## Thinking in Git

### What Is This Thing Called Git?

Okay. I think it's safe to assume that at some point you've heard about Git, otherwise, you probably wouldn't be here. It is by far the most widely‑used version control system in the world today. Developers use it, non‑developers use it, and basically anyone who just contributes, manages, or even just monitors projects that need to be versioned are most likely using Git, it's a foundational tool to any technology company, and nowadays, every company needs to be a technology company to innovate or just stay relevant. So whether you're just getting started using Git with a new job or new project or you've been using Git already and you want to understand it better, in this course, we'll be covering the fundamentals of Git, so you can better understand how it works, how to get started using Git and its fundamental commands and use cases, and what you need to know to get started using Git with team members in a collaborative environment. The goal of this course is not to be a complete guide to all things Git. There is a lot of information that is beyond the scope of this course. The goal is to give you a solid understanding of the fundamentals that will give you a strong foundation to Git and help set you up for success to learn more. But before we jump into the hands‑on activities and real‑world situations in this course, let's take a few minutes and cover first things first with a quick overview of what Git is, its characteristics, and how it manages changes within a project. Let's start with the definition of Git. Git is a version control system, also known as a VCS. This version control refers to the process of saving files or versions throughout the stages in life of a project where you can retrieve past versions at any time. The best metaphor that I can come up with for version control is that of a time machine. Git gives you the ability to go back in time at any point during your development and see the work at that specific point in time. To do this, Git tracks changes made to files in a project and creates a unique identifier to each change so that it's easily discoverable if you ever want to go back and retrieve a version of the project or revert a change that you've made. You can also think of this as an autosave at a specific checkpoint that you can easily reference for your project. The great thing about Git is that you can version control nearly any type of file, not just source code files. Students often use version control to keep track of the different versions of an essay or a research paper. A designer can use version control to keep versions of an image without worrying about modifying the only version of that image, and version control gives you the ability to explore new ideas within your project with the assurance that you can always revert back to a previous version no matter how far you've altered the content. Now, you've likely worked with some form of version control before, for example, Google Docs has a version history tool where you can view changes to a document at different points in time, and Microsoft Office has something similar with its Track Change feature, and you can even save multiple copies of a file, such as V1, V2, and so on, and then store them in a folder on your desktop computer. This type of version control is called local, it's simple, it's lightweight, and all of your changes are being made and stored locally on your computer, but because of this, collaborating with other developers or team members becomes a bit of a problem, and if your local copy is deleted, well, your out of luck. So to deal with these issues and help address the collaboration bottleneck, you could use a centralized version control system. Centralized version control systems are based on the idea that there is a single central copy of your project somewhere probably on a server, and team members will make their changes to this central copy. This allows for multiple clients to check out files from that centralized server at the same time, make their changes, and then send them back to the centralized server for others to see. This is a big upgrade to local version control, but it also has its flaws. The biggest problem is that everything is being stored in a centralized server, so if that server goes down, nobody can save changes, pull down files, or collaborate on anything that they're working on. Similar to local version control systems, if that central database becomes corrupted and no backups have been kept, everything is lost for that project, except for those single‑working file sets that each team member was working on at that point in time. The third type of version control is called distributed, and you probably guessed it, it's the type of version control that Git is. With this type of version control, the complete project, including all of its history and metadata, is copied to every developer's copy of the project on their own hard drive. This eliminates the sole reliance to a central server to store the project and all of its data. If for some reason a project's code based becomes corrupt or deleted, anyone working on that project can be a backup of the entire project and all of its history and metadata. With this type of version control, since every copy can act as a centralized copy of the project, the misconception is that there isn't a central project repository. Well, this is simply not true. There is nothing stopping you from saying that this particular copy of the project is the authoritative main copy. This just means that, instead of a central repository being required by the tools that you use, it is now a collaboration and social choice. These designated central copies of the project are often hosted on third‑party collaboration platforms, such as GitHub, GitLab, and BitBucket. We'll dive more into these collaboration platforms later on in this course because to collaborate with anyone else in your projects, you'll need to send or push your work to a place where others can access it. But now that we understand that Git is a version control system, let's look into its core characteristics to help explain why Git is the most popular and widely‑used version control system in the world today.

### The Characteristics of Git

Now Git has a bit of a reputation for being confusing or having a steep learning curve, not to mention the fear of running a Git command that you don't fully understand with unintended consequences. This is a great time to take a minute and learn about Git's core characteristics. In my experience as a Git trainer and daily Git user, understanding Git's characteristics and how it tracks and manages versions of your project early on in your Git learning will help reduce that steep learning curve and make learning how to use Git less confusing. So let's take a few minutes to understand these characteristics and lay the foundation for you to start thinking in Git. Git isn't just a distributed version control, it's an advanced version control system that is incredibly robust with commands and options for you to perform that give you a lot of control over how your project is being tracked and maintained unlike other version control systems. Git does not track version changes as just a list file‑based changes between one version to another by only representing the changes to a base version of a file. These version differences are commonly referred to as the deltas or diffs. Instead, Git stores each saved version of your project as a complete snapshot of the entire project's file system at that specific point in time and not just the files being modified and their deltas or diffs. This snapshot captures everything in the project, including its history and all of its metadata. You can reference old snapshots whenever you need to and new snapshots are created when your project is modified. In this example for Git, version 2 snapshot captures the change made to file A, a reference to the complete file for File B from a previous snapshot. Since File B wasn't modified in version 2, it still includes a complete reference to the entire file and all of its contents in the version 2 snapshot, and the change to File C is also included. These snapshots of your project are foundational to how Git works, and it's part of what makes Git the type of version control system that it is. This snapshot concept is really important to understanding Git, so we'll talk more about these Git snapshots in a few minutes to see how they are made. But for now, just remember that Git tracks changes as a complete snapshot or picture of the project and not just the version deltas or the diffs. Git's next characteristic is that it's optimized for local development. Now, what this means is that Git is a distributed version control system. We have a complete copy of the project and all of its history and metadata on our own local machine. So if the main version of our Git project is being hosted on let's say a remote server, which will be the case for nearly every Git project when working in a team environment, you can work disconnected from the main server since Git doesn't need a constant connection for you to work in the project. This gives you the ability to work completely offline and still be able to navigate the project and use Git. Then when you're online again, you have access to the remote server, you can push your changes up for everyone to see and also pull down any new changes that other team members have made. The third characteristic is that Git is explicit, and by explicit, I don't mean that it's full of profanity. In programming, there is the act of implicit and explicit coding. Implicit is often referred to something that's done for you by other code behind the scenes, and explicit is the approach to accomplishing the change you wish to have done by writing out clear instructions to be done, well, explicitly. So in other words, Git won't perform any action without you clearly telling it what to do. So for example, Git won't automatically sync your local copy of the project with the main copy on the remote server unless you tell it to do so. Git also won't autosave your work or complete those snapshots, unless you explicitly tell Git to do it. Every Git action requires your explicit command or instruction. And so in short, you need to be intentional with your actions. At first, this may seem like a bottleneck, and you have to explicitly tell Git to do every little thing, but once you get the hang of things, you'll find that this level of interaction gives you complete freedom to tell Git when to do things, how to do things, and where to do it. The last of these four main characteristics is that Git doesn't just support non‑linear development, it's designed to optimize it. When working on new features, resolving bugs, or just experimenting within your project, it doesn't seem great to do all of this on the same linear development path. Sure, you can always use Git to go back in time and change things or revert to a previous state, but what do you do when you want to work on two, three, or four versions of your project at the same time, Git enables you to diverge your work by creating parallel timelines giving you the freedom to explore and test out new ideas without the fear of messing up the main timeline of your project. Now, this concept of creating alternate timelines is called branching, and it's a foundational feature to how you and others work using Git. If you're familiar with other types of version control systems, you'll notice that branching isn't a new concept or unique to Git. What may be new or unique is that branching is lightweight and considered inexpensive. So when you create a new branch in Git, you are really just creating a pointer to a specific snapshot or saved version of your work in the project. This pointer marks the beginning of a new parallel timeline at that specific point in time and then allows you to try out new ideas or incorporate new features without affecting the main development timeline. Then when you're ready to incorporate those changes into the main development line of your project, you can merge your branch into the main development line and those new changes will now be incorporated. Branching is a main concept of Git, so we'll be going over this in detail later on in this course.

### The Statuses of Your Files

Now for a few minutes, I want to talk about the states your files can be in when working with Git. Once you have your Git project and you're ready to start working, it's important that you understand the different states of those files and the different statuses that those files will move into as you make and save those changes. Now, it's a bit more complicated than just making a change and clicking a save now button, but these extra steps are by design, and they give you a lot of freedom and flexibility to craft your changes exactly how you want. Naturally, this will make more sense as you learn more about Git, but understanding these different statuses at the beginning will make learning Git a lot easier. With the Git project, a file can be in one of two states, tracked or untracked. If you recall one of Git's characteristics, one thing about Git is that it is explicit, so when you add a new file to an existing Git project, Git won't automatically start tracking that file, you have to tell Git to track it. Again, this is by design and it gives you more control on how and when to perform your Git commands. Tracked files are files that were in the last Git snapshot of the project, and once you tell Git to track a file, Git will continue to track that file until you tell Git to stop tracking it, but telling Git to stop tracking files is a destructive command because you'll lose Git's metadata and tracking for that file, so it needs to be used with caution. This isn't something you'll do often or perhaps not at all, but something to be aware of for now. Once a file is being tracked, it can reside in one of three different statuses, unmodified, modified. and staged. Unmodified means that you are tracking the file, but the file doesn't have any unsaved changes from the last time it was saved in that Git snapshot. The next status is called modified. A file will move into this status when you have made changes to the file, but haven't yet saved them to the Git project. A file can remain in this status for as long as you want, and you can continue to make changes until you want to save those changes in the next snapshot, but when you're ready to save your changes by creating a snapshot, you move that file with the unsaved changes into the staged status. You can think of this status as the staging area to make sure that things look how you want before you make your snapshot. The cool thing about the staging area is that you can move different files into this status at different times and work in different files that have different statuses all while having some files remain in the staging area. And then once you're ready to save your changes and make that Git snapshot, you run a Git command to create that snapshot and commit those changes to the project. Any files that were in this staged status when you make that saved snapshot will then move back into the unmodified status. Now this concept may take a couple of times to stick, so let's run through it again to see how a file moves between each status. If we start with an untracked file, we can add the file to the project, and it will move the file into the staged status. We can then commit the file to the project by creating a Git snapshot. Now, let's say we want to make some changes to that file that is now being tracked. As we make changes to the file, it moves from the unmodified status and into the modified status. We can make as many changes as we want and the file will remain in the status. And then when we want to save our changes, we then move the file into the staged status. Again, this is the staging area where we can make sure that the changes in this status look good and what will be committed or saved in the next Git snapshot. Then we can use Git to create that snapshot of the files in the staged status, and those files then move back to the unmodified status. And if we decide we no longer want Git to track this file with version control, we can remove Git from the file, and the file status will move from unmodified status and the tracked state to the untracked state. So if these steps still seem a bit confusing, as you start to use Git and make changes to your files in your Git project, these steps happen with just a few Git commands and it soon becomes something you don't have to think much about.

### The 3 Directories of Git

Let's now use what we just learned about file states and statuses and add one more thing to it. Git has three directories that it uses to store and manage the data for your Git project. The different statuses that we just talked about are stored in one of three directories. Now, I won't be going into too much detail here since this can quickly get beyond the scope of this course and into more intermediate to advanced Git management, but knowing the different directories will help you better understand Git as we move along, and you'll see these directories being referenced throughout this course and as you continue to learn more about Git. When you first initialize Git to a project, it will create a hidden directory called .git that is stored at the root level of your project. It is in this .git directory that all of the Git metadata and tracking is stored for every saved snapshot of your project. This directory is referred to as the local history, and it is where the unmodified status of your files live. This is really just a folder that contains all of your saved snapshots of your Git project that you can look through at any time. The next directory is called the working directory, and just like it sounds, this is the directory where you perform the work within your project when you make changes to the files that are being tracked in your Git project. This is where the modified status of your files live and is one of the two directories that will always remain local to you on your machine. This is considered a workspace area, and changes that remain in this area will always be considered as still being worked on. The next directory is where the files are moved to when you want to commit or save your changes in a Git snapshot. This directory is often called the staging area, but it is often referred to as the index or cache. This directory is where the files in the staged status live waiting for you to save them and commit them in the next Git snapshot. So in short, if a file version is in the .git directory, it's committed and saved in a snapshot to the Git project. If a file is modified and has been added to the staging area, it is staged and waiting to be committed to the next snapshot. And if a file was modified since the last version of that file from the .git directory, but has not been added to the staging area, it has a modified status and lives in the working directory.

### Introduction to Common Git Commands

Now up to this point, we've looked at what Git does, its characteristics, and the basics on how it manages and stores your project's data, but we haven't yet talked about how exactly we use Git to do those actions. Remember, Git is explicit, so we need to tell Git to do something before it will do it. We do this by using specific commands and sometimes options with those commands to get even more detailed on how we exactly want Git to do something. So throughout this course, we'll be using a lot of commands to tell Git to do what we want. In total, there are over 150 different Git commands, and most, if not all of them, can take options that extend its capability further and give you a vast amount of options to use. So in short, there's often more than one way to accomplish some tasks, but most of these commands tend to be more advanced and take a good amount of time to understand before using them. About 20 or 30 of these commands are considered foundational or common commands, and those are the ones that we'll be spending time on during this course. So as an introduction to these foundational commands, let's quickly go over each one so you can get a sense of what we'll be covering, what they look like, and a brief description of what they can do. Before we go over these commands, I want to introduce an important new term. Up to this point, I described our Git project as, well, a project, and from this point on, we'll be referring to our Git project as a repository. This is the term that Git uses when describing the project, so you see that a command does something to the Git repository, well, it's describing the Git project. When it comes to Git commands, let's categorize them into the following topics, Git basics, Git branches, Git repositories, and undoing changes. The following commands will be placed into one of these categories to show the general topic that it falls into, so let's start with Git basics. The first Git command is called git init, and this command creates a Git repository and begins the tracking of a new project. Git add moves modified files into the staging area or index for the next commit snapshot. Git status shows the status of your files in the modified in staging area. Now, this is probably the most common of all Git commands, and it only provides information, it's one we'll use frequently throughout this course. Git commit is the command that makes that snapshot and records the changes to the repository. Git config sets and then reads specific Git configurations, and then git log shows the commit history logs in your repository. This is really helpful any time that you want to see the history of your saved snapshot commits in your project along with information like who committed that snapshot, the timestamp that it was committed, and the commit message description that was provided just to name a few. Git diff can be used to show you the file changes that reside in each file status, modified, unmodified, and staged. This is a really useful command if you're making a lot of changes to many files and have added some changes to the staging area, and then you want to see what file changes remain in each status before you add more changes or commit those changes from the staging area. Moving into the Git branches category, let's start with Git branch. Git branch is used to list, create, or delete branches. Git checkout lets you switch between those branches, or using an option with this command, it lets you create a new branch and check out to it or switch to it at the same time. Now, this is just an example of multiple ways to get the same result, and git merge is the command that allows you to bring changes that you've made from one branch and merge them into another branch. Now, going into the remote repositories category, we'll start with git clone. This command copies an entire repository into a new local directory. This command is used to copy the initial setup and then other Git commands are used to send and then retrieve changes from that remote repository. Next, we have git remote. This command shows tracked repositories that are linked to your local copy. Now, we'll go over this one later in the course when we talk more about working with others in a shared repository. Similar commands that interact with shared repositories include git push, this is a really popular command because it is used to send updates from your local copy of the repository to associated repositories for other collaborators to see. And the git pull command retrieves changes from a remote copy of the repository and then merges those changes from that copy of the repository into your local repository. And git fetch is similar to git pull, but it only retrieves the changes and so you can view them locally, but it doesn't yet merge them into your local copy. The last category is undoing changes. Understandably, at times, we may need to remove a commit that we've made to the repository. Maybe we've committed sensitive information or we just want to restructure our local commit history before we push those changes to the remote copy of our repository that is hosted on a cloud platform. These can be destructive commands, so make sure that you understand what you're asking it to do before you run the commit altering command. Now, we'll be covering a couple of these commands in this course, and we'll start with git revert. This command is actually a safe command because it doesn't actually alter commit history. This command takes a previous commit and then reverts all of the changes from that commit by making another commit with the inverse. This command is safe to use even when working with others with a shared history, but it does undo committed changes, so it falls into this category. The next command can be destructive and it's called git reset. This command resets file changes that you've made that reside in your staging area or working directory. Depending on the options used with this command, you can move changes around in these different file statuses to even throwing away file changes into the trash. This command is especially important to understand what options you're doing, so we'll go over each example of a git reset later on in this course. Alright. This is a really good list with a lot of commands to remember, but don't worry, we'll be going into each one of these commands and maybe a couple more throughout this course. You'll be able to use each command, see what it does, and the common use cases for each command. It's a lot to remember, so to help you out with the description of a command and its available options, Git has a help subcommand that you can run on any Git command to get more information. In your terminal or command prompt, you can type git and then the command and then ‑‑help to get this information. So for example, if I type git add ‑‑help, Git will output all of the information I need for the git add command, starting with the brief description, a synopsis, and then into a longer description and all of the available options and descriptions of what they do for the entire git add command. This is incredibly useful and something to use if you need a refresher on a specific Git command or you just want to know what a Git command does.

### Coffee Shop Analogy: Steve and Wired Brain Coffee

Now that we understand some of the main benefits to Git and how it manages data, hopefully you can see that managing changes made to a file or files is Git's main purpose, and it does it extremely well. Understanding exactly how Git manages these changes is one of the most, if not the most, important thing to remember about Git. Everything Git does is based upon how it manages data. But before we get started learning the inner workings of Git, I found that it's much easier learning a new technical subject by using an analogy to make these concepts, well, more relatable. So for the rest of this course, we'll use an analogy of a coffee shop to help us understand more about Git. Our story starts with Steve. Steve is the single owner of a well‑known coffee shop downtown called Wired Brain Coffee. Steve manages all operations and has even developed most of the shop's recipes himself. In order for Steve to keep the popularity of his shop, he constantly tries out new ideas by altering existing recipes and changing out certain techniques and ingredients that he's used. He wants to keep the existing recipes, but also keep track of all of his changes in case he decides to make a change to the new menu item of the month. Currently, Steve stores all of these recipe files on his computer, and when he wants to make a change, he alters the actual file and then resaves it with a different recipe name. His approach to saving files is common and simple, but as we've learned earlier, it's incredibly error prone as Steve has made several mistakes and has even accidentally resaved over files he didn't intend to. Another issue Steve has encountered is the accessibility of these files. Steve occasionally works on his home computer as well as other devices when he's not at his work computer. This has made it difficult to keep up‑to‑date symmetry on the changes Steve is making across these multiple devices. Steve's whole approach is a very manual and time‑consuming process. Steve is looking for a better way to manage his files that adds both clarity and simplicity to the process. And Steve is also thinking about expanding his business and hire some store managers to help develop and work on his recipes, and when ready for team collaboration, he needs a process that is enhanced for easy collaboration and seamless team contributions into a managed location. So with this analogy in mind, let's dive into the fundamentals of Git and see how Steve can harness its power to improve the way he manages his recipe files for his Wired Brain Coffee shop project.

## Setup Your Local Environment to Start Using Git

### Git and the Command Line

Now that we have a good understanding of what Git is and how it manages our files between the different statuses and directories, we need to make sure that we have Git installed before we can start using it. In many cases, Git comes installed by default on most Mac and Linux machines, so let's verify that we have Git already installed, and if we don't, we can go through the steps to install it. If you're using a Mac, you can open up terminal, which is Mac's default command line prompt. Keep in mind that although this is the default program, you can download and use a number of other programs that give you different features and options. A popular terminal replacement for Mac users is called iTerm2, and it gives you features like split panes, autocomplete, and better on page search just to name a few, and you can navigate to iterm2.com and download this terminal emulator. But for the purposes of this course, I'll be using a Mac, and we'll stick to the default terminal prompt for all examples. Now, if you're using Windows, you'll need to use the CMD window, which is similar to the terminal for a Mac and is simply a command‑line prompt window, or you can use Git BASH, which tends to be the more popular of the two options. If you're on Linux, you'll need a terminal shell. Now using our command line, we can first see if Git is already installed by typing git version. If you get a Git version returned, well, that's great, you already have Git installed, or it will alert you that Git is an unknown command and that you need to install it. Here, you can see that I have Git version 2.37.1 installed. Now, for all those Mac users out there, you'll probably see in parentheses after the Git version Apple Git and then some number. If you see this, Apple has preinstalled Git on your computer with most likely an outdated version, but that's okay. With Git, there isn't many major feature changes with minor versions. Now, if you're a hardcore Git user, you may want to upgrade to each Git version to use the most updated subcommand and options, but for the average Git user out there, you'll most likely be fine working with several minor versions behind, and the core basics of Git will remain unchanged. So where is Git stored when we install it? If you input into your prompt type git, you'll see that for me the version of Git I'm using is stored in the usr/local/bin directory, and if I type /usr/local/bin/git and then space version, I see the version of Git that I'm using. Now, since I'm on a Mac, you might be wondering why I don't have that Apple Git version message by my Git version. That is because I have since upgraded my Git version from the one that Apple had installed, and my updated version is stored in my local directory which overrides where Apple installed the old version. We can see the original version of Git that was installed by typing /usr/bin/git and then space version, and then I can see the Git version that Apple has installed with their Apple Git message. Now, will it really matter if I use version 2.37 over 2.24? Well, in my opinion, again, not really. These are minor version updates, and the changes being made are 95% advanced nuances to subcommands that you can use. The basics and core fundamentals of Git will remain unchanged, but again, it's always a great idea to update or, at least, review the changes if you're thinking about it. Now, before we go over the options for installing Git, let's quickly cover a few shell commands for navigating the file system in your command prompt. Remember that these are not Git commands, but shell commands just used for navigating your command line. The first is pwd, which stands for print working directory, and this will display the file path from the root directory to where you are currently working. Mkdir means make directory, and this shell command is used to make a new directory in the file system wherever you are currently at. Ls is used to list the contents of a directory. If no arguments are provided, it will list the contents of the current directory. And cd stands for change directory and is used to move you throughout the file system. Touch is used to create a new file in the current working directory that you're in. These are just a few of the many different shell commands, but these are some of the basic ones that we will be using throughout this course to navigate our file system and create new files and directories that Git can track.

### Installing Git

Depending on what operating system you're using will slightly change how you install Git, so if you need to install Git or are looking to update it, we'll look at how to do this on a Mac, Windows, and Linux system. But regardless of what system you're using, the best place to download Git will be to navigate to git‑scm.com. This is Git's official open source documentation, and right here on the home landing page, you can see a prompt to download the latest Git version. Now, what's really cool is that this site notices the operating system you're using and will display the default download option for the system that you're on. So in my case, it's displaying the latest version for Mac, but if I wanted to download Git, I could simply click the Download button and it would send me to the instructions for downloading with Mac. Now, I have a couple of options. Probably the most common way is to use Homebrew to install or update Git. Homebrew is a free and open source software package management system that makes it really easy to install software on a Mac and Linux computer. It's really popular and easy to use, so I recommend installing it if you haven't already. Another option is to use MacPorts, which is another open source system used to install and upgrade software on your Mac. These are the two most common options, but again, you have a few others if you want. Navigating back to my terminal, I have Homebrew installed, and I used it to download Git, but I can also use it to update it. So, I can type brew install git, and it will install the latest version of Git if I'm not already using it. Now, if you're on a Windows, you'll have the Windows option by default provided to you on git‑scm, but you can always click on the Downloads button and then select the operating system that you want, and with Windows, you'll need to download the latest Git for Windows installer and then go through the Git setup wizard. It will ask you what you want to install. I recommend to select the defaults and then you can open Git up in the cmd command prompt, or you can use Git BASH, which is a version of the Bash shell similar to what you see on Linux systems, but with Git installed. Last, we have Linux. Now, fun fact, Git was originally developed to version the Linux operating system, so it only makes sense that it's easy to configure to run on Linux. If using Debian or Ubuntu, Git packages are available through apt‑get, and Fedora users is done through Yum or dnf commands. Now, regardless of the operating system you're using, once the command output has completed, you can confirm that you have Git installed by typing git version in your terminal or command prompt, and you should see the Git version as the output. Now, you may be wondering why we're using the command line instead of using a Git GUI, which is a graphical user interface for using Git. In fact, when you install Git, it actually comes with built‑in GUI tools that you can use or you can download third‑party tools for using Git. Some of these are GitHub Desktop, Sourcetree, and GitKraken. So, why aren't we using one of these tools? Well, the reason for that is actually simple. All of Git's documentation is written as if you're using the command line, so by learning how to use Git on the command line will be the best way to learn Git and understand its concepts. Then, once you have the hang of Git and you want to use a GUI, it will be much easier and you can always fall back to the command line.

### Create a New Git Repository

And now that we have Git installed, let's create a new Git repository. As we mentioned earlier, the repository represents your entire project, so when you initialize Git for the first time in a project, you are creating a new Git repository inside that project called .git. This is where Git lives and where it will store and track all of the file changes within your project. You can create a new project with nothing in it and initialize Git, or you can initialize Git with an existing project that already has a lot of content. Now, just keep in mind that wherever you initialize Git to a project, it will start version control from that moment in time. Let's now create a new project called demo and initialize it with Git. We can do this by running the mkdir command and then the directory or project name, and then don't forget to cd or change directories into it. Okay. Are you ready? Let's now turn this demo directory into a Git repository by typing git space init. Boom! Done. It's really that simple and easy, and it takes a second to run. You can see that Git returned with a message saying initialized an empty Git repository, and then the location of the new repository, which is in the .git folder inside my demo directory. Now, something else to mention is that you might see in the hint messages displayed at the top, depending on the Git version you're using, you may see these messages or not. If you're on a newer version, you'll see that Git is trying to move away from the default naming of master for the initial default branch to use the naming of main instead. It then gives you a git config command to use to make this a global setting that will take place for all future Git repositories that you create, or you can just change the name of the branch for this repository only. Now, we'll be talking more about these configuration settings in just a minute, and this is something we'll want to include, but for now, let's use this second git command to change the name from master to main. Now in your prompt, type git branch ‑m and then the name we want which is main. You can see that we just changed the branch name from master to main. Now that we've installed our Git repository, it's something we only need to do once. Now, we can run git init again and again and Git will just tell us that it can reinitialize the existing Git repository, but nothing is really happening since we already did it. It won't override anything, but it won't give us an error message either. So let's see what git init really did. Now, if we type ls to list the contents of our directory, we don't see anything, but this isn't true because we know from earlier that we should see a .git directory. You can type ls ‑a and that will list all hidden files and folders, and check it out, we now see the .git directory. Now, real quick, don't ever, and I mean ever, remove or delete this .git directory. This is where all of the Git tracking and data is stored in your directory, so removing this will remove everything you are having Git version control for you in this entire project. This is like deleting the memory card from your camera. You'll lose all of the precious family photos from years past, and you won't be able to view the events that happened leading up to the present moment. This is what turns your project or directory into a Git repository. Okay. So now that our project is a Git repository, what is the status of our repository? Will Git tell us that? So let's type git status. Git status is probably the most common and most used Git command, use it often to understand the status of your Git repository and the state that your files are in. Git will tell you the branch you're currently on, which is main, if we have any commits that we've made and if we have any changes in our working or staging area waiting to be committed. Now, this is all true because we haven't done anything in our repository since we initialized it. We haven't created or modified any files, and as we make changes to our repository, you can run the git status command and Git will start to output this information to you so you can know what is going on at all times. But before we start adding and modifying files, let's take a quick look at that .git directory that was created and see what's inside. Let's cd into .git and then type ls to list the contents of this directory. Here, we have HEAD, which is something we'll cover a little bit later, but it's basically the pointer to the current branch reference that you're currently checked out to. Since we're on the main branch, our head is currently pointing to the main branch reference. Now, at this point, don't spend too much time thinking about the other things you see listed here in this docket directory. For the common Git user, you won't ever be getting into this docket directory, but it's good to know about it if you want to learn more. There are other things like hooks and objects and rests that all represent how Git is tracking the changes in this repository, and once you have a good grasp of the fundamentals of Git, you can dive into what these mean and how the content is actually being stored, but for the scope of this course, just knowing that the Git directory is where Git does all of its magic is good for now.

### Setting Git Configurations

However, there is one thing in this .git directory that I want to point out and it's the config file. If you remember when we ran the Git init command to initialize our Git repository, Git provided a hit message that told us that Git is trying to move on from using the term master to start using main as the initial default branch. Git provided us with a Git command that we can run to set a Git configuration that will rename all future Git repositories to use main, instead of master. The command was a git config command to set the default branch when created with Git to be a specific name. We can create Git configurations that will allow us to set certain settings when using Git. In fact, there are two Git configurations that you are required to set before you can make any commits to the Git repository, but before we do that, let's talk through the different configuration options. Git configurations are done at three different levels. The first is system, and these are configurations set at the system‑wide level and applied to all users on a machine. For what it's worth, I have never set a system‑wide setting when using Git because I have been the only one using my computer, but you can use system‑wide configurations if multiple users use the same machine, and you want to standardize some settings between the different users. The next level is called global. Now, these are settings at the user level and will apply to all Git repositories created under that user. These global settings are common and the ones that I use and see others using the most. We'll be setting some global Git configurations in just a moment so that they are set for every repository we work in, but the next level down is called local, and these are specific settings to a single Git repository that will only apply to that one repository. Global configurations will take effect unless you have a local configuration set for a specific Git repository and then the local setting will overwrite your global ones, but again, just for that specific Git repository. Local configurations are useful if you want to work in a repository that has different credentials than the global ones that you have set. Now, this can come in handy if you have global settings with your company or school credentials, but then you want to contribute to personal or maybe open source Git repositories with your own set of credentials. And the great thing about setting those Git configurations is that you set these the same way, but just change the wording from either system, global, or local, and just a minute ago, I mentioned that Git requires two configurations before you can make commits to your Git repository. Git needs a user.name and a user.email that will be assigned to every commit you make, so let's set those right now. In our command prompt, we can set a Git configuration by typing git config ‑‑global user.name and then the name you want in quotes. Next, we need to set our email by typing git config ‑‑ global user.email and then the email address that you want associated with your Git commits, and that's it. We can see what we did by typing git config ‑‑global ‑‑list, and Git will output our global Git configurations. And check it out, we see our required configurations that we just set. Now really quick, if you recall again that message that Git gave us to change the name of master to main when we run git init, Git gave us a global configuration that we can run that will change the name of that initial default branch to main, so let's do that right now. We can do this by typing git config ‑‑global init.default branch and then the name we want, which is main, and to verify that we did it correct, we can run git config ‑‑global ‑‑list to see our global configurations. And nice, we now have that configuration set. If for any reason we need to remove a configuration, we can use ‑‑unset after level name of the configuration and Git will remove that setting. So if I type git config ‑‑global ‑‑unset, Git will remove that setting and then we can verify this by checking the global config list again, and sure enough, Git has removed that setting. Well, we need this setting so we can set it again by typing git config ‑‑global user.name and then our name again in quotes. If I wanted to set a local setting, I just change the wording of ‑‑global to ‑‑local or ‑system, whichever I need, but if you don't provide a level configuration name when setting a configuration, by default, Git will set that configuration as a local setting, and you can view those by typing git config ‑‑list or git config ‑‑local ‑‑list.

### Set a Default Text Editor

With Git installed and the required configuration set, we are now ready to start using Git to track and manage changes to our project. But before we jump over to Steve and his Wired Brain Coffee Shop project, let's talk real quick about text editors. Now, for this course, we'll need to use a basic text editor to interact with the files in our project. There are a lot of options out there, and you can use almost any text editor. Some common ones are Visual Studio Code, Sublime Text, TextMate for Mac, Espresso, Notepad++ for Windows, and Vim just to name a few. Now, I won't be going into how to install each one of these text editors, but with a quick Google search, you can navigate to their websites and download the applications. For this course, I'll be using Visual Studio Code to make all of my file changes, but I'd also recommend Sublime Text as both of these work across Macs, Windows, and Linux. Now, once you have your text editor installed, you can navigate to the Git repository and be able to add, remove, or modify its files from your editor. One cool thing you can do is use your terminal to open up your text editor exactly in the directory that you're currently at in your terminal. So I can navigate to my Git repository in my terminal and then type code space dot to open up Visual Studio Code. I can do this by first opening up Visual Studio Code and open up the Command palette by navigating to the View tab and then the Command palette or using the hot key of Cmd+Shift+P. I can search for shell and then select Shell Command: Install code command in PATH. Now, when you use your terminal, you can just type code space dot and Visual Studio Code will open right up. You can do something similar with other text editors, but the process may be slightly different. So if you're using something different other than Visual Studio Code, you may need to do a little research to see how to achieve a similar effect using your editor, but you can always just navigate to your project using the editor, instead of opening it up with the command prompt. Nice. Now that we have a text editor ready to go, it's time to jump over to Steve and his Wired Brain Coffee Shop project and see how we can start using Git to track and manage its file changes.

## Working in a Local Repository

### Add Git to an Existing Project

So now that we've gone over the main concepts of Git, the next best thing is to see these concepts and Git commands in action by using Git in a project and seeing how we can add files, make changes to them, create branches, see our different file statuses and states, and how to merge branches and view our commit history. So to do this, let's use the story of Steve we mentioned earlier and see how we can use Git to manage his files and his Wired Brain Coffee Shop project. Now that we're back at the command prompt, let's navigate to Steve's project which I have under this directory labeled pluralsight. Typing ls, I can see the Wired Brain Coffee Shop project, so let's navigate into this project by using cd and then the name of the directory. As you can see in my command prompt, I know exactly where I'm at since I have the path provided to me with some fun colors to help separate this information. Now, I'm guessing that this will look pretty different than what you're using, especially if you're using Terminal or Git BASH for the first time. Now this is because I have set up a custom bash profile to customize how it looks and the information that it provides, which makes it look real cool, but it also makes it easier to know the working directory I'm in and the branch I'm currently checked out to when working in my Git repository. So if you're interested in learning how to customize your own command prompt, we won't be diving into that in this course, but I'll provide some helpful resources at the end and some recommended next steps for learning more about Git. But for now, if you want to confirm where you're at, you can always type pwd for print working directory, and it will output the exact path that you're checked out to. Now, typing ls, we can see the contents of this project which lists three folders, hot‑beverages, iced‑beverages, and syrups, and if we type ls ‑a, we see all of the contents including any hidden directories or files. Now, we don't see the .git repository inside this project, and we can confirm this by typing git status to show us the current status of our project. And just as we expected, we get a message telling us that this is not a Git repository and that the .git directory is not found. So let's make Steve's project a Git repository by running the git init command to initialize Git. And with a fraction of a second, Git has initialized this project as a Git repository, and we can confirm this by running ls ‑a and Git will output the contents of this directory, including the newly‑created .git repository. Notice that since we set a global Git configuration to change the name of master to main, when we initialize Git, Git created the default branch with the updated name. And nice, our Git configurations are working as expected. Now, I want to take a moment to mention that throughout this course, we'll be working in several different Git repositories to demonstrate the fundamental commands, but most notably, we'll use this demo Git repository of Steve and his Wired Brain Coffee Shop project to demonstrate most of these commands. Now, of course, you can sit back and just watch and absorb this information, but if you want to follow along as you watch, you can navigate to this URL which will take you to an exact copy of the project which you can then download to your local computer and follow along. It's in a GitHub repository, and all the instructions on how to start using it are in the README file which is shown front and center when you navigate to this link.

### Adding and Tracking New Files

Okay. Now that we see the .git directory inside of our project, we have added Git and this is now a Git repository, so let's run the git status command again to see what has changed. Git now gives us some different information. Now, if you recall when we ran git init on an empty directory with no existing files or folders, when we ran the git status to check the current status of our repository, Git told us that we had no commits and that we had nothing to commit, meaning we don't have any existing files to add for Git to start tracking. Now with Steve's project, we have existing files when we ran git init, so now Git is telling us that we don't have any commits yet, which is true, we haven't committed anything yet, but now Git tells us that we have untracked files. Remember, these are files that exist in our project, but we haven't explicitly told Git yet to track them. Adding Git to a project won't automatically start tracking all of its contents. We need to tell Git to track each file, and the last message tells us that we have nothing added yet to commit, meaning we don't have any changes to files that we've added to the staging area waiting to be committed. Remember, git status tells us the current status of our Git repository in relation to these different file states. Content that is untracked, you'll see as untracked files, and content that is being tracked, will show up as committed in our working directory or in our staging area. We'll be moving files throughout these different states all throughout this course, so you'll get a lot of practice. We can start tracking these files by adding them to our staging area to be committed. Now, when working in Git, there are often several ways to accomplish the same task. Here we have three folders inside our project that are being untracked, so we need to add them using the git add command, but we can do this a number of ways. One thing we can do is use git add dot to add everything. Now, this can be useful, but you need to be careful when using the git add everything command because you may add something you're not intending to do. So let's try it out and see what happens when we run git status. Okay. We now have new information. Git is telling us that we now have changes to be committed. Again, these are changes in our staging area waiting to be committed to our Git repository for tracking. The three folders, hot‑beverages, iced‑beverages, and syrups all have several files inside of them that were included when we added the folders. Now, I can commit these now and Git will start tracking them, but what if you wanted to add a file or folder to your staging area, but you also have changes you don't want to add at the same time. Well, the git add dot command wouldn't be what you want because it adds everything in the working directory, so let's remove these changes from the staging area and go back to having them be untracked. We can do this by using a command called git reset, and then to confirm, we can type git status again to see that we're back to where we were before we ran the git add dot command. Now, git reset can be really useful, but it does deserve its own section, so we'll spend more time with git reset and its options a little bit later on in this course. The other way to add changes, and this is the recommended way so that you don't use the add everything command, is to identify each file or folder that you want to add. Now, we can do this by typing git add and then the name of the file or folder that you want to add. So I can type hot‑beverages to add that directory or folder. I can hit Enter and then repeat this process to add the other two, or I can just add them to the same command by using a space in between each file or folder name. So if I wanted to add only these three files to my staging area, it would be git add hot‑beverages/ space iced‑beverages/ space and then syrups/, and running git status shows all three folders and their contents added to the staging area waiting to be committed. So let's go ahead and make our first commit by typing git commit ‑m and then a message in quotes. When making a commit, Git requires you to provide a commit message. Now, this is just a brief message to outline the changes that you're making in this commit, and you'll see why this is important in a few minutes when we look at our commit log history. The ‑m stands for message, and then we can type a message in quotes, so let's type Add hot beverages, iced beverages, and syrups. Now, after this command runs, Git will output some information letting us know that the command worked, or if you get an error, why it didn't work. Here, we see that everything was committed successfully to our Git repository. We see that it added 14 files and then the list of files that it added. You'll also see number data next to our commit message and next to the files that were added. Now, the number next to our commit message is called the commit ID. Now, we won't be diving into the inner workings of Git in this course, but what's important to know at this point is that with each commit, Git gives it a unique ID for tracking and reference. This ID is unique to each commit and it's comprised of 40 characters, but often, the first 7 characters are enough to link to the commit, so Git will only show you the first few, instead of showing the entire 40 character ID. This is the identity of the commit and the integrity of how Git tracks changes. Now, we'll look more into this when we look at our commit logs, but the 100644 next to the files we added is just an identifier of that type of file. The 100644 means a regular file and other file types will have different numbers next to them. And again, this is diving more into the inner workings of Git, so we won't go much further into this specific set of information, but it's good to have a basic understanding of what this information is because you'll see it every time you make a commit. So let's now run git status and see what we see. We get a short, clean message telling us that we don't have anything to commit. No untracked files are being identified, and we don't have any changes in our working directory.

### Making Changes to Files

We now have a Git repository, and we have made our first commit to add files to this Git repository so Git can start tracking them. So far, we have just been adding entire files to be tracked, but what if we want to make changes inside of a file that is already being tracked. Well, this is a main function of Git, so let's see how we can do this. To clean things up a bit, let's run clear to bring our command prompt back up to the top of the screen. Let's now open up our text editor and navigate to our Git repository. In my text editor, I can see my project and the folder and files inside of it. A nice feature with VS Code is that I can open up a terminal or command prompt right inside of my editor complete with my custom bash profile and all of my Git configurations that I set up. In looking at our files, let's make a change to one of them and see how Git responds. Let's open up the caramel.md file in the syrups folder. Now, the h1 header of this file is incorrectly titled. It says Cinnamon Dolce Syrup, when it should say Caramel Syrup. So let's make this update and then save the file in our text editor. Now running git status, we can now see that the caramel.md file shows as modified and in our working directory. Remember that changes in our working directory will not be committed if we run the git commit command. We first need to add this file to the staging area and then we can commit it. So let's add this file using git add and then the file name, or you can run git add dot to add everything in the working directory to the staging area. And since we are confident that this is our only change, let's do the first option. Now, if we run git status, we see that our file status changed from being in the working directory to now residing in the staging area. Now, let's commit this file using git commit ‑m and a commit message of Fixed caramel file header. When that commit runs, we see that git has committed that file change to the Git repository and it's assigned a commit ID with the information on what occurred in that commit. We see that we made one file change with one insertion and one deletion. This may be confusing because, at first glance, you may think that Git is saying that we deleted a line and then added a different line, but essentially, that's what we did. We removed a line that was previously committed in our Git repository and added something new in place of it. So now if we run git status, we see that we don't have anything to commit and that our working tree is clean. So let's visualize really quick what we just did. We had the caramel.md file committed to our Git repository and the .git directory. When we made the change to fix the file header from the Cinnamon Dolce to Caramel, that change moved the file into the working directory. Since this was the only change we wanted to make, we added this file from the working to the staging area by using the git add command, and once in the staging area, we committed the file change to our Git repository by using the git commit command and the file moved from the staging area back to the .git directory as a committed change. And when we ran git status, our working tree is clean, meaning we don't have any changes remaining in our working directory or our staging area that are waiting to be committed.

### Creating a New Branch

So far, we've done everything on our main branch, but since branches are an essential concept in Git, we can and should use multiple branches when committing changes to our project. It's not good practice to make changes directly on our main branch. Of course, with Git, you can always reverse those changes with version control, but branching allows you to work on new ideas and try out new things without affecting your main branch, so let's try this out and create another branch. In the command prompt, you can create another branch and name it whatever you want. Now, depending on the nature of the branch may help you decide on what to name it, and some projects that I've worked on ask you to prefix the branch name with specific naming for automation and continuous integration and continuous delivery purposes, but for our case, we can just create a new feature branch, and so we'll call it feature. So let's run the Git command, git branch feature. Let's now run git status to see if Git outputs any different information. We see that Git tells us that we are still on the main branch and that we don't have anything to commit and that our working tree is clean. So this looks exactly the same, and we're still on the main branch. Remember with Git, it won't do anything unless we explicitly tell Git to do it. We only told Git to create a new branch called feature, but we're still checked out to the main branch. So if we type git branch and then use the ‑a as an option, it will display the local branches we have in our Git repository. And look, we see two branches, main and our new feature branch. Notice that there is a star next to the main branch and that it is highlighted in green. This is Git just telling us that we're currently checked out to the main branch. So to switch to our new feature branch, we need to tell Git to check us out to that branch, and we can do that with the git checkout command and then the name of the branch, so we can type git checkout feature. Notice now that my command prompt now shows that we're on the feature branch, but if we ask Git to list the branches again with git branch ‑a, notice now that we are now checked out to the feature branch and Git confirms this by applying the star next to the branch that we are on. As I've mentioned before, Git only does what we ask, but there is more than one way to do the same thing. A popular option when creating a new branch is to use the git checkout command, but also use the ‑b option that allows you to create a new branch and then check out to it at the same time. So for example, if we wanted to create this new feature branch, but also check out to it at the exact same time, we could have used the git checkout ‑b and then the name of our branch, which was feature. This command creates and then checks out to the branch at the same time, it's really useful and one that I use all of the time.

### Viewing the Different Files Statuses

Now that we're on a new feature branch, let's make some changes to this branch. Steve has received some feedback on altering his cafe latte recipe, so let's make that change. Navigating to the cafe‑latte.md file under hot‑beverages, Steve wants to change one of the ingredients. He wants to change 1 cup of whole milk to 1 cup of almond milk. We can save this file, and now in our command prompt, let's run git status to see the file change show up in our working directory. Perfect. We see that our change is not yet staged for commit, but it remains in our working directory. So for now, Steve is done with this file, so let's add it to our staging area with git add and then the file path and name. Or again, you can add it with git add dot, and by running git status, we see that our file change has now moved into our staging area. We can now go ahead and commit this change, but Steve wants to quickly make another change before he forgets about it. He wants to add a new step to the frozen‑mocha‑blended recipe under iced‑beverages. After Step 3, Steve wants to add another step called Step 4 with the instructions of Top with whipped cream and chocolate shavings. So let's go ahead and add that new step. Let's save the file and run git status. Okay. We now see multiple file changes in multiple file states. We have our cafe‑latte change that is staged and waiting to be committed, and now we have our new Step 4 change in the frozen‑mocha file still in our working directory. And at this point, it's late on a Friday night and Steve decides to log off for the weekend. Now come Monday morning, Steve is eager to jump back into work, and he remembers that he has made changes, but he doesn't remember exactly what change he has made. And as we can see with git status, we see which files are in the different states, but we don't see the file contents to know exactly what changes have actually occurred. Of course, Steve can always go to the text editor and scan through it to see what has changed, but Git provides a useful Git command that will show us the file differences that are occurring in both the working directory and the staging area. This command is called git diff, and there are several different options that we can use with it. Now, if you want to see just the file changes that reside in the working directory, you can run git diff, and if you want to see the changes that reside in the staging area, you can run git diff with the ‑‑staged option. And if you want to see both changes in the working and staging area, you can run git diff head which shows all changes that haven't yet been committed. So let's try each of these options in our command prompt. Now, running git dif, we should just see the change that resides in our working directory. Here, we see the new step that was added to the frozen‑mocha file, and we see exactly what was added or removed. Now, this is a great way to see the actual changes that are in this state, and it shows exactly the change that we should expect to see. So let's now run git diff ‑‑staged, and here we see the changes that are in our staging area waiting to be committed. We can see that we removed the line of 1 cup of whole milk and replaced it with 1 cup of almond milk. Now, you'll notice additional information that Git provides that defines the location in the file where those changes take place, but again, this is going deeper into the inner workings of Git tracking and so we won't be covering all of this tracking data in this course, but I like to point it out so you have an idea of what it means so you're not completely lost when you see it. And the last option is git diff head, and as you probably expected, it shows all changes, both in the working directory and the staging area if you want to see both changes side by side. So by using git diff, Steve is able to recall his changes without having to look through his files in his text editor. Let's now run clear to give us a clean command prompt and run git status to see the files again. Let's now run the git commit command and give it a message of Change whole milk to almond milk, and then running git status shows us that only our stage changes were committed and the changes in our working directory didn't move. This lets us keep changes in our working directory while still committing other changes. So let's now add them to the staging area and then run the git commit command with the commit message of Add new step to frozen mocha. Now, if we run git status, we should see a clean working directory and staging area. So let's run that now to confirm what we expect to see. We see a clean working directory with nothing to commit.

### Merging a Branch into Another Branch

We've now made several different changes to different files in our Git repository, and we've even seen how we can keep work that is in progress in our working directory while still being able to commit changes to other files. This is a basic use case of using Git to make changes to multiple files that can be widely different from each other and not feel like you have to stop your work and always move that change from the working directory to the staging area to commit every time. You are free to make lots of different changes and then choose what you want to add together when you commit. Now, this works well if you only want to add entire files and all of their changes to the staging area for a commit, but what if you wanted to make different changes within the same file and then only add some of those changes while leaving others in the working directory. Well, Git allows you to do this as well. It's beyond the scope of this course, but if you're wondering how that would work, you would use the git add command and then the ‑p option which stands for patch, and this will allow you to scan through the changes in one file and select the specific file line changes that you want to add to the staging area while leaving the others behind in the working directory. It works a little bit differently than just say git add because you need to say either yes or no to each file change to be committed, but after you identify the file changes you want to add, committing the changes works the same. So if you're interested in learning more about the patch option with the git‑add command, check out this link to the Git documentation and it will talk more about it. Alright, now that we have added a few commands to our feature branch, let's return back to the main branch and make another commit so that we have a true branched history from the changes that we've made on the feature branch. Let's run git checkout main and navigating to the vanilla file under syrups, let's add a pinch of salt to the list of ingredients. Let's save and run git status to confirm that it's in our working directory. Let's now run git add syrups/vanilla.md to add the file to our staging area, and then we can run the git commit command with a message of Add pinch of salt to vanilla ingredients. And to confirm, let's run git status again, and we should see a clean working tree. Perfect. So now we have a true deviated history. We have created a feature branch off of the main branch at a specific commit location and then ran two more commits on that feature branch. We then went back to our main branch and made a different commit only on that main branch. So let's merge our feature branch into the main branch so that those two commits are now incorporated into the main branch timeline. Now, since we've made these changes on the feature branch, our main branch doesn't have these changes yet, so we need to run the git merge command to merge our two branches together. It's important to note that you want to make sure you're currently checked out to the branch that you want to merge into. So since we want to merge our feature branch into the main branch, we want to be checked out to the main branch when we run the merge command. Remember, if you don't have it displayed in your command prompt, you can type git branch ‑a and Git will show you which branch you're currently on, or you can always use git status to show you the branch you're currently checked out to. Now that we have confirmed we're on the main branch, let's run the git merge command. Now, when you run the git merge command, Git is actually making a commit on the branch you're merging into that includes the changes from the other branch. So since we're basically making a commit, we need to provide a commit message whenever we merge. So we can type git merge feature ‑m and then a commit message, so something like Merge feature branch into main. And just like when we made a commit, Git will respond back with some output information to let us know what happened. We see that the two file changes from our feature branch were added to main and Git provides a message to let us know what type of merge strategy it used. Now, Git has multiple ways that you can merge branches together, and each way has its own benefits and outcomes. We won't be diving into these different merge strategies right now at this time, but just know that the ort strategy is a default merge option for newer Git versions and is a type of what's called a three‑way merge. If you're running an older version of Git, you'll most likely see what's called a recursive merge strategy. Now knowing the difference is beyond what you need to know at this point in the course, but again, I want to point out so you're at least familiar with what Git is telling you. Now these are basic merge strategies for just merging two branches together. Now going through this for the first time, it's really helpful to see a visual representation of what we did, so let's run through these same steps again in a visual way. Here we have our main branch. Head is just a Git pointer to where we are currently checked out to. You'll see this change when we check out to a feature branch. The first thing we did on main was create a couple of commits, the first being that commit that added all of our files from untracked to committed in our Git repository, and the second commit to fix the header in our caramel recipe file. Now, at this point, we decided to create a new branch called feature. Now even though we created the new branch, it's still pointing to the last commit on main, so essentially these two branches as they currently remain are exactly the same. Now notice the head marker as we change from being on the main branch to the new feature branch. Since we are now checked out to feature, head moves with us. We now made two commits on our feature branch, one was to change whole milk to almond milk and the other was to add a new step to the frozen mocha recipe. Notice now that the feature branch has moved along in history and the main branch remains in place. At this point, we switched back to the main branch to create a new commit on main so that we had a true deviating history between the two branches. Notice again how head follows where we are currently checked out to. And last, we ran the git merge command while on the main branch to merge the feature branch into the main branch. Now, if this is still confusing, don't worry, we'll spend some more time on these different merge strategies, but hopefully this visual helps you understand how we created a diverted history with our two branches and then merged them together with the new merge commit that points to the two branch timelines.

### Viewing Your Commit History

We've now made several commits in our Git repository, and as we continue to make more and more commits, it would be nice to see a history of our commits without having to know how to navigate the .git directory and try to find our commits there without messing things up. So, let's return back to our command prompt and see how we can run a Git command to list out the history of our commits. And you have probably guessed it, but Git provides a log command that will list the commit history of our Git repository. It includes all branches, and there are a lot of options that we can use to define exactly what information you want to see. In fact, git log has well over 100 options that can be used together to create all sorts of outcomes. Now keep in mind that git log is a show command, so all options are just ways to display the information that can't be used to change or modify anything in your Git repository. So, let's clear out our command prompt and run git log. This is the git log command without any options. It displays each commit in history, starting with the most recent commit. It displays the entire commit ID. Remember, each commit has a unique ID assigned to it, and when we are adding changes from our working directory to our staging area, we saw just the first six or so characters of this ID. So git log provides the entire ID for each commit. You can also see the author of these commits and the date and timestamp that the commit occurred and the commit message for each commit. And I also see where my feature branch is still pointing, as well as where main and head are currently at. And hitting Q will quit the log and return us to the command prompt. Now this is good, but maybe I just care about each commit and its message. I don't care too much about the timestamp and the author of the commit. So, let's clear out our prompt and run git log ‑‑oneline. This option simplifies each commit to one line with the shortened commit ID, similar to how we've seen it before. And we have each commit message, and we can see where each branch is currently at. Now, this is better, but it would be nice to see a visual representation of how our branch is diverted and then when they were merged back together. We can do this by running git log ‑‑oneline and then ‑‑graph. Now we get a one‑line visual representation of each commit and when our feature branch was created and when it was merged back into main. Now to clean things up, we still see our feature branch hanging around. Since we have merged it into main, we know no longer need that branch, so we can delete it. Remember, a branch is really just a pointer to a specific commit in time, so deleting a branch that doesn't have any remaining changes won't remove any content from the Git repository. Of course, if you've made changes on a branch and delete it without merging it into another branch, you'll lose the content on that branch. Now we can do this by typing git branch ‑d for delete, and then the branch name, which is feature. Git then provides a quick message telling us that it deleted the feature branch that was pointing to this specific commit ID. And running our git log ‑‑oneline ‑‑graph command again, we see that our feature branch is no longer showing in our log history. And we can also confirm this with the git branch ‑a command to list our branches, and we only see the main branch listed. We've done a lot up to this point. We added Git to Steve's project to make it a Git repository. We then added the files to the Git repository with our first commit, and we learned how to make changes in a file and move them from the working directory to the staging area with the git add command. And then we learned how to create new branches and how to see the different changes when we had files in both the working and staging area using the git diff command. We then learned how to merge and delete branches and how to view the commit log history with some useful options to simplify the commit information to one line and also provide a visual representation with the graph option. Now, at this point, we've done everything locally, so next, we'll learn about working with remote repositories and how to collaborate with others in the shared Git repository.

## Working with Others in a Shared Repository

### Git on the Network

Now, this part of the course is all about working with Git on the network, and if I was going to guess, I would assume that you and most people working with Git will be doing so while working with a remote network. Now this is a centralized place for you and others to collaborate together to share changes to the project in the form of Git commits with team members and other collaborators who have access to the network. If you recall from earlier, one of Git's characteristics is that it is local. That means that everything you do in Git is done locally on your computer, even if you're working in a shared Git repository on the network, but wait a minute, if we're working on a shared repository, do we make changes directly on the server? And the answer is no. Everything in Git is done locally and then shared with the repository on the network for then others to see and then retrieve. Now this centralized Git repository is just a copy of the same repository that you and others copy down to work on on your own local machine. Essentially, you can think of this centralized Git repository on the network as just a branch that you want to exchange commits with. You can send or push your commits to the centralized repository and then you can also pull down any commits that you don't have that others have pushed up using their own copy of their repository so that you can have them on your own local copy. The operations of sending commits to a shared repository or server in the network and retrieving commits are two basic operations that you'll perform when working on a shared network. So we talked a little bit about this earlier, but there are several options that you can use when working with Git on the network for collaboration. The most popular options are Git repository hosting services like GitHub, GitLab, and BitBucket. Now these are cloud‑based hosting platforms that you can use as that centralized main version of your project that everyone copies to their own local machines and then pushes changes back to the network for others to see and then build on, and all you have to do to keep things up to date is pull down any changes from the centralized version of your project and then you're good to go. Now, since Git is distributive in nature, everyone has their own local copy of the entire project, so you can perform all of your Git commands and changes locally off the network and then you can push them up to the network when you're ready. This makes Git incredibly powerful and very nice to use since you're not dependent on the network to work on your project. Another thing to mention is that if you didn't want to use a code hosting service like GitHub, GitLab, or BitBucket, you can set up your own Git server. Now it's completely free, and the tools are open source, and you can get things up and running in about an hour or so. Now, this approach requires a lot more maintenance and setup, and you won't have a lot of the extra collaboration features and tools that these cloud hosting services provide. So setting up your own server is definitely beyond the scope of this course, so we'll stick to one of these cloud platforms for now, but it is nice to know that you can always spin up your own Git server down the road if you ever want to.

### Setting up a Remote Repository on GitHub

Now, Steve is anxious to get his project up on the network so others can view, collaborate, and contribute to his project, and it was just mentioned, there are a number of code hosting platforms that we can use to establish our central Git repository on. So we'll pick the popular choice of GitHub to demonstrate how we can set up our Git repository on a GitHub server and use it to collaborate with others by sending and retrieving commits to our project. Now, this isn't a GitHub course, so we won't be going into the details of setting up a username and going through the GitHub interface to show you its capabilities, automation, and collaboration features like using a pull request. Now, all of these tools and features extend Git's ability to collaborate and manage your project while making your life easier when working with Git. So whatever platform you choose, make sure to check out their list of tools and features because they will improve the way you collaborate with others. But for now, we'll stick to the basics of just setting up a remote Git repository on GitHub so that we can use it to send and retrieve changes, the two basic operations when working on a shared network. Now, once you've signed into your user profile and navigated to the Repositories tab, you'll see an option to create a new repository. We'll need to give it a unique name, something to call the repository on GitHub. We'll use the same name as Steve's local project so that we can keep things consistent, which is coffee‑shop‑recipes. Now, we'll give it a brief description so that others can be sure that this is the correct repository to contribute to and something like a place to store, create, and explore new coffee shop recipes. We'll make it public so others can see it, and we'll skip the other settings for now. We can select the button to create the new repository. Now, we have a couple of options that are presented to us. We can create a brand new Git repository locally on our command line and then link it to this one on GitHub as the central copy that will remain on the network. This is the option if you're starting with a new project and want to immediately create the link to that central repository on the network. And as you can see with some of the instructions, it'll ask you to run git init to initialize the Git repository, and then it will have you add a README file, which is pretty standard for Git repositories and is a place that you can provide instructions for collaborating on the project or a detailed description of what the project does or pretty much anything else you feel is important to add. Now, Steve doesn't have one on his project yet, but he can add one at any time. It will then instruct you to rename the default branch from master to main and then add the remote link using the git remote add origin command with the URL to the central Git repository on GitHub, and then it will ask you to run git push to complete that link and send the project to GitHub. However, we have a project that already is a Git repository, so we can follow the second step to push an existing repository that is local to GitHub, and as you can see, we can skip the first several steps, and the last three steps in both situations is exactly the same. In fact, we only need to run two commands to complete the link to this new Git repository on GitHub. And since we already told Git to switch our default branch from master to main, we only need to run the git remote add origin command and then the URL to this Git repository on GitHub that is provided to us. Now, before we run these two commands, let's take a minute and better understand the git remote command. The git remote command is responsible for creating the link to other copies of the repository and is used for syncing changes between those two copies. This command is also used to view the links to those connections and then also delete them. These registered copies of the project use specific Git commands that are responsible for sending or retrieving changes to and from the network, and we were introduced to these commands earlier, and the ones you see here are the most commonly‑used remote specific commands. In fact, I can't think of one that I've used in the last several years that isn't on this list. Now, we'll see how to use each one of these remote repository commands and what they do starting with git remote. Now, when adding a new git remote, you can run the git remote command and the add option. You can then give this remote a name that can be used to call this remote connection instead of the long URL that was provided to us each time you want to send and retrieve commits. Now, the default name is typically called origin and that's what we see in the instructions from GitHub, but you can name this really whatever you want. You then provide the full URL that was provided to you from the code hosting site, and this command will create a new connection to the remote repository on GitHub. And when we want to send and retrieve data between the two, we can use the name we provided, in our case origin, instead of providing the URL each time. So let's copy that link from GitHub and run the command locally on our command line. Let's now make sure that we are in the correct location. We can type pwd for print working directory to double‑check, and then let's type git remote add origin and then paste in that URL and then hit Enter. Now from the looks of things, it's hard to know if Git did anything because Git didn't return anything for us to see, but more often than not, this is a good sign and the command worked. Sometimes no news is good news. So let's see if we can investigate a little bit to see what happened. Now, we should be able to run just git remote with no options and it should output any linked remotes that we have set up, and running that command, we see a remote connection called origin. Nice. This is exactly what we just asked Git to do, but let's investigate a step further and make sure that this origin connection really is pointing to the URL that we want. Now, we can run git remote and then add the option of ‑v, which stands for verbose, which will give you more detail and we'll show you the remote URL next to the name we gave the remote connection, and running that command, we see two connections to origin that we have the same URL pointing to our coffee‑shop‑recipes Git repository on GitHub. Nice. This is perfect. We now have a remote connection to our linked Git repository on GitHub.

### Sending Local Commits to the Network

We just linked our local Git repository to a Git repository on the network by using GitHub, but you may be wondering why we have two origin connections to the same URL, instead of just one. Now, this has to do with the two basic operations when working with remote repositories. Git refers to sending changes as a git push and retrieving changes as a git fetch or a git pull. We'll go over the git pull and git fetch commands in the next section, but here, let's focus on the git push command. So looking back at the instructions that Git had provided to us, the last step is to run git push ‑u origin main. So let's dissect this command to see exactly what we're being asked to do. Git push is the Git command to send or push any commits we have locally that are not yet represented to the remote Git repository on the network, but just running git push for the first time, we need to tell Git where we are sending or pushing our local commits. So we need to add the option of ‑u, which stands for upstream, or specifically setting up an upstream to where we are pushing the commits to the remote repository. Remember that every commit we make with Git needs to be made on a branch. Now, the same is true when working on a remote Git repository, so just telling Git to send our local commits really isn't enough. We need to tell Git exactly how we want these commits to be pushed to the remote repository by providing the branch we want these commits to be on, but sometimes, the remote repository doesn't recognize the branch we want to push changes to. So this ‑u option needs to be used the first time we want to push any commits on a branch that we created on our local repository, but not yet on the remote repository. So this ‑u option tells the remote Git repository to create a remote tracking branch on the network that will link to our local branch. Once we have set up that upstream connection that links the branches together, we'll be able to just say git push the next time we want to send any commits from our local repository to the remote repository using the same branch. So if we created another local branch to make some different changes and then want to send those changes to the network on that new branch, we'll need to use this ‑u option again just for the first time to establish that link for the new branch. Next, we tell Git we want to use the origin connection that we made. This is telling Git the exact location we are using with the URL, and then we tell Git what branch we are sending the changes to. So again, in our case, we are making these changes on the main branch so we list main. The reason we see two different origin connections for pushing and fetching is because, by default, well, they are the same, but we can establish different remotes for each of these different operations. Perhaps you want to fetch or pull from one location and then push changes to a different location. Now, Git gives you the flexibility of working with multiple remote locations. It's not something we'll explore in this course, but it's something you should be aware of and the reason why Git provides different connections for pushing and pulling changes. So with that said, let's push our changes to the network, but what if we forget to use the ‑u option and then just say git push? Well, Git will decline the push because the remote repository doesn't recognize the current branch because we haven't established that upstream yet, so Git will tell us to do the same thing that GitHub instructed us to do. We need to run git push ‑u and then origin main. Now at this point, Git may prompt you for your GitHub credentials to push to the GitHub repository. Now, these credentials are your GitHub username and the password is what's called a PAT, or personal access token. Now, if you're having issues, Git will provide some helpful information, or a quick search in the GitHub docs will help you create a PAT and add it in the command‑line prompt when it asks you for it. But again, this isn't a GitHub course, so we're just going to focus on the Git‑specific operations here. So once you have authenticated with the network, you'll see that Git output some interesting data back to you. We see a lot of numbers and percentages, well, and this is just related to the specific commits that we pushed up to the network. What's important for now is seeing that we completed 100% of our objects and that we didn't get an error back in return. If the output you get when pushing to the network looks really different than what you see here, Git should provide some helpful information on what went wrong and how to troubleshoot your situation, but here we see we pushed our local main branch and all of the commits on that branch to the remote version of that branch so that they are perfectly in sync with each other. And to confirm, let's navigate back to our GitHub repository in the browser and refresh the page. We should now see our project contents pushed up to the remote network for our team members to see. And nice. We have now pushed up our changes to the network.

### Cloning a Git Repository on the Network

Now that we have our Git repository up on the network, we can now invite other team members to collaborate with us in our project, but before they can collaborate, they need a copy of our project on their own local computer. Now, in Steve's situation, they could just reach out to him and have him send them a copy of his coffee shop recipe project to each person that he wants to collaborate with, but by having this centralized Git repository on the network, each collaborator can just copy Steve's project from the network and then they can work on the project locally just like we do and be able to track changes with Git and then push up their changes to the network for others to see, so let's see how this would work. Navigating back to Steve's repository on GitHub, all a team member needs to do is to navigate to the repository on GitHub and select the Code tab and then copy the GitHub URL that is provided. This is the same URL that we use to create a remote to origin to establish that connection to the network when we pushed our local project to GitHub for the first time. By copying this URL, each member can run a simple Git command that will copy this project, and at the same time, create that remote origin connection back to this Git repository on GitHub. Now, since the project is on GitHub, anyone who copies this project will automatically have that remote connection created, so let's navigate back to the command line and see how we would do this. Make sure that you have navigated to the correct location before you copy down the Git repository because where you are currently at is where the Git repository will be copied, and all we have to do is run git clone and then paste in the URL that we copied from GitHub. Git will then tell us that it's cloning the Git repository into the current directory, and then Git will return some data telling us that the clone was either successful with 100% of the objects being cloned or copied to our local Git repository. And if this looks familiar, it should because it's very similar to the output that Git provides each time we interact with the Git repository on the network. Okay. The Git output looks like it worked, but let's CD into our newly cloned Git repository by typing ls to see the new repository and then cd into our project. We can now type ls to see the contents of our project, and nice, we see the three recipe folders. And let's check the remote connection to make sure that it worked by typing git remote ‑v to see the remote details. And perfect, we see the two connections to our remote Git repository. So let's now make a new commit on this newly‑cloned Git repository as if it was a new team member collaborating on the project. Let's open this up in VS Code and make a quick commit by making a small change in a file. Let's open up the vanilla‑sweetcream file under the iced‑beverages folder and edit the list of ingredients to say 8 ounces of Cold Brew, instead of 8 ounces of Breakfast Blend. Let's save this file and run git status in our command prompt to see the change now residing in our working directory. Let's now add this file change to our staging area with git add, and then let's commit this file with git commit ‑m with a commit message of Change breakfast blend to cold brew. And after running this commit, let's now run git status again to see what our project status is. We now see some different information. Now that we have a remote repository linked to our local repository, Git is telling us that we are now one commit ahead on our local main branch than what we have on our remote main branch. Now, it's important to note, again, that this commit is still local and only visible to the team member at this point. In order for others to see it, we need to push this commit to the network and we can do that by running git push. We don't need to include the ‑u option here because that upstream link to the branch was already created when we cloned the repository, and after Git has returned a successful push, we can now return back to GitHub to see our new commit on the network, and we can navigate to the exact file we made the change in and confirm the change we made is now reflecting on the remote repository.

### Fetching and Pulling Commits From the Network

Now that we've seen how a team member can clone down the Git repository from the network to their own local computer, make a change to a file in their local copy of their repository, and then push that change in the form of a commit back up to the Git repository on the network, it's time for Steve to retrieve the latest changes from the network so that his local copy reflects the changes that the team member just made and that his local Git repository is up to date with the Git repository on the network and in sync with the latest changes. So Steve can do this by pulling down the latest changes from the network using the command git pull. The git pull command will automatically retrieve any commits from the network that your local Git repository doesn't have and then incorporate them into the local versions of your branches on your local Git repository. This is the most common way to get the latest changes pushed on the network because it automatically pulls them down and then merges those changes into your local copy of the Git repository to keep everything up to date and in sync. So when you run git pull, Git is essentially running two Git commands, one command to get the changes from the network and another command to merge those changes into your repository, but what if you only wanted to see the changes first before having Git automatically merge them into your local repository? Well, this is where git fetch comes in. Git fetch is the first command that is run with the git pull. Git fetch will retrieve those changes from the network, but it doesn't merge them into your repository. So let's see how this would work on Steve's repository. When Steve runs git fetch and then git status, we see that our main branch is behind the origin main branch by one commit and that the change can be fast forwarded. Git then tells us that we can run git pull to merge this commit and update our local branch, but before we do that, let's see what this change is actually doing. We can see this change from git fetch by using the familiar git diff command. Remember, we use the git diff command to show us the changes in our working and staging areas, but we can also use it to show the differences between a local branch and the corresponding remote branch on the network. So let's run git diff and then the name of the local branch, which is main, and then origin/ and then the name of the remote branch, which is also main. And running this command, we can view the change that was pushed to the network before we automatically merge it into our local branch. Of course, we saw this change when viewing it on GitHub and that is what's nice when using a code hosting platform. You can easily view the change on the network and know exactly what you want to pull down, but git fetch allows you to view it locally in your command prompt. Now, since Steve wants this change, we can run git pull and it will merge that change into our local branch, and we can see the git output confirming the file update. We can then run git status, and we should see that we are now up to date with origin main.

### Introduction to Merge Strategies

When working with different branches, eventually you'll need to merge them together to bring commits from one branch into another one. Now, we've actually done this a few times already, and even if you haven't recognized it yet, we have done two different types of merges with Git, and this all depends on what the commit history looks like when we tell Git to merge the branches together. Now, there are different types of merge strategies that you can do with Git, and depending on how you want your commit history to look can depend on what type of merge you ask it to perform, but sometimes this takes planning and some merges rewrite history to make it cleaner, but can also be a problem if you're working with other team members who share the same commit history. So we won't be diving into all of these different merge strategies in this course because some are pretty complex and allow you to redesign history, which takes a good understanding of Git that is beyond the scope of this course. So let's talk about the two main merge types that you'll run into when working with Git, and then for fun, I'll show you what an advanced merge strategy looks like and why it can be useful, but also how it can create problems if you don't plan for it. The first merge type is called a fast‑forward merge. Now this is the most basic and straightforward type of merge and can only be done if you have a linear commit path between the two branches. So let's visualize what this looks like. Here, I have my main branch, so let's make a few commits and let's now create a new branch called feature and check out to it. Let's now create a couple commits on our new feature branch. Our feature branch is now ahead of our main branch by two commits, but we haven't made any other commits on our main branch to create a true diverting history. So right now, the commit history between these two branches is linear. So let's say we're done, and we want to merge our feature branch into our main branch. We can check out to our main branch and run git merge feature. This merge has taken the main branch and moving it forward to the timeline of our feature branch. So in other words, our main branch has fast forwarded to our feature branch. Now again, this only works if the two branches we are merging have a linear history, but in most cases, this isn't the case, and we have a true diverting history between the two branches. So let's back up before this merge and create a diverting history between the two branches. Let's create a couple of commits on our main branch. Now, we have a true diverting history between our main and feature branches. Now, when there is not a linear path between the two branches, Git has no choice, but to combine them using a three‑way merge. This three‑way merge gets its name because it uses three commits to then create a new merge commit in our commit history. These three commits are the last commits on each branch and then the first common ancestor commit that the two branches share. So now if we run git merge, we see that Git creates a new merge commit on our timeline that brings the two branch histories together. These two merge strategies, the fast‑forward merge and this three‑way merge, are the two types of merges that the average Git user will do probably 95% of the time, but I did mention that there are other ways to merge. So let's see a more advanced style of merge that is common to some experienced Git users. This merge is called Git Rebase, and it allows you to take a diverting history that exists between two branches and create a linear history, instead of a three‑way merge. Let's see how this would work. I'll undo our three‑way merge and check out to our feature branch and then run git rebase. Now there is a lot going on with this type of merge, so I'll run it again a couple of times. What Git is doing is taking the commits on feature and bringing them into the main branch to create a linear history. Now, on the surface, this just looks like Git took the commits on feature and moved them to the main branch, and in theory, yes, that is what Git did, but Git did some adjustments in order for this to work. Notice that the commit IDs for the old commits are not the same as the new commits we moved over with rebase. There is a lot that goes into the commit ID, including the exact location that it was created, so moving a commit will change the commit ID and how Git references that change. At this point, we can run a fast‑forward merge on main and our two branches will now be up to date and in sync with each other. So why go through the hassle of rebase, instead of just a three‑way merge? Well, there are several reasons, but the common ones are that with a clean linear commit history, it's easier to view the commit logs and then troubleshoot issues. It's also easier to view troubleshooting commands like git bisect, among others. But I mentioned that git rebase comes with a warning because this command alters historical commits, and you should only run this command on local commits that you haven't yet pushed up to the network that others have in their local repository. So if you haven't shared your local commits yet, you're free to alter, remove, delete, or whatever until you have what you want and are ready to share to the network, but when you share them with others, you need to be cautious on altering shared history without a plan, and failing to do so will create conflicts within the project's commit history and resolving them isn't fun and can take some advanced Git understanding to fix them, but this is a great transition to introduce our next topic of merge conflicts. These aren't conflicts that occur by altering the commit history, but conflicts that occur when you try to merge two branches together that have conflicting changes between the two. Now, this can be common when working with others in a shared repository, but resolving them with Git may be easier than you think to resolve.

## Merge Conflicts: What Are They and How Can You Resolve Them?

### Understanding Merge Conflicts

Unless you're working on a personal project by yourself, you'll be working with others in a shared Git repository with dozens, hundreds, or even thousands of other people contributing to the same project, and if your project is open source, then it's open for the world for contributions. Now working with others is how best ideas are grown and where innovation happens, but with all of the great contributions that come from a team of people working on the same project, it's very likely that, at some point, you'll experience a merge conflict. Now, there are some best practices that you can do to reduce the number of merge conflicts that you and your team experience, and we'll discuss these best practices in just a minute, but even if you're following these general guidelines, you can and probably will still experience a merge conflict. So what is a merge conflict and how do they happen? Well, there are several situations that can result in a merge conflict, but basically, a merge conflict occurs when there are conflicting changes being introduced to the same contents in the same file. Now this means that one person has made changes to a file and pushed it up to a centralized Git repository on the network, and then you make a different change to the same contents in the file and then you try to push up that change to the centralized Git repository on the network. Git then becomes confused on which change to accept moving forward. Now, there are several other situations where a merge conflict can occur, but here are the three most common situations that if you do experience a conflict, chances are it's because of one of these three. The first one is when more than one person changes the same line in a file and both try to merge the changes to the same branch. The second conflict is when someone deletes contents in a file, but another person edits the same contents and then they both try to merge their changes to the same branch. And the third situation is when someone deletes the entire file when someone else edits the contents and then they both try to merge their changes to the same branch. With each of these situations, Git is unsure which change to apply, so it notifies the second person trying to merge that there is a merge conflict and It needs help to know which change to accept. Now this requires the second person to manually tell Git which changes to accept and then which ones to disregard. So let's see how this would work by creating a merge conflict on Steve's Git repository and see how Steve can go through the steps to resolve the merge conflict.

### Creating a Merge Conflict

Up to this point in Steve's project, the last thing Steve did was pull down the latest changes from the central repository on GitHub to his local Git repository. So at this point, he is caught up and in sync with the latest changes, but let's switch back to Steve's team member and have them make another local commit and then push that change up to GitHub. I'm now currently at the team members command prompt where he has cloned down Steve's repository under a directory he made called Wired Brain Coffee, so let's skip this step to create a new branch and just make a change on the main branch so we'll make sure that we're checked out to main. Let's now open up VS Code and make a change to a file. Let's navigate to the white‑mocha recipe under hot‑beverages, and let's change 1 cup of 2% whole milk to 1 cup of whole milk. Let's make another change to Step 4 and update 2% milk to whole milk to keep the recipe consistent, and we'll make one last change to the description. The team has been discussing ways to sell more food, and this team member has an idea to add food pairing options to the recipe's description. So at the end of the description after creamy beverage that delights with every sip, let's add And pairs perfectly with an almond croissant. Let's now save this file and return back to our command prompt. Let's now run git status to confirm that our change is showing in our working directory and then let's run git add to move the white‑mocha file to our staging area, and then we can run git commit with a message of Update white mocha recipe. We see that Git returns the expected output for our commit, and we can now run git push to send this commit to our central Git repository on GitHub. Perfect. We see Git's output that our commit was successfully pushed to the network. Let's now switch back over to Steve's command prompt and make some changes. Steve also has some updates that he wants to make to the white mocha file, so he decides to create a branch called update‑mocha to make his changes. Let's create this new branch using the git checkout command with the ‑b option and then let's open up VS Code. Let's navigate to the same white‑mocha file under the hot‑beverages folder, and in the list of ingredients, Steve wants to make his recipe show almond milk by default, instead of 2% milk. So Steve updates the list of ingredients to say 1 cup of almond milk, and to follow up this change, Steve also needs to make a change to Step 4 to update this step from 2% milk to almond milk. After these changes, Steve also wants to update the recipe description so he adds the phrase, A staff favorite, at the end and hopes to increase its popularity. Okay, Steve now feels good about these changes. Let's now save this file and return back to the command prompt. Let's now run git status to confirm that these changes are showing in the working directory, and then let's run a git add to move Steve's changes to the white‑mocha file to the staging area, and then run git commit with a message of Make changes to white mocha. Nice. We see that Git returns the expected output for our commit. So this branch was just for Steve to try out new ideas locally before he sends them to the network. He doesn't have any intentions to push this Update‑mocha branch to the centralized Git repository and just wants to merge this local Update‑mocha branch to his local main branch and then just push up his main branch to the central Git repository. Now, he's satisfied with these changes and he's ready to merge them into his local main branch, but before he does that, he wants to make sure his local main branch is up to date with the origin main branch on the centralized Git repository, so he checks out to his main branch and runs the command git pull. We see that the latest changes are pulled down to Steve's local main branch, and in the output, Git tells us the file changes that were made, and look, we see the changes that the team member made on the white‑mocha file with some insertions and deletions. So at this point, Steve sees the update, but he isn't sure yet what exactly those changes are, so he decides to merge his update‑mocha branch into main with the latest changes that he just pulled down. So on his main branch, Steve runs git merge Update‑mocha, and now, Git returns something new. We see that we have a merge conflict.

### Resolving a Merge Conflict

Git is telling us that we have a conflict with the content we are trying to merge into this main branch, and Git can't complete the merge until we resolve this conflict. So let's run git status to see some more information. Git gives us a couple of options. The first is that we can fix these conflicts and then run git commit to confirm the changes, or we can abort the git merge that we are trying to do. Git then tells us the exact file or files that are having the conflict and it has placed them back into our working directory. We see that the white‑mocha file is the conflicting file, so we need to resolve the conflicts in this file and then add the file back to the staging area and commit our fixes to resolve the merge conflict. So let's navigate back to the white‑mocha file in our text editor to resolve these conflicts. And opening up the file, we see some new information. What you see here are merge conflict markers. This is Git's way of helping identify the exact lines in the file that are having the conflict. For every conflicting change, Git will surround that conflict with these markers to make it easier for us to decide on what we want to keep and what we want to remove. The top section is labeled head as the current change. Now remember, head is where we are currently checked out to, and since we are currently on the main branch, this head section is showing the conflicting content on the main branch, and the second section is labeled as the incoming change on the update‑mocha branch that we are trying to merge. These two sections are separated by these series of equal signs in the middle. So at this point, all we need to do is keep what we want and then remove what we don't. So for this first conflict, Steve was reviewing the changes and decided that he really likes the change that the team member made. So he wants to keep the change on head over the change he made on his update‑mocha branch. So Steve can simply remove the incoming change section and remove these conflict markers. Now, this is important to remember when resolving merge conflicts. Don't forget to remove those conflict markers. If you forget to remove them from the file, Git will keep those markers in the file and they will be added when you commit the file and resolve those conflicts. Also, Git doesn't care what you decide to keep or remove. Git just needs you to tell it what to do. So if you keep the current change, the incoming change, or remove them both and add something completely different, it's completely up to you on what to keep, and if you're working in a team environment, you may need to discuss these changes with the team before deciding what to keep and what to remove. Now for these other conflicts, we can just repeat the same steps. For most text editors, you will have an option to just click a button to choose an option, and it will remove everything else. Now, this will look different depending on what text editor you use, but with VS Code, I can just select the option I want. So for these next conflicts, Steve wants his incoming changes to be the change moving forward, so he is going to click on the Incoming Changes button provided to him. And nice. Our editor has removed the other head option and the conflict markers for us. We can now save this file and return back to our command prompt. Let's now run git add to add this file to our staging area, and we can confirm this with git status. Let's now commit this change by typing git commit with a message of Fix merge conflict. Let's now run git status. We see the that we are now two commits ahead of our origin main, and well, this is true. We have made the first commit on our update‑mocha branch, and then after resolving the merge conflicts, we made the merge commit on our main branch that brought the updated changes from the mocha update branch, and we can confirm this by running git log ‑‑oneline, and we can see that our local main branch is two commits ahead of origin main. So, to finish, let's send these latest changes to our centralized Git repository by running the git push command, and Git will output a successful response, and we can run git log ‑‑oneLine again to see now that our local main and origin main are now up to date and in sync with each other.

### Preventing Merge Conflicts

Now, we just went over how to resolve merge conflicts, and hopefully, it doesn't seem too intimidating. Just remember that some conflicts you experience may be a lot bigger than what we just resolved, and you may have several files to resolve, instead of just one, but remember that you resolve all conflicts the same way. You choose which option you want and simply remove the rest or remove both options and add something else entirely. Just remember to remove those merge conflict markers. So knowing how to resolve those conflicts is great, but there are some best practices to follow that will help reduce the number of merge conflicts you and your team experience. One way is to standardize your formatting. A lot of conflicts happen simply because of formatting discrepancies. These are things like extra whitespaces and different coding styles, and you can enforce code formatters and linting rules to make sure that your team is aware and help reduce the number of these types of merge conflicts. Another way is to make small and frequent commits. Taking weeks to make changes in a branch and adding lots of changes to lots of files is almost begging for merge conflicts to happen. It's usually best to make small and direct changes and then merge them frequently into your main branch. Don't wait long periods of time to merge changes if you can help it. Another way is to just communicate and pay attention. Talk to each other and know who is working on what. If multiple people are making changes to the same files, well, work together and communicate often. This is a great way to help reduce the number of merge conflicts within your team.

## How to Modify and Fix Your Commits

### Fix a Commit Mistake with Git Amend

Up to this point, we have made a lot of commits and for good reason. Committing changes is one of the main reasons of using Git. And as we've discussed previously, with each commit, Git creates a unique ID to reference and manage the project snapshot at that specific point in time and then stores this snapshot in the form of a commit in our Git history, and we've seen this history already by reviewing our Git logs and seeing the commits that we've made. These commit IDs are really important and what Git uses to track and manage each commit snapshot, and there are several things that are used to create each commit ID. Git uses the author, the timestamp, the commit message, the actual changes being made, and even the commits that come before it. All of these things and others are used when Git creates the commit ID. So as you can imagine, modifying an existing commit will alter that commit ID, and if there are other commits that use that commit in its history, especially if you have pushed this commit to a centralized Git repository and others are basing their work on this commit, it will create a broken history, and you'll have a mess on your hands and can take some deep Git knowledge to fix it. However, when it comes to changing commit history, the general rule is that if your commits only exist locally and you haven't pushed them to a centralized Git repository that others are using, you are free to alter these commits as much as you want and rewrite those commit IDs. And then when everything looks good, you can push them up for others to use and to see. So let's look at a few examples of why you may want to rewrite your local commits and then how you can do it. To show these examples, we're going to get out of Steve's Wired Brain Coffee Shop project and start a whole new Git repository. If you want to follow along, you can navigate to this GitHub link with instructions on the README page where you can find the initial folder for Steve's project, as well as this bash script. We'll copy the script and return back to our command prompt. Now, make sure to navigate to a directory that you want to create a new Git repository, so I'll create mine in this pluralsight directory and run mkdir to create a new project called practice‑reset. Let's now make this a Git repository by running git init, and check it out. Our global configurations are working again as expected and Git has created our default main branch. Let's now paste in that script that we copied. Now what this script is doing is it's using the touch command to create six different files and then commit them individually. Let's run ls to list the files that we just created. And nice. We have our six different files. So if we run git log ‑‑oneline, we can see these six files all committed separately to give us some commit history to work with. So let's say we actually want seven files, so let's just create that seventh file. We can use the touch command and name the file file7.md. Let's now add this file to our staging area with the commit message of Adding file 8. Nice. We have just added our seventh file, and let's confirm this by running git log ‑‑oneline, and we can see our new commit in our history, but as you probably saw, we committed this file with the commit message of file 8 when it should have been file 7. So to fix this, we can use the git amend command to fix the commit that the head is pointing to. And since this was our last commit, head is pointing to this commit so we can use this command to fix our commit message. So let's run this command to fix our commit message, but before we do that, watch the commit ID as it changes when we complete the fix. In the command prompt, let's type git commit ‑‑amend. This will now open up an editor within our command prompt that we can use to make this change. Now, you can type i for interactive mode where you can then make your adjustments. And then when you're ready to save the changes, you can hit the Esc key and then type :wq to save or write the changes and then quit. Git will then complete this change and send you back to your command prompt with the confirmation output. We see that we changed the commit message, and we have a new commit ID. Let's now run git log ‑‑oneline again to see the commit ID change in our history with the updated commit message. So again, because this commit was our latest commit and it's only a local commit, we haven't pushed this commit up to a central repository. We are free to rewrite our commits. Git amend is a useful git command to know if you need to quickly change something in your latest commit.

### Rewrite History with Git Rest

Now git amend is a useful command, but sometimes we need something a little bit more robust, something that we can use to reset some or even all of our files to look like they were at a different point in history if we decide that things aren't going as planned in a particular branch. For these situations, we can use the git reset command. Before we jump right into using git reset, let's talk through the different options that we can use. When using git reset, Git provides three options that are available. The first is git reset soft. This type of reset will take a commit from our history and place it back into our staging area. This is useful if you need to regroup commits or alter something within the changes being staged for commit. The second is called git reset mixed or just git reset. This is the default option for reset and will take a commit from history and place it in your working directory This is if you want to take those changes all the way back to the drawing board and do some changes before staging them again. And last, we have git reset hard. Now, a bit of a warning here, this type of reset will take a commit from history and throw it away in the trash almost like it didn't happen. This is a destructive command, so make sure you double‑check your work before using this type of reset. We'll go over each type of reset so you can see how it works, and I'll even show you where you can get back those commits that you've thrown away with the git reset hard. So let's start with git reset soft. We can do this by typing git reset ‑‑soft, and then we need to tell Git what we want to reset. We can do this a couple of ways. We can provide the commit ID that we want to reset to, or we can tell head to reset everything back to a specific location. So let's say we want to reset our last two commits. We can say HEAD~2. Now, before we run git status, we should expect to see our last two commits back in our staging area and running git status confirms what we expected to see. Both of the commits that added files 6 and 7 are now back in our staging area, so let's commit these files together in one commit by running git commit ‑m adding files 6 and 7. Let's now run our git log ‑‑oneline command to see our updated commit history. We have removed the previous two commits and have added a new commit that adds both files 6 and 7. Notice we have a new commit ID, but the older commit IDs stay the same. Let's now try git reset mixed or just git reset. This will send changes back to our working directory and will allow us to restage different parts of a commit. So let's say we want to go back and recommit our files 6 and 7 separately, instead of together. We can do this with just a git reset and then tell HEAD to move back one commit. This should move our latest commit with both files back to our working directory. Let's now run git status, and sure enough, we can re‑add these files back into our staging area. So let's run git add to file6 and commit the file with the message of Adding file6 and then let's run git add on file7 and commit the file with the message of Adding file7. Let's now run git log ‑‑oneline again, and we should see new separate commits for adding files 6 and 7. Nice. This is looking great. The last is a hard reset, and this will take a commit or commits from history and throw them away, not in staging, not in working, but gone. Again, be careful when running this command because it can be destructive. If you have changes in the staging area or in the working directory and you run a git reset hard on a commit, not only will it do a hard reset on that commit, but it will also erase everything you have in your staging area and working directory. So again, be careful when running this command. But with that said, let's see how this would work. Let's say that we decided that we no longer want the last three commits and we want head to point to the commit that adds file4. We don't want to restage these commits, we just want them gone. So let's run git reset, and then instead of moving head back, we can just provide their abbreviated commit ID to the commit that we want. So we can type git reset ‑‑hard and then let's copy the commit ID that we want. Notice that Git now tells us that head is now pointing to our desired commit and running git log ‑‑oneline, we see that our history no longer has files 5, 6, or 7, and running git status, we see that our staging area and working directory are clean.

### Get Back Deleted History with Git Reflog

We just deleted several commits from our history that added files 5, 6, and 7, but after thinking about it, we decided we made a big mistake. We don't want files 6 or 7, but we really want file5 back, and because we used a hard reset, it's not on our staging or working directory to simply add back and commit. Well, Git provides a way for us to retrieve commits, even if we've thrown them away using a git reset. Similar to git log that tracks our commits, Git has a log that tracks everywhere that head has been, and it's called the git reflog. We can use the reflog to restore previously committed changes, but first, we need to be aware of some of reflog's limitations. The first is that the reflog is only local. It is not pushed to the remote Git repository and only includes your local history. In other words, you can't see the reflog for someone else's commits on their computer and they can't see yours. And second, the reflog is a limited time offer. By default, these commits are only displayed in the reflog for up to 90 days, and after the 90 days, these commits are completely gone and you can't get them back, but lucky for us, we are well within our 90‑day period, so let's restore our commit for file5. We can do this by running git reflog and Git will output a similar log history that we see with gitlog ‑‑oneline. This is a history of where head has been, so we can find where we've added file5 and then copy that commit ID. At this point, take notice of the commit ID that we are copying. Now that we have our commit ID copied, we don't add this commit to our working directory or staging area. We add it right back into our commit history. We can do this with a new command called git cherry‑pick. This Git command allows us to take a commit from the reflog and place it back into our commit history. So let's run git cherry‑pick and then paste in the commit ID. We see some Git output that looks like we committed that change back into our commit history. So let's run git log ‑‑oneline one more time to see our commit history. We now see our latest commit adding file5 back into our history, and look, the commit ID that we copied is not the same ID in our commit log. Again, there is a lot that goes into this commit ID, so by altering the timestamp among other factors, the commit ID has been altered. Now, this is just an introduction to what you can do with git amend, git reset, and git cherry‑pick. There are lots of other ways to use these commands and even use them together with merge strategies like git rebase. You can restructure your local history before pushing your commits to the remote repository on the centralized server. Now I've been using Git for almost a decade, and I don't use these commands a lot, but when I do, it's really nice to know how they work and what I can do with them. So as you learn more about Git, a good understanding of some of these commands like rebase and reset will help you better manage, plan, and structure your Git projects.

## Popular Team Workflows

### Introduction to Git Workflows

At this point, we've gone over the main concept and fundamentals of Git, and we've even taken a few minutes to show you what you can do with some more advanced features like rebase and reset. These concepts and fundamentals will remain true in every Git project that you work in. Now it doesn't matter if you're working in a personal project alone, only on your local machine, or if you're contributing to a centralized Git repository on the network with thousands of other contributors. The process of adding files from the working directory to the staging area and then committing those changes is a fundamental concept of how Git works. However, what can change is the workflows and how you incorporate your changes into the Git repository. And depending on the needs of the project, these workflows can be simple or they can be a little bit more involved. Now, you may not have realized it, but we've actually learned the fundamentals of Git by using a couple of different workflows. Also, depending on the code hosting platform that you're using, there are collaboration features like pull requests that you can incorporate into your Git workflow that will improve the way that you review, discuss, and secure your Git projects, but let's take a look at some of the most common workflows that you'll experience when working in a Git project.

### Basic Workflow

The most simple way to use Git is by using a basic workflow. This workflow uses the default main branch as the only branch that contributors use to make their changes. So using a centralized Git repository on the network, each contributor can clone down the repository, work locally on the project, and then push up their commits to the central repository all on the main branch. This workflow works in small projects with just a few contributors, something like a simple website, but it can quickly become problematic if more than one contributor wants to work on different features inside the project. Having everyone contribute to the main branch is a great way to encounter a lot of merge conflicts, and this workflow doesn't provide a lot of feature building without affecting the production‑ready project on the main branch. Now, I use this type of workflow a lot with my personal projects where I am the only person contributing and the project is something simple. We saw this work well in Steve's coffee shop recipe project at first, but with the addition of multiple contributors, it's best to move to a workflow that better supports diverting history, something like a featured branch workflow.

### Feature Branch Workflow

A feature branch workflow takes the basic workflow, but instead of committing directly to the main branch, contributors create feature branches off of main when they want to work on new features or try out new ideas. This allows for more flexibility when working in a team environment, and most notably, it allows for team members to better review code changes in feature branches before they are merged into the main branch. This is a good workflow if you want to work with feature branches, but still only need to maintain a single version of the project in a production environment. With a feature branch workflow, anything in the main branch is deployable. If you want to work on a new feature, you can create a new feature branch off of main, and you can also create feature branches off of other feature branches. This isn't a complete list, but here are some advantages and disadvantages to using this type of workflow. Let's start with the advantages. This workflow offers feature branching, but it's not too complex. It's a friendly workflow for continuous integration and continuous delivery, and it's a good option if you only need to maintain a single version in production. Now the disadvantages with this workflow is that with these feature branches merging back into main, the only production version of the project, it's easy for errors to get pushed up to production. It's not a good option if you need release branches or multiple versions of the project in a production environment. So if you want the addition of feature branches, but you still only need one version of the project deployable to production, then this workflow could be a good option. But if you want these features, but also need to better secure the main branch with release and develop branches, then you'll probably want the next workflow which is called the git flow.

### Git Flow

The git flow takes the feature branch workflow and adds a few things to it. This is the most common workflow that you'll come across when working with Git repositories in larger teams. Now, as I mentioned, it's similar to the feature branch workflow, but the main difference is that contributors branch off of a develop branch and not the main branch. This develop branch is a pre‑production environment, so changes merged from feature branches can be reviewed and tested in this preproduction state before merging into the main branch, which is the production version of the project. This adds that additional level of review and testing that helps eliminate errors and buggy code from getting pushed into the main branch. With this type of workflow, you can have other supporting branches, such as release branches and bug fixes, but with every workflow, there are some advantages and disadvantages, so let's start by looking at the advantages. Git flow is very popular, especially in the open source community. Project maintainers can better review and approve code going into release or develop branches before getting merged into main. The git flow works well with established products and multiple versions of the project. It adds that extra layer of review so it greatly reduces the amount of errors going into production, but some disadvantages are that sometimes this workflow can slow things down. It can take a while to get reviewed and pushed to production, and going through all of these different branches, it can also be confusing for new team members to contribute to a workflow and know all of the different steps to merge their code. Another thing to mention is that with all of these extra layers of merging branches, you can run into merge conflicts more often. Of course, how these workflows are documented and communicated across the team makes all of the difference when working in a large project environment. Now, these are not the only options when working with Git. There are several other types of workflows that you can try out and see what works best for your Git project, and the advantages and disadvantages are definitely not a complete list. So when you get to the point of needing to pick a workflow, I recommend that you start with the basic workflow and then work up from there. There's no need to overcomplicate your workflow if a more simpler workflow will do the job, and if you're using a code hosting platform like GitHub, GitLab, or BitBucket, you can use their collaboration features that give you that extra level of review and security for your project while maintaining a noncomplex workflow. So again, my recommendation is to start basic and then add additional steps as your project requires them.