes A Merge Conflict

As you've learned, Git tracks *lines* in files. A merge conflict will happen when *the exact same line(s)* are changed in separate branches. For example, if you're on a `alternate-sidebar-style` branch and change the sidebar's heading to "About Me" but then on a different branch and change the sidebar's heading to "Information About Me", which heading should Git choose? You've changed the heading on both branches, so there's no way Git will know which one you actually want to keep. And it sure isn't going to just randomly pick for you!

Let's force a merge conflict so we can learn to resolve it. Trust me, it's simple once you get the hang of it! Remember that a merge conflict occurs when Git isn't sure which line(s) you want to use from the branches that are being merged. So we need to edit *the same line on two different branches*...and then try to merge them.

Forcing A Merge Conflict!

Remember that a merge conflict occurs when *the exact same line(s) are changed in separate branches*. Let's alter the page's heading on two different branches. So Let's:

- change the heading on the `master` branch
- create a `heading-update` branch *that's located on the commit right before the recently modified* `master` *branch*
- change the *same* heading
- switch back to the `master` branch
- merge in the `heading-update` branch

Change Heading On Branch 1

Since the `master` branch is just a regular ol' branch like all the rest, let's just alter the heading while we're on the `master` branch. So change the `<h1>` heading from whatever you have it to something else. For me, the heading is currently "Adventure" on line 13, and I'm changing it to "Quest".

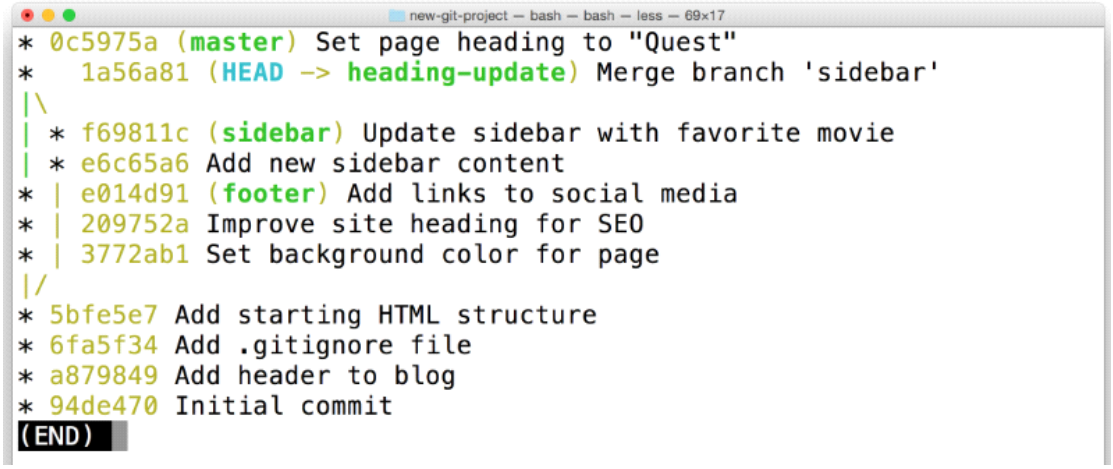
Once you've made the change, save the file and commit it to the repository.

Change Heading On Branch 2

Now we need to create a different branch and update the heading on that branch.

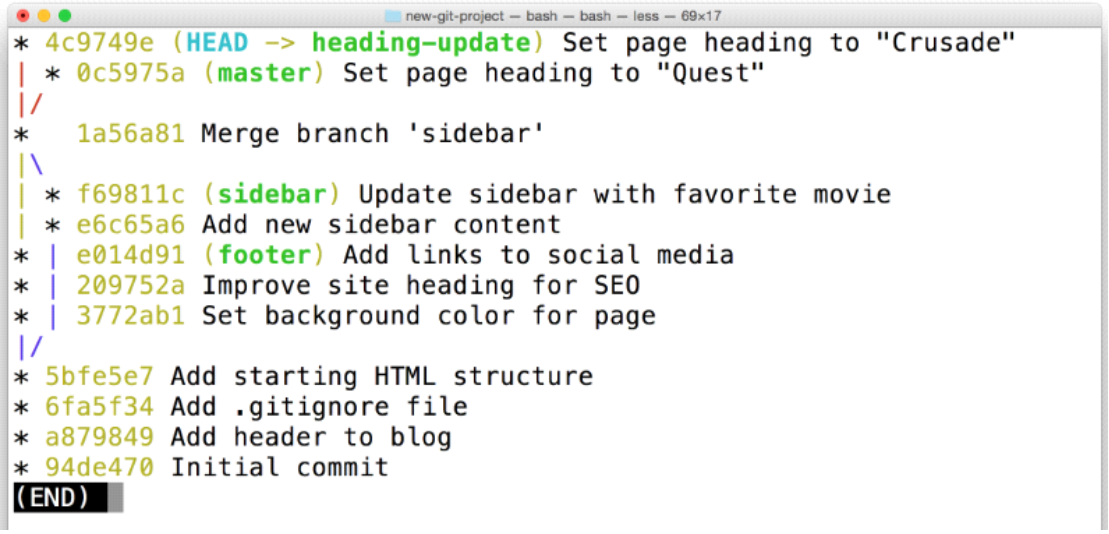
Now this is important, we need to create a branch that's *not branching from the master branch*. If we make a change that branches *off* of the master branch, then *that* change will be "ahead" of this one and Git will just use that change instead of the one we just made on `master`. So we need to put the branch "in the past".

Let's just create a branch that's on the commit right *before* the most recent one. So use Git log to get the previous commit's SHA and create a branch on that commit. Here's what my Git log looks like after creating a `heading-update` branch:



Ok, now that we've got our branch created, we just need to update the heading, again. Now make sure you change *the exact same line* that you changed on the `master` branch. I'm changing "Adventure" on line 13 to "Crusade".

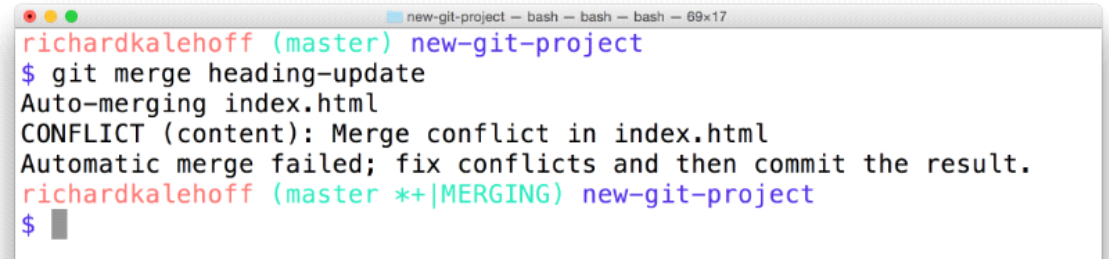
Then save the file and commit it to the repository.



Make sure you're on the master branch (we could really do this on either branch, but I just have a habit of keeping the `master` branch the main one that other topic branches get merged into) and merge in the `heading-update` branch:

```
$ git merge heading-update
```

You should see the following:



Merge Conflict Output Explained

The output that shows in the Terminal is:

```
$ git merge heading-update
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

Notice that right after the `git merge heading-update` command, it tries merging the file that was changed on both branches (`index.html`), but that there was a conflict. Also, notice that it tells you what happened - "Automatic merge failed; fix conflicts and then commit the result".

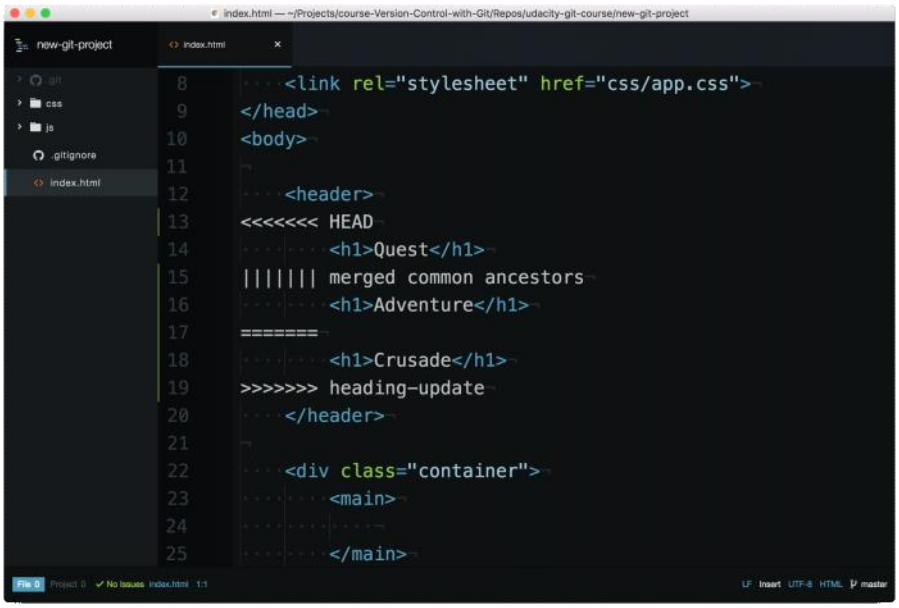
Remember our good friend `git status`? Well he'll come in really handy when working with merge conflicts.

Question 1 of 2

Try running `git status` right now. Which of the following information does it show?

- [x] You have unmerged paths
- [x] Unmerged paths
- [x] fix conflicts and run "git commit"
- [x] use "git add ..." to mark resolution

The `git status` output tells us to that the merge conflict is inside `index.html`. So check out that file in your code editor!



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)

Resolving A Merge Conflict

Git is using the merge conflict indicators to show you what lines caused the merge conflict on the two different branches as well as what the original line used to have. So to resolve a merge conflict, you need to:

1. choose which line(s) to keep

- remove all lines with indicators

For some reason, I'm not happy with the word "Crusade" right now, but "Quest" isn't all that exciting either. How about "Adventurous Quest" as a heading?!?

Commit Merge Conflict

Once you've removed all lines with merge conflict indicators and have selected what heading you want to use, just save the file, add it to the staging index, and commit it! Just like with a regular merge, this will pop open your code editor for you to supply a commit message. Just like before, it's common to use the provided merge commit message, so after the editor opens, just close it to use the provided commit message.

And that's it! Merge conflicts really aren't all that challenging once you understand what the merge conflict indicators are showing you.

Question 2 of 2

You've made numerous commits so far in your exploration of Git. If a merge conflict occurs in a file and you edit the file, save it, stage it, and commit it but *forget* to remove the merge conflict indicators, will Git commit the file?

- (●) Yes ✓
- ( ) No

That's right! Git will commit the lines with the merge conflict indicators! They're just regular characters, so there's no reason Git will stop the commit because of them. It's up to you to actually remove them. Don't forget to use `git diff` to check what's going to be staged/committed!

Merge Conflict Recap

A merge conflict happens when the same line or lines have been changed on different branches that are being merged. Git will pause mid-merge telling you that there is a conflict and will tell you in what file or files the conflict occurred. To resolve the conflict in a file:

- locate and remove all lines with merge conflict indicators
- determine what to keep
- save the file(s)
- stage the file(s)
- make a commit

Be careful that a file might have merge conflicts in multiple parts of the file, so make sure you check the entire file for merge conflict indicators - a quick search for `<<<` should help you locate all of them.

Undoing Changes

Making commits to a repository can be nerve-wracking. What if I make a mistake?

What if I want to undo something I did in the past? Or what if I want to completely erase the last few commits because I'm certain they will never be used? That's exactly what we're going to explore. In this lesson, we'll look at git commit --amend, git revert, and git reset.

- With **git commit --amend**, you can alter the most recent commit. This is useful if you forgot to include a file in the commit or if there was a typo in the commit message.
- With **git revert**, you provide the SHA of the commit you want to revert. The changes made in that commit are reversed. For example, lines that were added will be deleted. Don't worry—we'll examine this in detail.
- With **git reset**, you can actually delete commits. However, you can't delete just any commit; you must delete commits in order. Be cautious with this command, as it removes items from the repository permanently.

With these powerful commands, you'll have the Git skills to tackle anything your job throws at you. Let's start by exploring the git commit --amend flag.

Modifying The Last Commit

Changing 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.

```
$ git commit --amend
```

If your Working Directory is clean (meaning there aren't any uncommitted changes in the repository), then running `git commit --amend` will let you provide a new commit message. Your code editor will open up and display the original commit message. Just fix a misspelling or completely reword it! Then save it and close the editor to lock in the new commit message.

Add Forgotten 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`

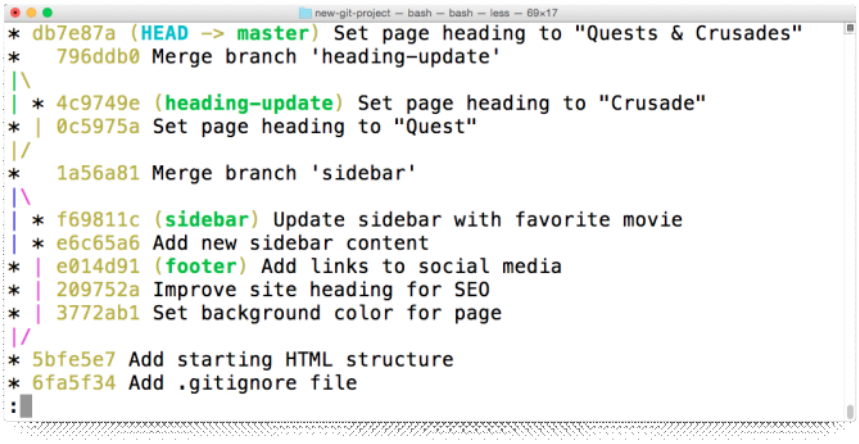
So you'd make changes to the necessary CSS and/or HTML files to get the forgotten link styled correctly, then you'd save all of the files that were modified, then you'd use `git add` to stage all of the modified files (just as if you were going to make a new commit!), but then you'd run `git commit --amend`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

Reverting A Commit

What Is A Revert?

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!

We ended the previous lesson with a merge conflict and resolved that conflict by setting the heading to `Adventurous Quest`. Let's say that there's a commit in your repository that changes the heading now to `Quests & Crusades`.



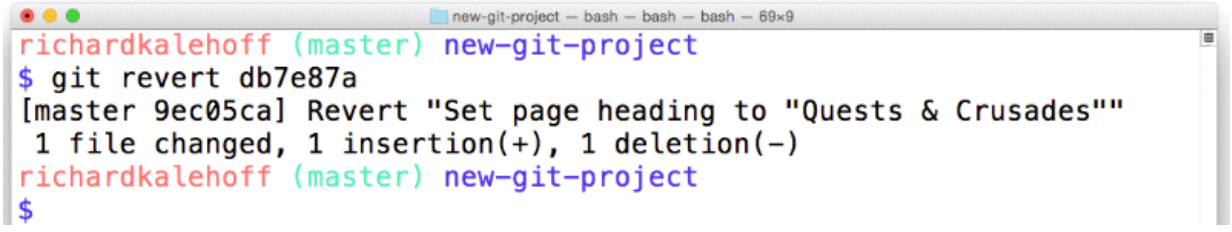
The `git revert` Command

Now that I've made a commit with some changes, I can revert it with the `git revert` command

```
$ git revert <SHA-of-commit-to-revert>
```

Since the SHA of the most-recent commit is `db7e87a`, to revert it: I'll just run `git revert db7e87a` (this will pop open my code editor to edit/accept the provided commit message)

I'll get the following output:



Did you see how the output of the `git revert` command tells us what it reverted? It uses the commit message of the commit that I told it to revert. Something that's also important is that it creates *a new commit*.

Revert Recap

To recap, the `git revert` command is used to reverse a previously made commit:

```
$ 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

Reset vs Revert

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!

⚠ Resetting Is Dangerous ⚠

You've got to be careful with Git's resetting capabilities. This is one of the few commands that lets you erase commits from the repository. If a commit is no longer in the repository, then its content is gone.

To alleviate the stress a bit, Git *does* keep track of everything for about 30 days before it completely erases anything. To access this content, you'll need to use the `git reflog` command. Check out these links for more info:

- git-reflog
- Rewriting History
- reflog, your safety net

Relative Commit References

You already know that you can reference commits by their SHA, by tags, branches, and the special `HEAD` pointer. Sometimes that's not enough, though. There will be times when you'll want to reference a commit relative to another commit. For example, there will be times where you'll want to tell Git about the commit that's one before the current commit...or two before the current commit. There are special characters called "Ancestry References" that we can use to tell Git about these relative references. Those characters are:

- `^` – indicates the parent commit
- `~` – indicates the *first* parent commit

Here's how we can refer to previous commits:

- the parent commit – the following indicate the parent commit of the current commit
  - HEAD^
  - HEAD~
  - HEAD~1
- the grandparent commit – the following indicate the grandparent commit of the current commit
  - HEAD^^
  - HEAD~2
- the great-grandparent commit – the following indicate the great-grandparent commit of the current commit
  - HEAD^^^
  - HEAD~3

The main difference between the `^` and the `~` is when a commit is created *from a merge*. A merge commit has *two* parents. With a merge commit, the `^` reference is used to indicate the *first* parent of the commit while `^2` indicates the *second* parent. The first parent is the branch you were on when you ran `git merge` while the second parent is the branch that was merged in.

It's easier if we look at an example. This what my `git log` currently shows:

```
* 9ec05ca (HEAD -> master) Revert "Set page heading to "Quests & Crusades""
* db7e87a Set page heading to "Quests & Crusades"
*   796ddb0 Merge branch 'heading-update'
|\
| * 4c9749e (heading-update) Set page heading to "Crusade"
* | 0c5975a Set page heading to "Quest"
|/
*   1a56a81 Merge branch 'sidebar'
|\
| * f69811c (sidebar) Update sidebar with favorite movie
| * e6c65a6 Add new sidebar content
* | e014d91 (footer) Add links to social media
* | 209752a Improve site heading for SEO
* | 3772ab1 Set background color for page
|/
* 5bfe5e7 Add starting HTML structure
* 6fa5f34 Add .gitignore file
* a879849 Add header to blog
* 94de470 Initial commit
```

Let's look at how we'd refer to some of the previous commits. Since `HEAD` points to the `9ec05ca` commit:

- `HEAD^` is the `db7e87a` commit
- `HEAD~1` is also the `db7e87a` commit
- `HEAD^^` is the `796ddb0` commit
- `HEAD~2` is also the `796ddb0` commit
- `HEAD^^^` is the `0c5975a` commit
- `HEAD~3` is also the `0c5975a` commit
- `HEAD^^^2` is the `4c9749e` commit (this is the grandparent's (`HEAD^^`) *second parent* (`^2`))

Which Commit?

Use this repository to answer the following quiz questions:

```
* 9ec05ca (HEAD -> master) Revert "Set page heading to "Quests & Crusades""
* db7e87a Set page heading to "Quests & Crusades"
*   796ddb0 Merge branch 'heading-update'
|\
| * 4c9749e (heading-update) Set page heading to "Crusade"
* | 0c5975a Set page heading to "Quest"
|/
*   1a56a81 Merge branch 'sidebar'
|\
| * f69811c (sidebar) Update sidebar with favorite movie
| * e6c65a6 Add new sidebar content
* | e014d91 (footer) Add links to social media
* | 209752a Improve site heading for SEO
* | 3772ab1 Set background color for page
|/
* 5bfe5e7 Add starting HTML structure
* 6fa5f34 Add .gitignore file
* a879849 Add header to blog
* 94de470 Initial commit
```

Question 1 of 3

Which commit is referenced by `HEAD~6`?

- ○ 4c9749e
- ○ 0c5975a
- ○ 1a56a81
- ○ f69811c
- ○ e014d91
- ◉ 209752a ✓

You did so well on that last one, why not give this one a go! Using the same repository, which commit is referenced by `HEAD~4^2`?

f69811c

The `git reset` Command

The `git reset` command is used to reset (erase) commits:

```
$ 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

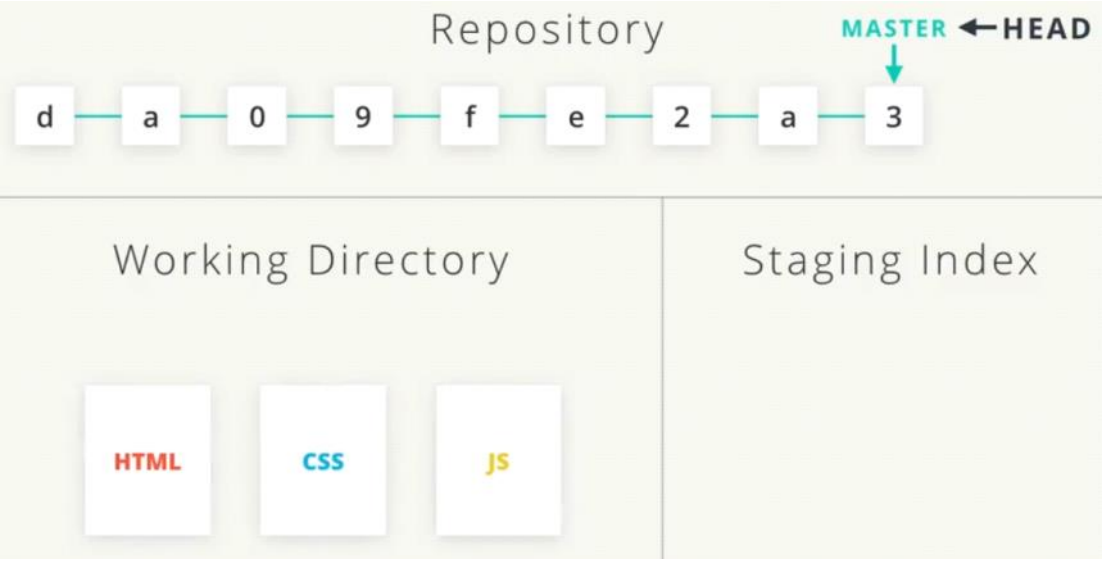
Git Reset's Flags

The way that Git determines if it erases, stages previously committed changes, or unstages previously committed changes is by the flag that's used. The flags are:

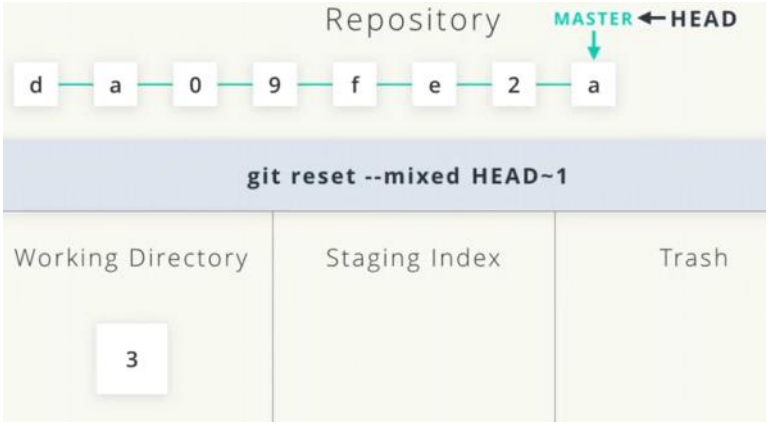
- `--mixed`
- `--soft`
- `--hard`

It's easier to understand how they work with a little animation.

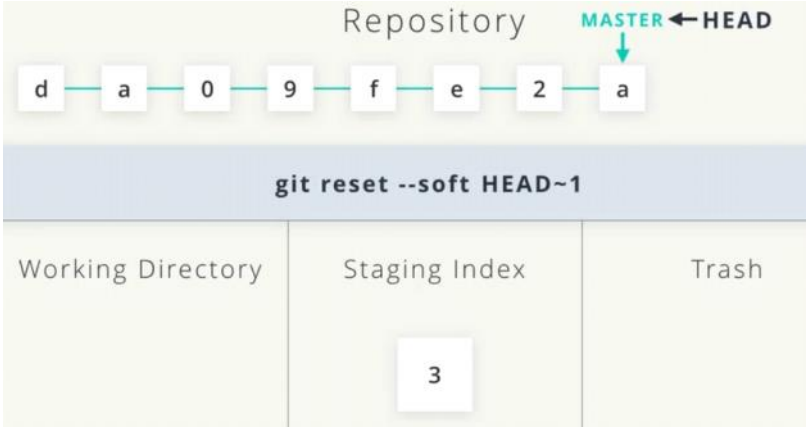
We've got here the three sections of our repository: the working directory, the staging index, and the repository up at the top. Let's say our repo currently has these commits in it, with **master** currently pointing at the most recent commit. We have **HEAD** pointing at **master**, and our files are in the working directory.



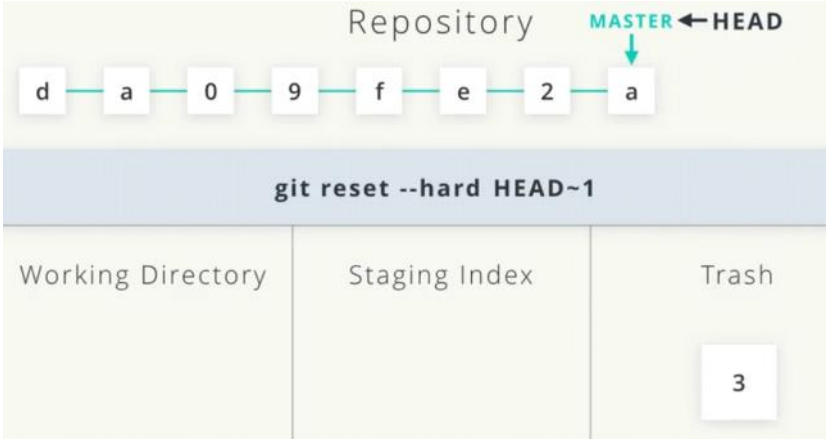
Let's shift things around for a second. These are the three places content from a reset commit will go: the working directory, the staging index, and the trash, meaning it'll be erased.
Running git reset HEAD's parent will move HEAD and master to the previous commit. But what happens to the commit with the number three? The changes that were made could be moved to the working directory or the staging index, or they could just be thrown out. It all depends on the flag that's used with the reset command.



Let's move this commit back. If we run git reset without any flags, then the changes that were made in the commit with SHA-3 are applied to the files in the working directory. This is because --mixed is the default. So running git reset without a flag is the same as running it with --mixed. If we stage the files and commit again, we'll get the same commit content. We'll get a different commit SHA just because the timestamp of the commit will be different from the original one, but the commit content will be exactly the same.



Using the --soft flag will move the changes that were made in the commit with SHA-3 to the staging index. It's the same changes, and now they're even staged for you. All you have to do is run git commit to get the commit back. Again, since the timestamp is different, the new commit SHA will be different.
The last is the --hard flag. This one will throw out all of the changes that were made in the commit with SHA-3.



💡 Backup Branch 💡

Remember that using the `git reset` command will *erase* commits from the current branch. So if you want to follow along with all the resetting stuff that's coming up, you'll need to create a branch on the current commit that you can use as a backup.

Before I do any resetting, I usually create a `backup` branch on the most-recent commit so that I can get back to the commits if I make a mistake:

```
$ git branch backup
```

Reset's `--mixed` Flag

Let's look at each one of these flags.

```
* 9ec05ca (HEAD -> master) Revert "Set page heading to "Quests & Crusades""
* db7e87a Set page heading to "Quests & Crusades"
* 796ddb0 Merge branch 'heading-update'
```

Using the sample repo above with `HEAD` pointing to `master` on commit `9ec05ca`, running `git reset --mixed HEAD^` will take the changes made in commit `9ec05ca` and move them to the working directory.

💡 Back To Normal 💡

If you created the `backup` branch prior to resetting anything, then you can easily get back to having the `master` branch point to the same commit as the `backup` branch. You'll just need to:

1. remove the uncommitted changes from the working directory

- merge `backup` into `master` (which will cause a Fast-forward merge and move `master` up to the same point as `backup`)

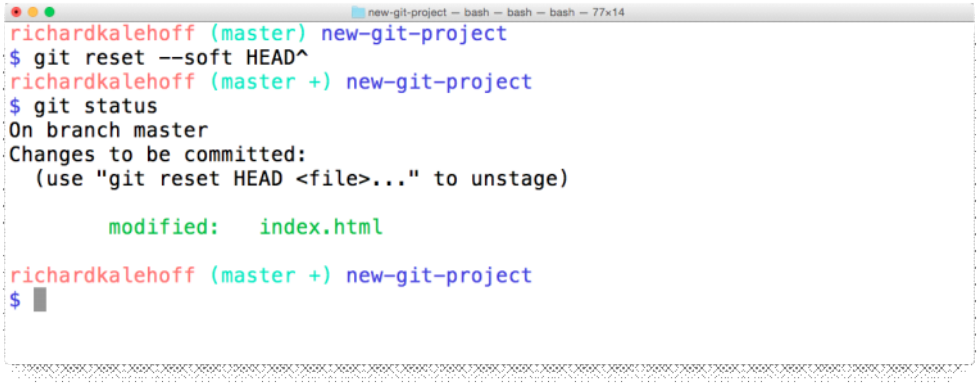
```
$ git checkout -- index.html
$ git merge backup
```

Reset's `--soft` Flag

Let's use the same few commits and look at how the `--soft` flag works:

```
* 9ec05ca (HEAD -> master) Revert "Set page heading to "Quests & Crusades""
* db7e87a Set page heading to "Quests & Crusades"
* 796ddb0 Merge branch 'heading-update'
```

Running `git reset --soft HEAD^` will take the changes made in commit `9ec05ca` and move them directly to the Staging Index.

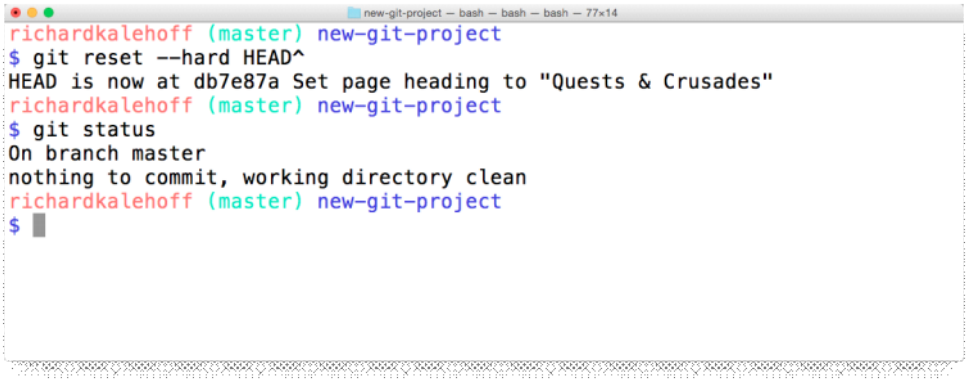


Reset's `--hard` Flag

Last but not least, let's look at the `--hard` flag:

```
* 9ec05ca (HEAD -> master) Revert "Set page heading to "Quests & Crusades""
* db7e87a Set page heading to "Quests & Crusades"
* 796ddb0 Merge branch 'heading-update'
```

Running `git reset --hard HEAD^` will take the changes made in commit `9ec05ca` and erases them.



Now it's your turn!

Refer to the following repository:

```
* e014d91 (HEAD -> master, footer) Add links to social media
* 209752a Improve site heading for SEO
* 3772ab1 Set background color for page
* 5bfe5e7 Add starting HTML structure
* 6fa5f34 Add .gitignore file
* a879849 Add header to blog
* 94de470 Initial commit
```

Question 2 of 3

What will happen to the changes from the `3772ab1` commit if `git reset --hard HEAD~3` is run? Will the changes be in the Staging Index, in the Working Directory, or complete erased?

- ◯ Staging Index
- ◯ Working Directory
- ◉ erased ✓

Question 3 of 3

What will happen to the changes from the `209752a` commit if `git reset --soft HEAD^^` is run? Will the changes be in the Staging Index, in the Working Directory, or complete erased?

- ◉ Staging Index ✓
- ◯ Working Directory
- ◯ erased

Reset Recap

To recap, the `git reset` command is used erase commits:

```
$ git reset <reference-to-commit>
```

It can be used to:

- move the HEAD and current branch pointer to the referenced commit
- erase commits with the `--hard` flag
- moves committed changes to the staging index with the `--soft` flag
- unstages committed changes `--mixed` flag

Typically, ancestry references are used to indicate previous commits. The ancestry references are:

- `^` – indicates the parent commit
- `~` – indicates the first parent commit

Further Research

- git-reset(opens in a new tab) from Git docs
- Reset Demystified(opens in a new tab) from Git Blog
- Ancestry References(opens in a new tab) from Git Book

We explored the git commit --amend flag, which allows us to reword or update the most recent commit. We also reviewed the git revert command, which is used to undo a specific commit, and the git reset command, which resets or erases changes. However, remember to use git reset with caution. Knowing how to fix mistakes has definitely boosted my confidence to experiment, and I hope you feel more confident making changes now that you know how to undo them!

Expand On Your Git Skills

- take the companion GitHub course
- create a repo to track your computer's settings - https://dotfiles.github.io/(opens in a new tab)
- develop the next, awesome feature for your personal project
- try tackling some Git challenges with the Git-it app

Version Control in Data Science

To familiarize yourself with how you may use version control at work, we'll first take

a look at an example scenario where you are data scientists using Git at work. We'll practice the commands to use in these type of scenarios, and then we'll go over some of the unique challenges and techniques for versioning data science work.

Scenario #1 (Branches)

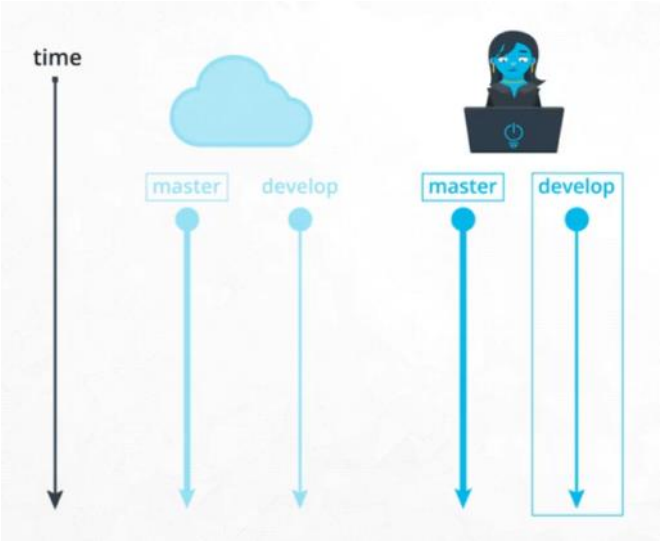
Imagine you're part of a data science team that's responsible for building a company's recommendation engine. You're sitting at your desk working on a feature that incorporates demographic data like age, gender, and relationship status to improve recommendations to users. You're midway through this implementation when your boss comes over to notify you that the business intelligence team predicts that a user's friend groups will produce the best recommendations in the short-term and wanted to prioritize this immediately.

Looking at the recommendation engine code you have on your screen, you can see there's a bunch of unfinished non-working code that you've added while working on the demographic feature. Running this code right now would break.

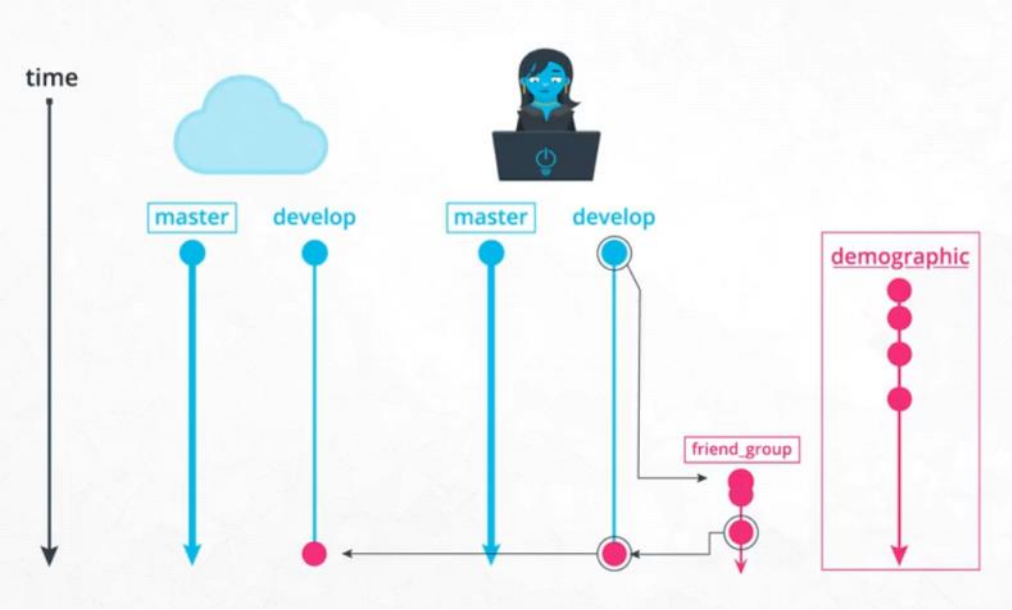
If you want to get started on this new friend groups feature now, you'd have to undo all of this work maybe by editing back lines you've modified and committing out lines of code you added.

This would be a very messy and risky route, or say you're using a version control system like **Git**.
You can simply commit your changes and create another branch for this new feature.
Here's what I mean.

Your company has a Git repository for its recommendation engine and has **a master branch, which holds the code used in production** and the **develop branch which holds the latest stable version of code with changes for the next release**. You have a local version of this repository on your laptop, and to get the latest stable version you pull from the develop branch.



When you start working on this demographic feature, you create a new branch for this called "Demographic" and start working on your code in this branch.
However, in the middle of your work, you need to work on another feature. So, you commit your changes on this demographic branch and switch back to the develop branch.
From the stable develop branch, you create another branch for a new feature called friend groups. After you finish your work on the friend groups branch, you commit your changes, switch back to development branch, merge it with the friend groups branch, and push this to the remote repository's develop branch.



Now, you can switch back to the demographic branch to continue your progress on that feature.
As you can see here, proper use of Git commits and branches can help you work on multiple features at once and switch between them with ease.

Scenario #1

Let's walk through the Git commands that go along with each step in the scenario you just observed in the video.

Step 1: You have a local version of this repository on your laptop, and to get the latest stable version, you pull from the develop branch.

Switch to the develop branch

```
git checkout develop
```

Pull the latest changes in the develop branch

```
git pull
```

Step 2: When you start working on this demographic feature, you create a new branch called demographic, and start working on your code in this branch.

Create and switch to a new branch called demographic from the develop branch

```
git checkout -b demographic
```

Work on this new feature and commit as you go

```
git commit -m 'added gender recommendations'
git commit -m 'added location specific recommendations'
...
```

Step 3: However, in the middle of your work, you need to work on another feature. So you commit your changes on this demographic branch, and switch back to the develop branch.

Commit your changes before switching

```
git commit -m 'refactored demographic gender and location
recommendations '
```

Switch to the develop branch

```
git checkout develop
```

Step 4: From this stable develop branch, you create another branch for a new feature called friend_groups.

Create and switch to a new branch called friend_groups from the develop branch

```
git checkout -b friend_groups
```

Step 5: After you finish your work on the friend_groups branch, you commit your changes, switch back to the development branch, merge it back to the develop branch, and push this to the remote repository's develop branch.

Commit your changes before switching

```
git commit -m 'finalized friend_groups recommendations '
```

Switch to the develop branch

```
git checkout develop
```

Merge the friend_groups branch into the develop branch

```
git merge --no-ff friends_groups
```

Push to the remote repository

```
git push origin develop
```

Step 6: Now, you can switch back to the demographic branch to continue your progress on that feature.

Switch to the demographic branch

```
git checkout demographic
```

Scenario #2 (Commit Messages)

After working on this recommendation feature for a while, you've created a model to produce recommendations based on friend groups.
The model is scored pretty well on your validation set, but you remember it did even better just a few hours ago.
You've been experimenting with tweaks to this model, and you're not sure what combination of parameters and tweaks were in place when your model scored the highest.
Luckily, you've been including a message with each commit noting what you did as well as the training and cross-validation scores for each commit.

Instead of spending a bunch of time trying to retrace your steps from memory or testing a bunch of tweaks again to try and get the same score, you can check your commit history seeing messages of the changes you need and how well it performed.
The model at this commit seem to be scoring the highest, so you decide to take a look.

After inspecting your code, you've realized what modifications made this perform well and use those for your model. Now, you're pretty confident merging this back to the develop branch and pushing the updated recommendation engine.

Scenario #2

Let's walk through the Git commands that go along with each step in the scenario you just observed in the video.

Step 1: You check your commit history, seeing messages about the changes you made and how well the code performed.

View the log history

```
git log
```

Step 2: The model at this commit seemed to score the highest, so you decide to take a look.

Check out a commit

```
git checkout bc90f2cbc9dc4e802b46e7a153aa106dc9a88560
```

After inspecting your code, you realize what modifications made it perform well, and use those for your model.

Step 3: Now, you're confident merging your changes back into the development branch and pushing the updated recommendation engine.

Switch to the develop branch

```
git checkout develop
```

Merge the friend_groups branch into the develop branch

```
git merge --no-ff friend_groups
```

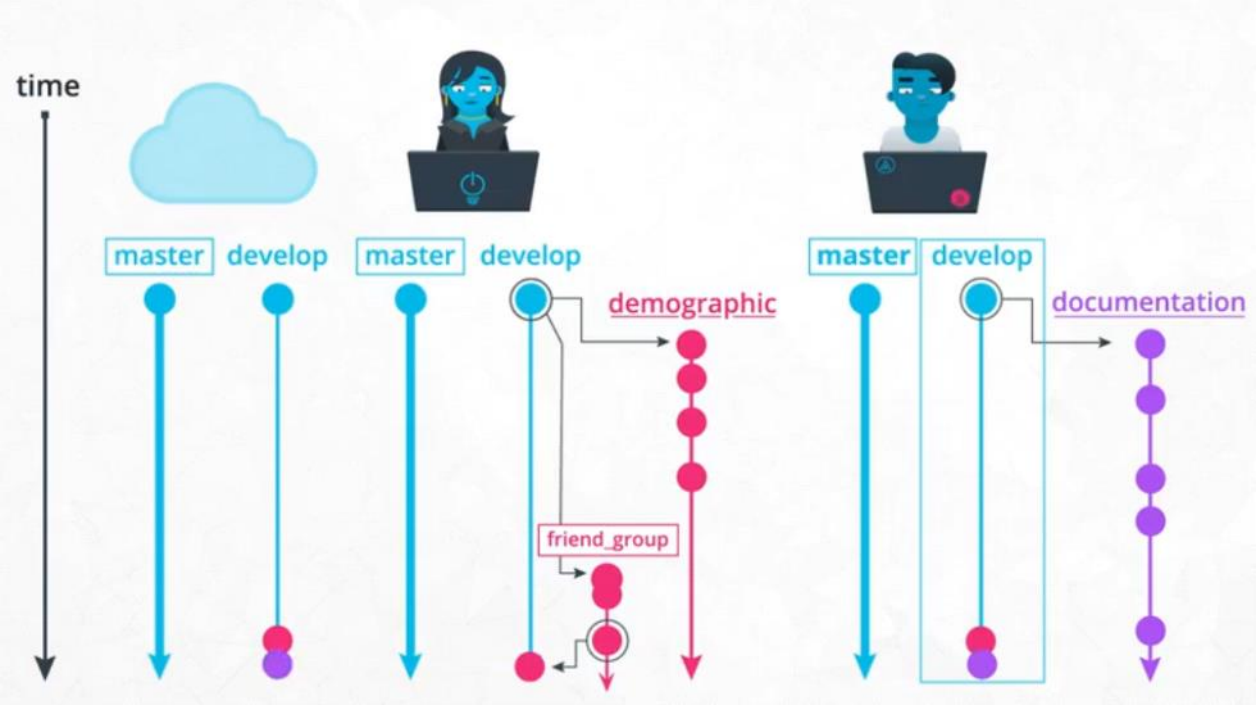
Push your changes to the remote repository

```
git push origin develop
```

Scenario #3

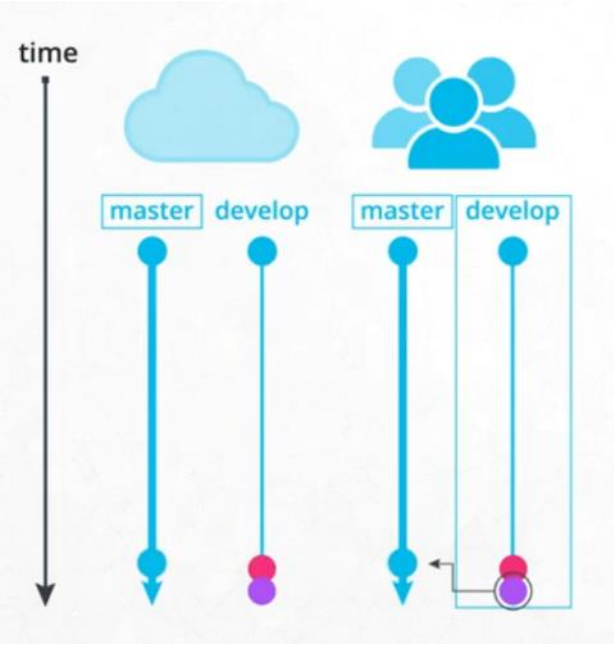
While you were working on these changes, your co-worker, Andrew, has been working on improvements to the documentation of the same recommendation engine on a different branch called Documentation.

Andrew commits his changes to the documentation branch, switches to the development branch and pulls down the latest changes from the remote repository on this branch, which includes the change I previously merged for the friend groups feature.



Then, Andrew merges his documentation branch to the develop branch on his local repository, and then pushes his changes up to update the develop branch on the remote repository.

After the team reviewed both of your work, they merged update from the develop branch to the master branch. Now, they pushed the changes to the master branch on the remote repository.
These changes are now in production.



Using branches, commit messages, and merging demonstrate just a few ways version control can be used in data science on a team, but the process will vary based on your team

Scenario #3

Let's walk through the Git commands that go along with each step in the scenario you just observed in the video.

Step 1: Andrew commits his changes to the documentation branch, switches to the development branch, and pulls down the latest changes from the cloud on this development branch, including the change I merged previously for the friends group feature.

Commit the changes on the documentation branch

```
git commit -m "standardized all docstrings in process.py"
```

Switch to the develop branch

```
git checkout develop
```

Pull the latest changes on the develop branch down

```
git pull
```

Step 2: Andrew merges his documentation branch into the develop branch on his local repository, and then pushes his changes up to update the develop branch on the remote repository.

Merge the documentation branch into the develop branch

```
git merge --no-ff documentation
```

Push the changes up to the remote repository

```
git push origin develop
```

Step 3: After the team reviews your work and Andrew's work, they merge the updates from the development branch into the master branch. Then, they push the changes to the master branch on the remote repository. These changes are now in production.

Merge the develop branch into the master branch

```
git merge --no-ff develop
```

Push the changes up to the remote repository

```
git push origin master
```

Resources

Read this great article(opens in a new tab) on a successful Git branching strategy.

Note on merge conflicts

For the most part, Git makes merging changes between branches really simple. However, there are some cases where Git can become confused about how to combine two changes, and asks you for help. This is called a merge conflict.
Mostly commonly, this happens when two branches modify the same file.
For example, in this situation, let's say you deleted a line that Andrew modified on his branch. Git wouldn't know whether to delete the line or modify it. You need to tell Git which change to take, and some tools even allow you to edit the change manually. If it isn't straightforward, you may have to consult with the developer of the other branch to handle a merge conflict.
To learn more about merge conflicts and methods to handle them, see About merge conflicts(opens in a new tab).

Model Versioning

In the previous example, you may have noticed that each commit was documented with a score for that model. This is one simple way to help you keep track of model versions. Version control in data science can be tricky, because there are many pieces involved that can be hard to track, such as large amounts of data, model versions, seeds, and hyperparameters.

The following resources offer useful methods and tools for managing model versions and large amounts of data. These are here for you to explore, but are not necessary to know now as you start your journey as a data scientist. On the job, you'll always be learning new skills, and many of them will be specific to the processes set in your company.

- How to version control your production machine learning models(opens in a new tab)
- Version Control ML Model