

About Version Control

This chapter will be about getting started with Git. We will begin by explaining some background on version control tools, then move on to how to get Git running on your system and finally how to get it set up to start working with. At the end of this chapter you should understand why Git is around, why you should use it and you should be all set up to do so.

About Version Control

What is “version control”, and why should you care? Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later. For the examples in this book, you will use software source code as the files being version controlled, though in reality you can do this with nearly any type of file on a computer.

If you are a graphic or web designer and want to keep every version of an image or layout (which you would most certainly want to), a Version Control System (VCS) is a very wise thing to use. It allows you to revert selected files back to a previous state, revert the entire project back to a previous state, compare changes over time, see who last modified something that might be causing a problem, who introduced an issue and when, and more. Using a VCS also generally means that if you screw things up or lose files, you can easily recover. In addition, you get all this for very little overhead.

Local Version Control Systems

Many people’s version-control method of choice is to copy files into another directory (perhaps a time-stamped directory, if they’re clever). This approach is very common because it is so simple, but it is also incredibly error prone. It is easy to forget which directory you’re in and accidentally write to the wrong file or copy over files you don’t mean to.

To deal with this issue, programmers long ago developed local VCSs that had a simple database that kept all the changes to files under revision control.

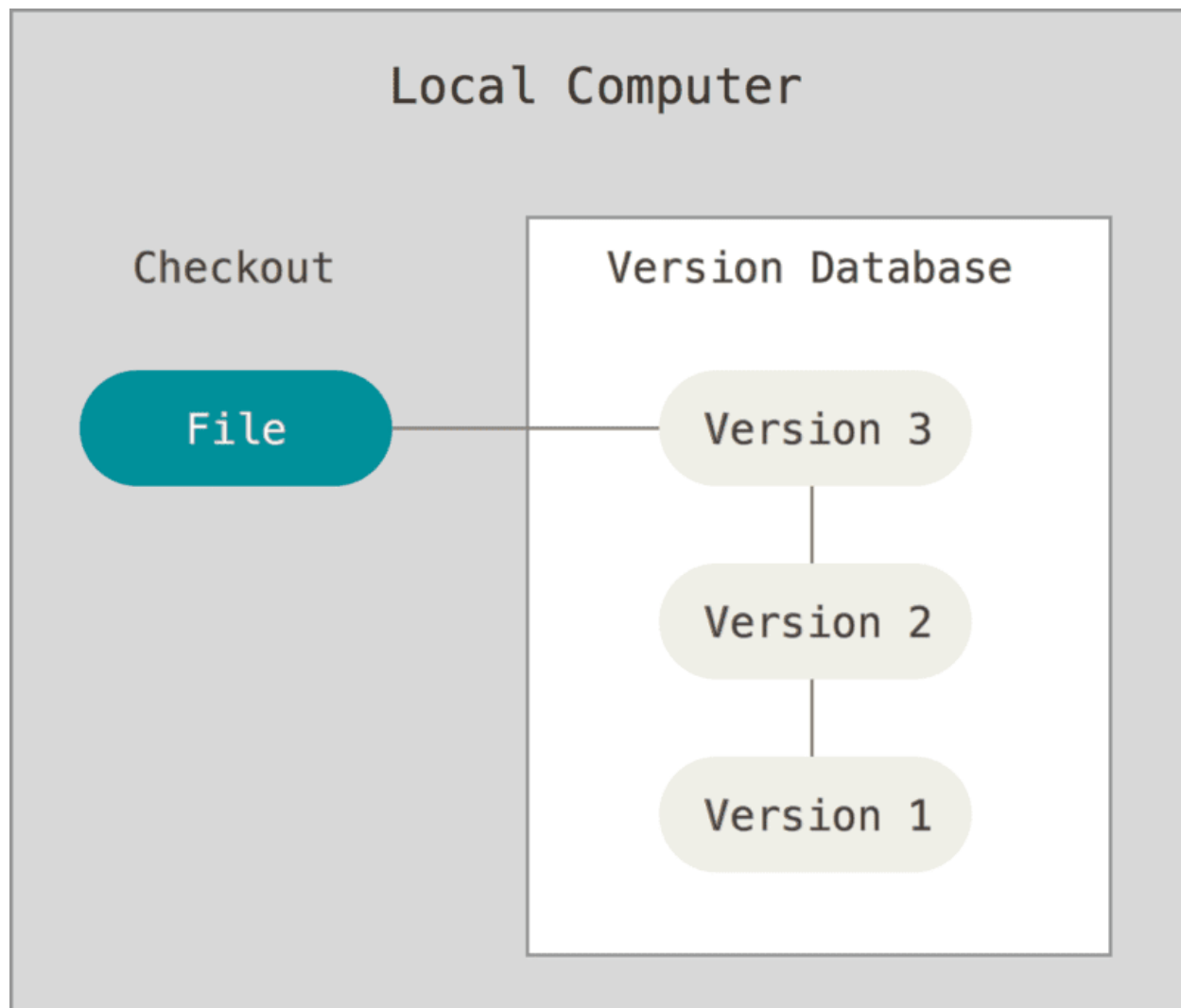


Figure 1. Local version control.

One of the most popular VCS tools was a system called RCS, which is still distributed with many computers today. [RCS](#) works by keeping patch sets (that is, the differences between files) in a special format on disk; it can then re-create what any file looked like at any point in time by adding up all the patches.

Centralized Version Control Systems

The next major issue that people encounter is that they need to collaborate with developers on other systems. To deal with this problem, Centralized Version Control Systems (CVCSs) were developed. These systems (such as CVS, Subversion, and Perforce) have a single server that contains all the versioned files, and a number of clients that check out files from that central place. For many years, this has been the standard for version control.

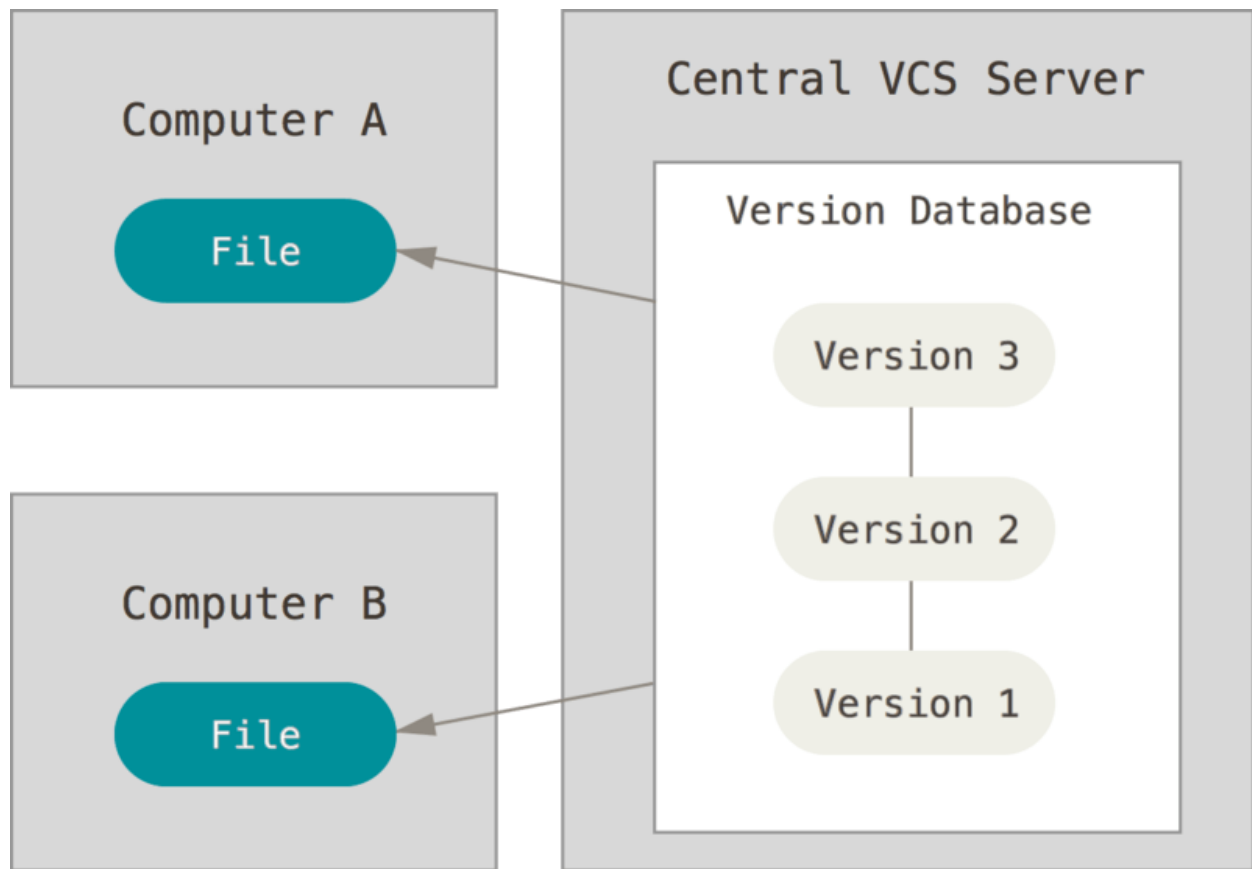


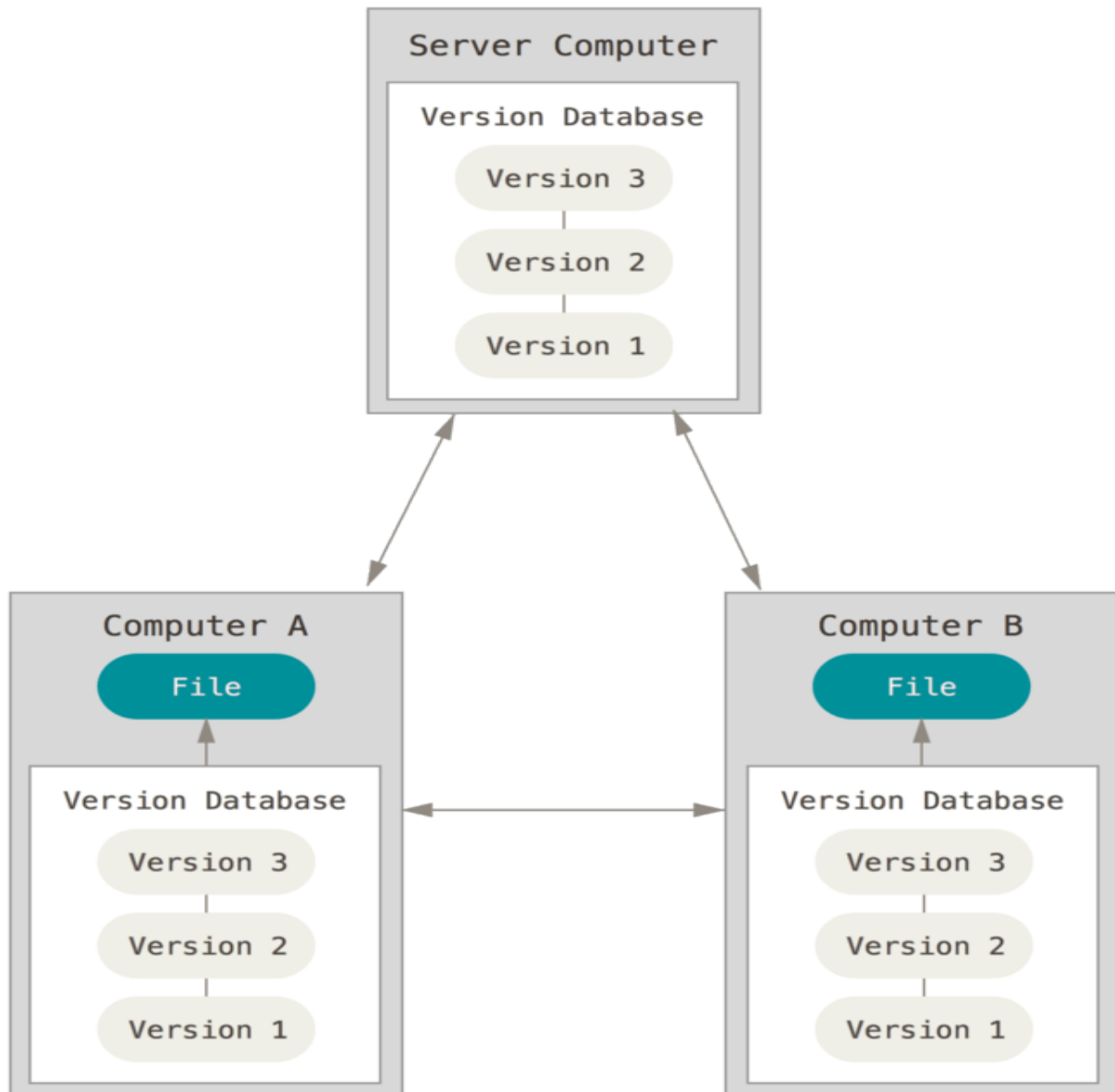
Figure 2. Centralized version control.

This setup offers many advantages, especially over local VCSs. For example, everyone knows to a certain degree what everyone else on the project is doing. Administrators have fine-grained control over who can do what, and it's far easier to administer a CVCS than it is to deal with local databases on every client.

However, this setup also has some serious downsides. The most obvious is the single point of failure that the centralized server represents. If that server goes down for an hour, then during that hour nobody can collaborate at all or save versioned changes to anything they're working on. If the hard disk the central database is on becomes corrupted, and proper backups haven't been kept, you lose absolutely everything — the entire history of the project except whatever single snapshots people happen to have on their local machines. Local VCS systems suffer from this same problem — whenever you have the entire history of the project in a single place, you risk losing everything.

Distributed Version Control Systems

This is where Distributed Version Control Systems (DVCSs) step in. In a DVCS (such as Git, Mercurial, Bazaar or Darcs), clients don't just check out the latest snapshot of the files; rather, they fully mirror the repository, including its full history. Thus, if any server dies, and these systems were collaborating via that server, any of the client repositories can be copied back up to the server to restore it. Every clone is really a full backup of all the data.



Furthermore, many of these systems deal pretty well with having several remote repositories they can work with, so you can collaborate with different groups of people in different ways simultaneously within the same project. This allows you to set up several types of workflows that aren't possible in centralized systems, such as hierarchical models.

A Short History of Git

A Short History of Git

As with many great things in life, Git began with a bit of creative destruction and fiery controversy.

The Linux kernel is an open source software project of fairly large scope. For most of the lifetime of the Linux kernel maintenance (1991–2002), changes to the software were passed around as patches and archived files. In 2002, the Linux kernel project began using a proprietary DVCS called BitKeeper.

In 2005, the relationship between the community that developed the Linux kernel and the commercial company that developed BitKeeper broke down, and the tool's free-of-charge status was revoked. This prompted the Linux development community (and in particular Linus Torvalds, the creator of Linux) to develop their own tool based on some of the lessons they learned while using BitKeeper. Some of the goals of the new system were as follows:

- Speed
- Simple design
- Strong support for non-linear development (thousands of parallel branches)
- Fully distributed
- Able to handle large projects like the Linux kernel efficiently (speed and data size)

Since its birth in 2005, Git has evolved and matured to be easy to use and yet retain these initial qualities. It's amazingly fast, it's very efficient with large projects, and it has an incredible branching system for non-linear development (See [Git Branching](#)).

Installing Git

Installing Git

Before you start using Git, you have to make it available on your computer. Even if it's already installed, it's probably a good idea to update to the latest version. You can either install it as a package or via another installer, or download the source code and compile it yourself.

Note

This book was written using Git version **2.8.0**. Though most of the commands we use should work even in ancient versions of Git, some of them might not or might act slightly differently if you're using an older version. Since Git is quite excellent at preserving backwards compatibility, any version after 2.0 should work just fine.

Installing on Linux

If you want to install the basic Git tools on Linux via a binary installer, you can generally do so through the package management tool that comes with your distribution. If you're on Fedora (or any closely-related RPM-based distribution, such as RHEL or CentOS), you can use `dnf`:

```
$ sudo dnf install git-all
```

If you're on a Debian-based distribution, such as Ubuntu, try `apt`:

```
$ sudo apt install git-all
```

For more options, there are instructions for installing on several different Unix distributions on the Git website, at <https://git-scm.com/download/linux>.

Installing on macOS

There are several ways to install Git on a Mac. The easiest is probably to install the Xcode Command Line Tools. On Mavericks (10.9) or above you can do this simply by trying to run **git** from the Terminal the very first time.

```
$ git --version
```

If you don't have it installed already, it will prompt you to install it.

If you want a more up to date version, you can also install it via a binary installer. A macOS Git installer is maintained and available for download at the Git website, at <https://git-scm.com/download/mac>.



Figure 7. Git macOS Installer.

You can also install it as part of the GitHub for macOS install. Their GUI Git tool has an option to install command line tools as well. You can download that tool from the GitHub for macOS website, at <https://desktop.github.com>.

Installing on Windows

There are also a few ways to install Git on Windows. The most official build is available for download on the Git website. Just go to <https://git-scm.com/download/win> and the download will start automatically. Note that this is a project called Git for Windows, which is separate from Git itself; for more information on it, go to <https://gitforwindows.org>.

To get an automated installation you can use the [Git Chocolatey package](#). Note that the Chocolatey package is community maintained.

Another easy way to get Git installed is by installing GitHub Desktop. The installer includes a command line version of Git as well as the GUI. It also works well with PowerShell, and sets up solid credential caching and sane CRLF settings. We'll learn more about those things a little later, but suffice it to say they're things you want. You can download this from the [GitHub Desktop website](#).

Installing from Source

Some people may instead find it useful to install Git from source, because you'll get the most recent version. The binary installers tend to be a bit behind, though as Git has matured in recent years, this has made less of a difference.

If you do want to install Git from source, you need to have the following libraries that Git depends on: autotools, curl, zlib, openssl, expat, and libiconv. For example, if you're on a system that has `dnf` (such as Fedora) or `apt-get` (such as a Debian-based system), you can use one of these commands to install the minimal dependencies for compiling and installing the Git binaries:

```
$ sudo dnf install dh-autoreconf curl-devel expat-devel gettext-devel \
openssl-devel perl-devel zlib-devel
```

```
$ sudo apt-get install dh-autoreconf libcurl4-gnutls-dev libexpat1-dev \
gettext libz-dev libssl-dev
```


In order to be able to add the documentation in various formats (doc, html, info), these additional dependencies are required:

```
$ sudo dnf install asciidoc xmlto docbook2X
```

```
$ sudo apt-get install asciidoc xmlto docbook2x
```

First-Time Git Setup

First-Time Git Setup

Now that you have Git on your system, you'll want to do a few things to customize your Git environment. You should have to do these things only once on any given computer; they'll stick around between upgrades. You can also change them at any time by running through the commands again.

Git comes with a tool called `git config` that lets you get and set configuration variables that control all aspects of how Git looks and operates. These variables can be stored in three different places:

1. `/etc/gitconfig` file: Contains values applied to every user on the system and all their repositories. If you pass the option `--system` to `git config`, it reads and writes from this file specifically. (Because this is a system configuration file, you would need administrative or superuser privilege to make changes to it.)
2. `~/.gitconfig` or `~/.config/git/config` file: Values specific personally to you, the user. You can make Git read and write to this file specifically by passing the `--global` option, and this affects **all** of the repositories you work with on your system.
3. `config` file in the Git directory (that is, `.git/config`) of whatever repository you're currently using: Specific to that single repository. You can force Git to read from and write to this file with the `--local` option, but that is in fact the default. (Unsurprisingly, you need to be located somewhere in a Git repository for this option to work properly.)

Each level overrides values in the previous level, so values in `.git/config` trump those in `/etc/gitconfig`.

On Windows systems, Git looks for the `.gitconfig` file in the `$HOME` directory (`C:\Users\%USER` for most people). It also still looks for `/etc/gitconfig`, although it's relative to the MSys root, which is wherever you decide to install Git on your Windows system when you run the installer. If you are

using version 2.x or later of Git for Windows, there is also a system-level config file at `C:\Documents and Settings\All Users\Application Data\Git\config` on Windows XP, and in `C:\ProgramData\Git\config` on Windows Vista and newer. This config file can only be changed by `git config -f <file>` as an admin.

You can view all of your settings and where they are coming from using:

```
$ git config --list --show-origin
```

Your Identity

The first thing you should do when you install Git is to set your user name and email address. This is important because every Git commit uses this information, and it's immutably baked into the commits you start creating:

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

```
$ git config --global user.email johndoe@example.com
```

Again, you need to do this only once if you pass the `--global` option, because then Git will always use that information for anything you do on that system. If you want to override this with a different name or email address for specific projects, you can run the command without the `--global` option when you're in that project.

Many of the GUI tools will help you do this when you first run them.

Your Editor

Now that your identity is set up, you can configure the default text editor that will be used when Git needs you to type in a message. If not configured, Git uses your system's default editor.

If you want to use a different text editor, such as Emacs, you can do the following:

```
$ git config --global core.editor emacs
```

On a Windows system, if you want to use a different text editor, you must specify the full path to its executable file. This can be different depending on how your editor is packaged.

In the case of Notepad++, a popular programming editor, you are likely to want to use the 32-bit version, since at the time of writing the 64-bit version doesn't support all plug-ins. If you are on a

32-bit Windows system, or you have a 64-bit editor on a 64-bit system, you'll type something like this:

```
$ git config --global core.editor '"C:/Program Files/Notepad++/notepad++.exe' -multiInst -notabbar -nosession -noPlugin"
```

Note

Vim, Emacs and Notepad++ are popular text editors often used by developers on Unix-based systems like Linux and macOS or a Windows system. If you are using another editor, or a 32-bit version, please find specific instructions for how to set up your favorite editor with Git in [git config core.editor commands](#).

Warning

You may find, if you don't setup your editor like this, you get into a really confusing state when Git attempts to launch it. An example on a Windows system may include a prematurely terminated Git operation during a Git initiated edit.

Checking Your Settings

If you want to check your configuration settings, you can use the `git config --list` command to list all the settings Git can find at that point:

```
$ git config --list
```

```
user.name=John Doe
```

```
user.email=johndoe@example.com
```

```
color.status=auto
```

```
color.branch=auto
```

```
color.interactive=auto
```

```
color.diff=auto
```

```
...
```

You may see keys more than once, because Git reads the same key from different files (`/etc/gitconfig` and `~/.gitconfig`, for example). In this case, Git uses the last value for each unique key it sees.

You can also check what Git thinks a specific key's value is by typing `git config <key>`:

```
$ git config user.name
```

John Doe

Summary

Summary

You should have a basic understanding of what Git is and how it's different from any centralized version control systems you may have been using previously. You should also now have a working version of Git on your system that's set up with your personal identity. It's now time to learn some Git basics.

Git 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.

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.

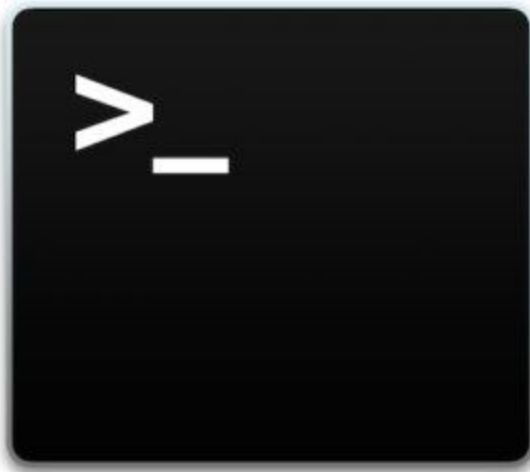
Branch

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.

Linux Shell or “Terminal”

So, basically, a shell is a program that receives commands from the user and gives it to the OS to process, and it shows the output. Linux's shell is its main part. Its distros come in GUI (graphical user interface), but basically, Linux has a CLI (command line interface). In this tutorial, we are going to cover the basic commands that we use in the shell of Linux.



To open the terminal, press Ctrl+Alt+T in Ubuntu, or press Alt+F2, type in gnome-terminal, and press enter. In Raspberry Pi, type in lxterminal. There is also a GUI way of taking it, but this is better!

Linux Commands

Basic Commands

1. pwd — When you first open the terminal, you are in the home directory of your user. To know which directory you are in, you can use the “**pwd**” command. It gives us the absolute path, which means the path that starts from the root. The root is the base of the Linux file system. It is denoted by a forward slash(/). The user directory is usually something like “/home/username”.

2. ls — Use the “**ls**” command to know what files are in the directory you are in. You can see all the hidden files by using the command “**ls -a**”.

3. cd — Use the “**cd**” command to go to a directory. For example, if you are in the home folder, and you want to go to the downloads folder, then you can type in “**cd Downloads**”. Remember, this command is case sensitive, and you have to type in the name of the folder exactly as it is. But there is a problem with these commands. Imagine you have a folder named “Raspberry Pi”. In this case, when you type in “**cd Raspberry Pi**”, the shell will take the second argument of the command as a different one, so you will get an error saying that the directory does not exist. Here, you can use a backward slash. That is, you can use “**cd Raspberry\ Pi**” in this case. Spaces are denoted like this: If you just type “**cd**” and press enter, it takes you to the home directory. To go back from a folder to the folder before that, you can type “**cd ..**”. The two dots represent back.

4. mkdir & rmdir — Use the **mkdir** command when you need to create a folder or a directory. For example, if you want to make a directory called “DIY”, then you can type “**mkdir DIY**”. Remember, as told before, if you want to create a directory named “DIY Hacking”, then you can type “**mkdir DIY\ Hacking**”. Use **rmdir** to delete a directory. But **rmdir** can only be used to delete an empty directory. To delete a directory containing files, use **rm**.

5. rm - Use the **rm** command to delete files and directories. Use "**rm -r**" to delete just the directory. It deletes both the folder and the files it contains when using only the **rm** command.

6. touch — The **touch** command is used to create a file. It can be anything, from an empty txt file to an empty zip file. For example, "**touch new.txt**".

7. cat — Use the **cat** command to display the contents of a file. It is usually used to easily view programs.

8. nano, vi, jed — **nano** and **vi** are already installed text editors in the Linux command line. The **nano** command is a good text editor that denotes keywords with color and can recognize most languages. And **vi** is simpler than **nano**. You can create a new file or modify a file using this editor. For example, if you need to make a new file named "**check.txt**", you can create it by using the command "**nano check.txt**". You can save your files after editing by using the sequence Ctrl+X, then Y (or N for no). In my experience, using **nano** for HTML editing doesn't seem as good, because of its color, so I recommend **jed** text editor. We will come to installing packages soon.

9. sudo — A widely used command in the Linux command line, **sudo** stands for "SuperUser Do". So, if you want any command to be done with administrative or root privileges, you can use the **sudo** command. For example, if you want to edit a file like **viz. alsa-base.conf**, which needs root permissions, you can use the command – **sudo nano alsa-base.conf**. You can enter the root command line using the command "**sudo bash**", then type in your user password. You can also use the command "**su**" to do this, but you need to set a root password before that. For that, you can use the command "**sudo passwd**"(not misspelled, it is **passwd**). Then type in the new root password.

10. apt-get — Use **apt** to work with packages in the Linux command line. Use **apt-get** to install packages. This requires root privileges, so use the **sudo** command with it. For example, if you want to install the text editor **jed** (as I mentioned earlier), we can type in the command "**sudo**

apt-get install jed". Similarly, any packages can be installed like this. It is good to update your repository each time you try to install a new package. You can do that by typing "**sudo apt-get update**". You can upgrade the system by typing "**sudo apt-get upgrade**". We can also upgrade the distro by typing "**sudo apt-get dist-upgrade**". The command "**apt-cache search**" is used to search for a package. If you want to search for one, you can type in "**apt-cache search jed**"(this doesn't require root).

1- 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!

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.

Git Init Recap

Use the git init command to create a new, empty repository in the current directory.

```
$ git init
```

Running this command creates a hidden .git directory. This .git directory is the brain/storage center for the repository. It holds all of the configuration files and directories and is where all of the commits are stored.

2- Git status

Status Update

At this point, we have two Git repositories:

- the empty one that we created with the git init command
- the one we cloned with the git clone command

How can we find any information about these repositories? Git's controlling them, but how can we find out what Git knows about our repos? To figure out what's going on with a repository, we use the git status command. Knowing the status of a Git repository is *extremely* important, so head straight on over to the next concept: 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.

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 ;-)

3- Git log

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.

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)

Git Log Recap

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

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

```
$ git log
```

By *default*, this command displays:

- the SHA
- the author
- the date
- and the message

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

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

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

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

Why wait?!? Click the link to move to the next lesson!

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**.

3-1 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!

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

3-2 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.

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 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?

3-3 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.

Annotated git log -p Output

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

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

git log -p 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

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.

4- 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

5- Git 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
```

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).

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.

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.

```
$ git add .
```

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.

Git Commit Recap

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

```
$ git commit [-m]
```

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!

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!

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.

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:

Globbering 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](#).

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

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

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

```
samples/*.jpg
```

Git Ignore Recap

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

6- Git tag

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"?

CAREFUL: In the command above (git tag -a v1.0) the -a flag is used. This flag tells Git to create an *annotated* tag. 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.

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
```

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.

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*

7- Git branch

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

```
$ git branch
```

It can be used to:

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

Create A Branch

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

```
$ git branch sidebar
```

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

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
```

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

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.

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

💡 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
```

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

8- Git merge

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
```


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.

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 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.

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.

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.

9- Git revert

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)

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

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

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))

10- Git reset

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 ⇒ into working directory
- --soft ⇒ into staging

- --hard ⇒ into trash

💡 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
```

💡 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
2. 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 --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.

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.

Reset Recap

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

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

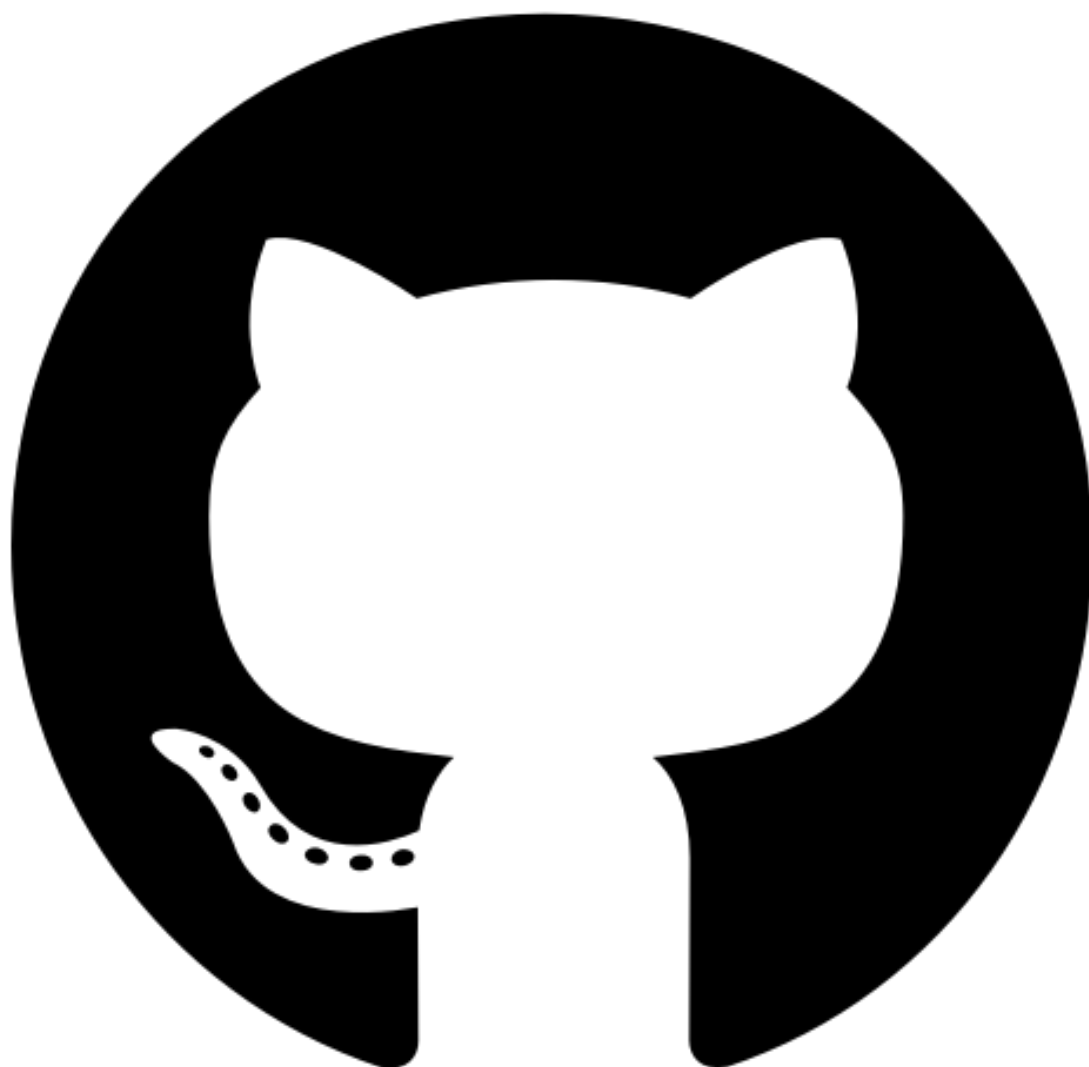
It can be used to:

- 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

Github



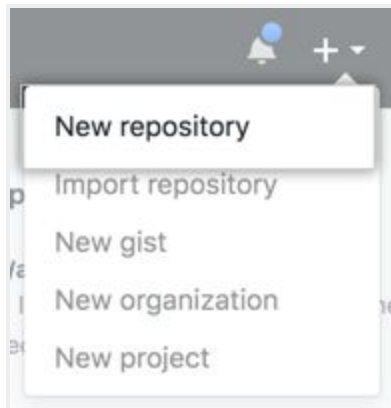
Create a repo

To put your project up on GitHub, you'll need to create a repository for it to live in.

You can store a variety of projects in GitHub repositories, including open source projects. With [open source projects](#), you can share code to make better, more reliable software.

Note: You can create public repositories for an open source project. When creating your public repository, make sure to include a [license file](#) that determines how you want your project to be shared with others. For more information on open source, specifically how to create and grow an open source project, we've created [Open Source Guides](#) that will help you foster a healthy open source community by recommending best practices for creating and maintaining repositories for your open source project. You can also take a free [GitHub Learning Lab](#) course on maintaining open source communities.

In the upper-right corner of any page, use the drop-down menu, and select New repository.




Type a short, memorable name for your repository. For example, "hello-world".

Create a new repository

A repository contains all the files for your project, including the revision history.

Owner

 octocat ▾

Repository name

hello-world ✓

Great repository names are short and memorable. Need inspiration? How about [potential-eureka](#).


Description (optional)

Optionally, add a description of your repository. For example, "My first repository on GitHub."

Create a new repository

A repository contains all the files for your project, including the revision history.

Owner

 octocat ▾

Repository name

hello-world ✓

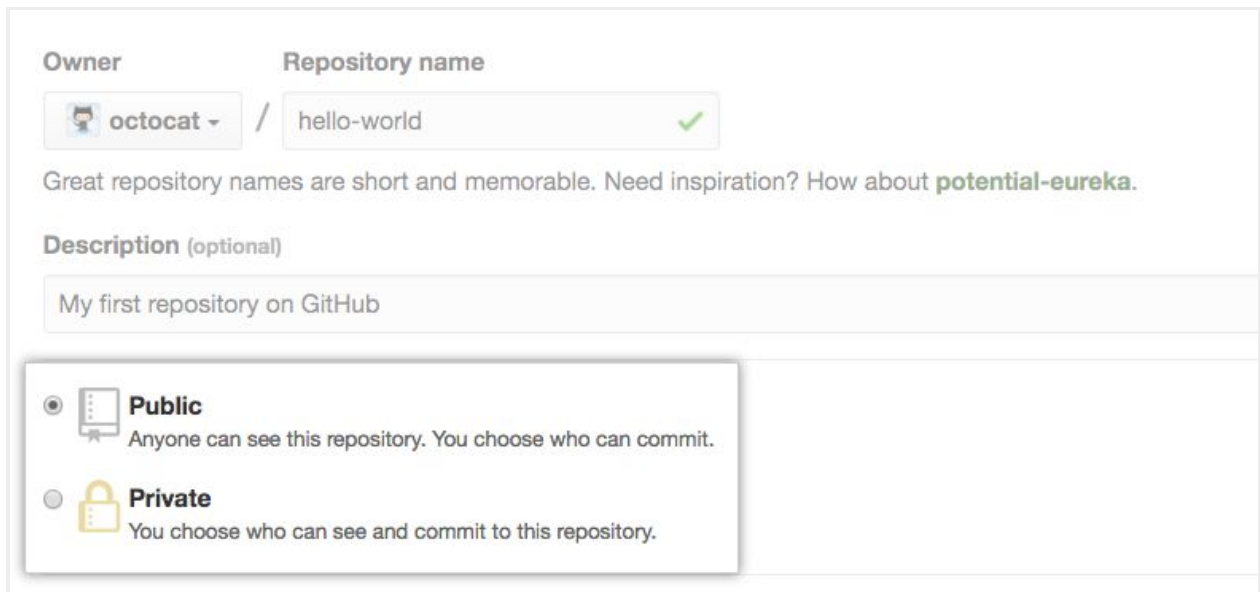
Great repository names are short and memorable. Need inspiration? How about [potential-eureka](#).

Description (optional)

My first repository on GitHub

Choose to make the repository either public or private. Public repositories are visible to the public, while private repositories are only accessible to you, and people you share them with.

For more information, see "[Setting repository visibility](#)."



The screenshot shows the GitHub repository creation interface. At the top, there are two input fields: "Owner" with a dropdown menu showing "octocat" and a "Repository name" field containing "hello-world" with a green checkmark. Below these is a text input for "Description (optional)" with the placeholder text "My first repository on GitHub". A section for repository visibility is highlighted with a grey border. It contains two options: "Public" (selected with a radio button) and "Private" (unselected). Each option has a corresponding icon (a document for public, a lock for private) and a brief description.

Owner: octocat / Repository name: hello-world ✓

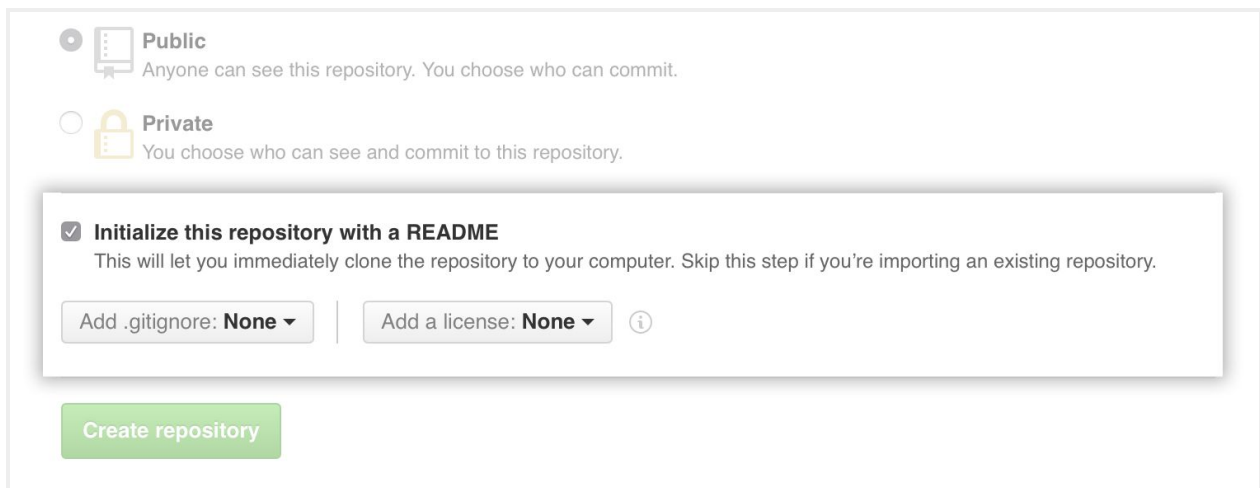
Great repository names are short and memorable. Need inspiration? How about **potential-eureka**.

Description (optional)
My first repository on GitHub

☒ **Public**
Anyone can see this repository. You choose who can commit.

☐ **Private**
You choose who can see and commit to this repository.

Select Initialize this repository with a README.



This screenshot shows the same GitHub repository creation interface as the previous one, but with the "Initialize this repository with a README" option selected. The "Public" and "Private" options are still visible. Below them, a section with a grey border contains the "Initialize this repository with a README" option, which is checked with a radio button. It includes a brief explanation and two dropdown menus: "Add .gitignore: None" and "Add a license: None", followed by an information icon. At the bottom of this section is a green "Create repository" button.

☒ **Public**
Anyone can see this repository. You choose who can commit.

☐ **Private**
You choose who can see and commit to this repository.

☒ **Initialize this repository with a README**
This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: None | Add a license: None ⓘ

Create repository

Click Create repository.

Congratulations! You've successfully created your first repository, and initialized it with a *README* file.

Commit your first change

A [commit](#) is like a snapshot of all the files in your project at a particular point in time.

When you created your new repository, you initialized it with a *README* file. *README* files are a great place to describe your project in more detail, or add some documentation such as how to install or use your project. The contents of your *README* file are automatically shown on the front page of your repository.

11- Git clone

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!

Git Clone Recap

The git clone command is used to create an identical copy of an existing repository.

```
$ git clone <path-to-repository-to-clone>
```

This command:

- takes the path to an existing repository
- by default will create a directory with the same name as the repository that's being cloned
- can be given a second argument that will be used as the name of the directory
- will create the new repository inside of the current working directory

12- Git push

Recap git push

The git push command is used to send commits from a local repository to a remote repository.

```
$ git push origin master
```

The git push command takes:

- the short name of the remote repository you want to send commits to
- the name of the branch that has the commits you want to send

13- Git pull

Pulling Changes with git pull

The local commits end at commit 5a010d1 while the remote has two extra commits - commit 4b81b2a and commit b847434.

Also, notice that in our *local* repository when we did the git log the origin/master branch is still pointing to commit 5a010d1.

Remember that the origin/master branch is not a live mapping of where the remote's master branch is located. If the remote's master moves, the local origin/master branch stays the same. To update this branch, we need to sync the two together.

git push will sync the *remote* repository with the *local* repository. To do the opposite (to sync the *local* with the *remote*), we need to use git pull. The format for git pull is very similar to git push - you provided the shortname for the remote repository and then the name of the branch you want to pull in the commits.

```
$ git pull origin master
```

Recap

If there are changes in a remote repository that you'd like to include in your local repository, then you want to *pull* in those changes. To do that with Git, you'd use the git pull command. You tell Git the shortname of the remote you want to get the changes from and then the branch that has the changes you want:

```
$ git pull origin master
```

When git pull is run, the following things happen:

- the commit(s) on the remote branch are copied to the local repository
- the local tracking branch (origin/master) is moved to point to the most recent commit
- the local tracking branch (origin/master) is merged into the local branch (master)

Also, changes can be manually added on GitHub (but this is not recommended, so don't do it).