## The Case for Git with a GUI: Minimizing the Learning Curve

### Using Git with a GUI

Hello, and welcome to this Pluralsight training course, Using Git with a GUI. I am Xavier Morera, and I'm very passionate about teaching. If you're here, I assume that you already know what Git is. If you don't know what Git is, it's a widely‑used source code management system for software development. Something else that Git is, or at least some might say, is that Git is hard, and you may agree or disagree, but what I can tell you is that sometimes it feels that Git is hard because of how powerful it is. The good news is that the whole point of this training is to help you either get started or become proficient when using Git. Having said that, let's start with the case for Git with a GUI, minimizing the learning curve, which means it's time to go to the recording booth and create a Git course.

### Why Git?

Let's get to the point. Why Git, and I know this might be a little bit subjective, but as I mentioned earlier, I think that Git is the best version control system for software development, but it's not only me, millions of other developers think in the same way. And there are many reasons among which you can find the fact that Git is distributed, which means that each person working on the code gets their own local copy, they can work and upload their work when ready. For example, you can work in a plane, then it has integrity and security, meaning it guarantees that whatever you add will be retrieved, checking for data corruption using cryptographic functions, as well as having access control, making it very secure. Then Git is fast, very fast. There are several reasons, including the fact that many operations are carried out locally. Git also uses many advanced algorithms to compress data. Additionally, it provides one of the best, if not the best, branching and merging capabilities out there, enabling collaboration and allowing teams to work in parallel easily. In fact, its branching capabilities are one of its killer features, one of its most powerful added values over other source control systems, and it's widely‑used by many companies out there. It's definitely a must in your skillset, but maybe you're not yet convinced. Well, maybe you've heard about GitHub, which as of now is the most popular Git hosting platform for open source, along with BitBucket, which provides a great user experience for Git users in the cloud, and there are others like GitLab. Also, you can even store your own bare repositories, but the point is that there are way, way too many interesting projects that are hosted out there, some of them are open source, while others are not. Just to give you an idea, here are a few cool projects that are hosted in GitHub. There's a whole bunch of interesting projects. I just picked a few popular and important ones to show you, and this is not only it, the growth has been staggering. GitHub was founded in 2008. By 2013, there were 10 million repositories that were created, but check this out. The first million repositories took almost 4 years, but just in 1 year, 2013, almost 5.5 million repositories were created, so about half in a single year. Five years later, the number of repositories reached 100 million, and they kept growing, and growth has not slowed down. By 2023, the number is reaching 400 million repos, but this is not a GitHub training, nor BitBucket, nor GitLab. This is a Using Git with a GUI training. However, I just wanted to make sure that you're convinced that knowing Git is a must‑have skillset. A large number of companies out there require that you know Git for you to be able to work with their code. So now, let's talk about why it may be a little bit hard to learn Git.

### Why Is It Hard to Learn Git

Perhaps someone told you, perhaps it is what you already think, or maybe you just read it somewhere, Git is hard. How hard? Well, there's even a joke going around the internet where one person that's using Git has a merge conflict and asks a more experienced developer how to fix the error. And the senior developer basically says, it's simple, just get a fresh copy of the code, make the changes again, and delete the old code with the conflict. Okay. First of all, this is a joke, so don't take it seriously, but I actually find it funny because it's true in some cases. I've done it once or twice, but jokes aside, why is it hard to learn Git? Learning Git is one challenge that many developers face, even experienced ones, and I'm not saying everyone, but many do. It is a possibility that Git can give you a headache or two if you're not sure of which steps to follow or which commands to use, which is what I will focus on this training today. And let me tell you a few reasons why some people find it hard to learn Git. First of all, Git is distributed, which is also one of its advantages. Other source control systems are centralized. In a distributed system like Git, each developer has a copy of the entire code base. They can work and do all kinds of things on their local environments, which is one of the reasons that makes Git so flexible. There are a lot of different ways of using Git when compared to other source control systems. Also, there are many different workflows. For example, you can have a main branch and a development branch, or each feature can go in its own branch and you merge when ready. There are many, many options, which means that Git supports a nonlinear history. You can create multiple branches and merge them in a non‑linear fashion. This flexibility can increase complexity. And on top of all this, some of Git's terminology is not what typically developers would encounter, for example, branch, merge, commit, stash, cherry pick, rebase, conflict, fast forward merge, and more. Oh, and there's the tooling. Git was created by Linus Torvalds, the creator of Linux. As most Linux developers, they tend to work with a command line which can be very powerful, but can make things a bit more complex than what they can be, in some cases. You can basically do everything with the command line, but you may face a somewhat steep learning curve, and just to give you some peace of mind in the fact that you're not alone, listen to this, and I quote, "Early versions of Git did require a certain amount of brain power to wrap your mind around," and you know who said this, Linus Torvalds, the creator of both Linux and Git. And while some things have changed and Git usage is more common now, some people still agree that even the current version still needs a little bit of brain power. So when it comes to Git, don't get scared, get started, which takes me to a demo of Git with the command line.

### Git from the Command Line

In this training, I assume that you already know how to use Git, be it from the command line or from the GUI. So what I'm going to do now is I'm going to show you a demo of it from the command line, which later on, I'm going to show you how much easier it is to work with a GUI. Here I am in my machine in an empty folder. I'll start by creating a new folder, which is where I'm going to create my repository. I'm going to call it python‑demo. I'll change into this folder and clear the screen. As you can see, it's just an empty and regular folder, and this is the command used to initialize a repository, an empty repository, in this case. Basically what it does is it creates this folder .git, which if I run there, I will not be able to see it because it's a hidden folder. I'll use that /a command to show it. It's right here. Okay, let's move forward. Let's check the status of this repository. This is the command that you use to display the state of the working directory and the staging area of your repository. I have not made any changes yet, so it's clean. So let me open these two folders. On the left, I have three files that I have prepared for this demo. On the right, it's the Python sample folder. Let me copy platform.pi into the Python demo folder. I paste it right here, and now when I move back to the command line, I can see the file right here. Now, if I run status, I can see that this file is an untracked file. That means that it has not been added yet to Git, which I can do by running git add platform.py. Now, if I run status again, I can see that this change is ready to be committed. Platform.py is in the staging area, but as you can see above, I have not made any commits yet on this new repository. Now to commit, I use the following command, git commit ‑m, that's the message, and adds platform.py. It's always a good practice to add a message to your commit that reads like this. This commit Adds platform.py. It makes it simpler for other people to understand what each commit does. Add good commit message is always a great idea. Let's see. One file change, 26 insertions, that's the number of lines, create mode. All went well. Now if I run status, I can see that there's nothing to commit. The working tree is clean, and that's the branch where I'm currently working on. If I use git log, I can see the history of my commits so I did a commit, and here's the hash which identifies the commit. Now, I go back and make sure that the files are there and I run git status which tells me that there are two untracked files. Now, I can add both of them using it ad dot This stages, all files, they are ready to be committed, hit, commit minus M and the commit was successful. Now, as you can see all files have been committed, the working tree is clean. Now, let me show you something that when we get to using git with a GUI, you'll see how much easier it is versus using the command line. I'm going to open one of these files platform dot pi and I'm going to make two separate changes, going to change the name of the directory and this message right here. This means that I made two changes. It's actually three lines, but the ones at the top are next to each other which are grouped into hunks more of that to come. Now, I save I close. And now when I run status, it tells me that one file has been modified. But remember that we made two changes. So how do I take a look at which changes did I made for this? I use it. This which shows me the changes at the top those two lines and the change at the bottom. So here's a question. Do I have to commit the entire file? Well, actually, as you probably know the answer is no, you can commit only one of those changes for which I can use it. Add minus P platform dot pi which starts the interactive mode. Here it asks me if I want to stage each one of the hunks individually. So I say yes to staging this hunk, which means it will be added to the staging area. But I don't want to stage a change in the print. But I say no. Now when I run git status, it tells me that there are some changes that are ready to be committed but others that are not yet in the staging area for which I can check with git TIFF, the print will not be committed. Now I commit the update to the paths and I still have this change which I don't need it anymore. So I'm going to run restore platform dot pi and the change has been discarded and then I can keep using it for other functionality. For example, I can check in which branch I am right now and I can create a new branch. And there it is, I just created a new development branch. And this was a short demo on how to use git from the command line. In just a few minutes, I will show you a demonstration or from the GUI, which will show you how much easier it is to work with a GUI versus the command line.

### Using Git with a GUI

So far, I've been talking about a GUI assuming you know what a GUI is, and I'm pretty sure you do know, but let me talk about what is a GUI and why a GUI to make a point. GUI, which stands for graphical user interface, is defined as a type of user interface that allows users to interact with digital devices, software, and applications using graphical elements, such as icons, menus, buttons, and Windows rather than text‑based interfaces. This is the definition as taken directly from Wikipedia. GUIs were introduced in reaction to the perceived steep learning curve of CLIs, the command line interface, which required commands to be typed. This whole story is fascinating, but I'll leave that for another day. In a nutshell, it is quite simple. If you're using a mouse or your fingers and looking at a window, instead of typing in the command line, well, you're pretty much using a GUI. It is something you most likely do a lot every day. At a high level, there are several advantages of using a GUI. First, by providing a visual representation of data and information, it is easier to understand and analyze complex information. How does this relate to Git? Well, you get a visual representation of Git workflows, understanding the different branches, commits, and history is way easier with a GUI, which talking about easier, using the GUI of an application makes it easy for you to interact with the system. Click here, click there, and you're done. Now with Git, in the GUI, you can access all the Git commands. Even better, a click here may mean that many commands are executed in the order that it's required with the correct parameters, which provides consistency. Things are done in the correct way in the same way every time by the application that you're using, which relating to Git, this consistency means that the application that you're using through its GUI can integrate with other tools, for example, GitHub or Bitbucket, which takes me to my next point, speed. Using a GUI is typically faster than typing. If you're working, modifying a lot of files, you may need to do a lot of typing in the command line. This speed allows you to have improved productivity. You can work faster and get more stuff done. Now, here's an age old question, which application with a GUI that works with Git should I use? Which one is the recommended one? And the answer which is typical in the programming world is it depends. There are a lot of factors to take into account, including personal preferences, company policies, licensing, and more, but there are three options that I use all the time that are great. You cannot go wrong with these three. There are way, way more options, but these three work great. I am talking about GitHub Desktop, which comes directly from GitHub. Then Atlassian SourceTree. Bitbucket is a product of Atlassian. And another one that I use frequently, which is not a GUI for working with Git repos, it's actually a code editor, Visual Studio Code, which integrates nicely with GitHub, Bitbucket, GitLab, and other hosting platforms. And in this training, I am going to focus on Atlassian SourceTree. In fact, if you look closely at this screen, I use Git to create this very own Pluralsight training, and it works wonderfully. And now, let's do the takeaway for this module, and then afterwards, we can begin with the action.

### Takeaway

As a takeaway for this module, Git is a version control system that the opinion of some people is that it's a little bit hard to learn. So why should you learn Git? Well, for one, it's very powerful, it can be very useful, it can really help you, and also it's extremely popular and widely used. How much? Well it's used by thousands of companies and millions of developers with many hosting platforms like GitHub, GitLab, and Bitbucket. Some of the features that make it so powerful is that it's distributed, it has integrity, security, great performance, and world class branching and merging capabilities, which again, it makes Git a little bit harder to learn when compared to other version control systems. However, I don't want to scare you. Learning Git is not that hard if you have the right resources. Yet still, one of the reasons that I found why some people say that it's a little bit hard is because they use the command line, which increases complexity. In my humble opinion, it's better to use a GUI because it provides a visual representation which gives you ease of use, integration with multiple tools and hosting platforms, which in turn helps you improve your productivity. And now please join me in the next module.

## What You Need to Know About Git

### What You Need to Know About Git

Now that I, hopefully, made it clear why you should use Git and why it is better with a GUI, Attlassian SourceTree being the one used in this training, it's time to start with this module, What You Need to Know About Git, and I'll start with the story of Git, but it's an interesting story after all, and I promise, I'll keep it short. So here's a question for you. Do you know how Linux, which is one of the biggest and most successful open source projects, was originally maintained? Well, basically using tarballs and patches. This is just moving files around. It's not ideal, right, but then it evolved into using BitKeeper, which is a source control system that's distributed, and here's the key, it's proprietary. It had a free version that was used by the Linux community, but in 2005, it stopped being free. And the legend says that Linus Torvalds, the creator of Linux, basically said, "We will stop work until we have the source control that we need," and then he created Git in what some people say it was over a weekend. Other people from the Linux community have helped since then, but he was the one that started it, and that's pretty much it. Since then, Git has risen to the top, it is widely used and many tools and hosting platforms have been developed. And now, let's do a Git overview.

### Git Overview

I know I mentioned some of this in a previous module, but let's make sure we are on the same page. Let's do an overview of Git. Git is a source control system that is very powerful and flexible. Why, you may ask. What makes it so powerful and flexible? Well, it is distributed, which is something that sets it apart. In a distributed control system, each developer gets an entire local copy of the code, including the complete history. This is how you typically work with Git. However, if you're working with a repository that's way too big, it is possible to use a partial clone and a shallow clone, a shallow clone being a ripple with its history truncated and a partial clone is a type of shallow clone that only downloads a subset of the objects in the repository that only downloads a subset of the objects in the repository, but these are out of the scope of this training, which is why I'm only going to cover a full clone. And then with Git, you perform local operations for most actions, which means that you can work disconnected. Let me explain myself better. Most people are used to something like this where we have an origin and a local repository. They get a file that they will be working on, and in some cases, the file is locked. They modify it and then upload the change, but with Git, you can have multiple remote and local repositories for multiple teams or individuals to collaborate, and you can keep adding and collaborating. How you collaborate is where workflows come into play, something we will see soon. Moving forward, in Git, you usually only add data. You don't modify data that you have already committed. You can, but it's not usual. The adding data is what's called a commit. A commit being at the core of the Git object model. Quick side note, on commits, there's a wise best practice around Git. You commit early, you commit often, and then perfect later. Have you ever been in a situation where you have spent many, many days working on your code, and when it comes to committing or checking in, you already forgot the context around some of the changes that you did? This would be probably an extreme case, but it's something that happens from time to time, so remember commit early, commit often, perfect later. Anyway, Git is snapshot‑oriented versus file‑oriented. What does this mean? Git tracks and stores the entire state of a project at a given time as a snapshot, instead of tracking individual files, which has several advantages, including the fact that it has integrity. Every commit is identified by a hash which prevents tampering of the data. All the repositories used SHA‑1, however newer ones use SHA‑256, and because it's working with snapshots, operations are really, really fast. Also, files that have the same content are only stored once. Git is pretty smart about repeated data. Let me expand on what a commit is. This is the history of commits that I've made in my repository for this course. Git gives me all the information that I need. Of course, with a GUI, it would look a lot better and easier to read, which is kind of the point of this training. Anyway, each commit is identified by a unique hash which is created based on the content of the commit, including the changes made after timestamp and other related information. Since the hash is based on the commits contents, any change that commits data will result in a different hash. The hash looks like this, a very long string, 20 bytes or 40 hexadecimal digits, and it is a reliable way to reference and identify commits. They are unique. There is a really slim chance of getting a duplicate hash, it is used for many operations. Oh, and by the way, only the first seven or eight characters should be enough for it to be unique, then the author which includes name and email of the person that did the commit. Also, Git includes the committer, which in many cases, it is the same person. However, in certain scenarios, like applying patches or rebasing, they can differ. The parent commit, which is used by Git to trace the history of changes to the repository. The only exception is the very first commit, which is the start of the history line. The timestamp, which is when the commit was made, and the message, which is useful to explain the purpose of each commit. In just a few minutes, I will cover other details regarding how to create a commit and everything related, for example, the index, staging, and more. The commit is one of the four types of objects that are stored in Git, which in a nutshell, a commit represents a single logical unit of work with this corresponding snapshot of the state of the repository at that point in time. The other types of objects are the blob, which is an object that represents the contents of a file, it contains the file's data, such as text, binary code, or any other type of content. Each file in a Git repository is stored as a separate blob object, and it's immutable, meaning that they cannot be changed once they have been created. If the content of a file is modified, a new blob is created to represent the updated version of the file. Then the tree, which represents the structure of a directory. It's kind of like a folder, it functions as a container for blobs which represents files and other trees, think like subdirectories. And finally, the annotated tags. In summary, a tag is kind of like putting a label on a commit. Then there's a special kind of tag called an annotated tag that includes additional information like the author. The Git object model is designed to be immutable, meaning that once an object is created cannot be modified. This is one of the key features of Git. The way it works is that Git relies on creating new objects when changes are made and referencing them accordingly. This design allows Git to provide efficient versioning and branching capabilities, which takes me to my next topic, branches. Each commit that you make references its parent commit, although to be fair, a commit can have multiple parents. Well, when you have multiple commits, each one is pointing to its parent which gives you a commit history. The commit history refers to nothing more than the recorded sequence of commits in a repository. The history provides a record of changes made through time, and you can view these changes, revert a change, for example, change a file to a previous state and allows for collaboration as developers can understand the context behind changes and apply changes to their code base as needed. In a nutshell, Git can track the complete history of changes in a repository which provides powerful features, such as branching and merging. Now what's called the killer feature of Git, branches. Branches are nothing more than pointers to specific commits. A branch always points to the latest commit in a particular history, and the branch pointer moves forward on new commits. A branch represents an independent line of development in a repository. Branches allow for isolating work and creating separate lines of development, including the ability to develop in parallel. Once ready, you merge your work. By merging, it means that you integrate changes from two or more branches into specific branch. As a side note, if you want to understand at a deeper level how Git works, you can check out this course by Paolo Perotta, which will show you the internals of Git. You'll see which folders and files are created inside the Git database, that's the .git folder, as well as other aspects that will help you gain a deeper understanding of Git. Once you're done with this course, Using Git with a GUI, I invite you to watch Paolo's course. Now, back to my course. Branching is great for many reasons, for example, testing different scenarios. For example, do you need to work on something? Well, just create a branch. Didn't work or something else came up? Well, go back to the main branch and create another branch, and when that works, well, commit and merge. I can't emphasize how much easier this makes development. Good, and that was a very brief overview of Git, in general. Let's now talk about working with repositories.

### Working with Git Repositories

Let's start with the definition of what a repository is. A repository is a data structure that stores and manages a collection of files and the revision history. It serves as a central hub for collaboration allowing multiple developers to work on the same project simultaneously. A Git repository keeps track of changes made to your project over time allowing you to view the entire history of your project, providing a comprehensive overview of all changes made in the repo, including who made the change and when. When you create a repo, a .git folder is created which contains all the necessary data to manage version control for your project. You can make local changes and then store them in a remote repository which enables collaboration. Getting your local changes to a remote repository is called push, and getting changes from a remote repo is called fetch and pull, which I'll talk about the difference soon. In fewer words, a Git repository is a powerful tool that keeps track of changes, provides a comprehensive history view, identifies contributors, enables collaboration, and allows for efficient synchronization between local and remote repositories, which is what I would like to talk about next. Remote repositories. Git repositories can be hosted on remote servers allowing collaboration and synchronization among multiple developers. Remote repositories serve as a central point for sharing and exchanging code changes between team members, but before going into the details of remote repositories, there's one thing that I want to talk about. Let me show you the bare minimum that is the bare repository. No pun intended.

### Bare Repositories

The bare repository is a special type of repository that does not have a working directory. It only stores the Git version history and metadata. This means that you cannot work and commit in the bare repo. It can only be used as a central repository, that is a location for collaboration, basically the remote repositories that I was telling you about. For this, let me show you a demo. In here, I have two terminals side‑by‑side. They are currently both pointing to the same directory, Users/savier/git‑demo. Now, what I'm going to do is I'm going to create a Git repository on one end and a Git bare repository on the other. This one, git‑demo, is my normal, if I may call it like that, Git repository, and on the right, I'm going to create the bare‑repository. What I'm going to do is I'm going to work on the left in my normal repository, and I'm going to push my changes to the bare repository. Remember, you cannot work on a bare repository. Now, let me show you. I'll start by initializing my normal repository, and I'm saying normal, but that's not common. You just call it a Git repository, git init dot. That's how you do it. At this point, a .git folder has been created which is the Git database, it's where my changes are going to be stored locally. On the right, I run git init ‑‑bare . which initializes an empty bare Git repository. This one, because it's a bare repository, does not have a .git folder. Let's see. If I run ls on both sides, however, take a look on the left, nothing was returned. That's because the .git folder is hidden, which I can see by running ls ‑a. Now, I change into the .git folder, and as you can see, it has the same file and folders as the bare Git repository, and that demonstrates the difference between a Git repository and a bare repository. Now, let's move forward. I'm going to run git status, but look at this fatal. This operation must be run in a work tree, that's because I'm running inside the .git folder. If I go up one level, and I run git status, I can see that there are no commits yet, and there's nothing to commit. I have a clean work tree. If I run git status on the right, I get the same error message, fatal: this operation must be run in a work tree, which a bare repository does not have one. Now, let me clear, and here's what I'm going to do. On the right, I'm going to get the location of the bare repository, because on the left, I'm going to add this as my remote repository for which I execute git remote add br, this the name of my remote, and then the address. In this case, it's a local folder, but typically, you point this to a Git hosting platform, which I'll be showing you in just a few moments. Now, if I run git remote ‑v, I can see that there's my remote for both fetch and push, and I'm almost set up to push my changes from my repository into the remote repository, that's the bare repository at this point. Now, let's create a couple of files and make a few commits in my repository. So I'm going to use printf to create a file that's going to contain this text. And just like that, I have created a README.md file that has this text, Bare Repository demo/nfile created to demonstrate a bare repository and a regular repository. Now, I run git status, and I can see that there's an untracked file that's basically a new file, but there's nothing yet in the remote repository. Now, I'm going to use git add to add this file to the index, which means that I can run git commit ‑m and the message. I have created my first commit in my repository. To confirm, I run git log, but check this out. This commit does not yet exist in the remote repository. For this, I have to push the change for which I do a git push. In this case, because it's the first one, I have to pass this parameter, ‑‑set‑upstream br main, which is how I specified that this branch is going to be pushed to this remote .br in the main branch, and the push worked as expected. How do I confirm? Well, I run git log in my remote repository and my commit has been pushed successfully, and that is a very simple example of how you can create a repository, a remote repository using a bare repository, and how you can commit and push from the repository to the remote repository, but that was a demo of a repository in my machine that stored its changes by pushing to a bare repository. It works well, but it's not the common scenario that you would run into. Instead, what happens is that many of these bare repositories are placed along with a whole bunch of computing power in the cloud or a data center, you throw in some security, and then all of a sudden you get a Git hosting platform of which there are several options, some of which you may know, some you may not. You can simply have a bare repository or go with GitHub, Bitbucket, GitLab, and there are many options. Typically, those first three are the ones that me and most of my acquaintances use. And in my humble opinion, if you stick to the main ones, for example, GitHub and Bitbucket, these will cover your needs and beyond, and even better, GitHub and Bitbucket have their own GUI tools to work with Git, GitHub has GitHub Desktop and BitBucket has Source Tree, both of which I will cover in just a few moments because there's one thing that I need to explain before moving forward. Let me talk a little bit about security.

### Security SSH And HTTPS

Earlier, I mentioned how Git is very secure, but I was talking about how it's not possible to tamper or modify information once you've added it to Git as the hash is created to guarantee that the data is not changed. There's another type of security that is quite important, which involves securing your repositories, that is who can access and modify your repos, the ones that you have hosted somewhere like GitHub or Bitbucket. For this, let me talk about HTTPS and SSH, which are the two mechanisms used to secure your repos. Just take a note of something, it is not yet itself the one that performs the validation. Remember the previous demo where I created a bare repository, added it as a remote without configuring any security? Well, the point is that it's the hosting platform, the one that's in charge of security, it performs the authentication and authorization. It could be GitHub or it can be Bitbucket, or for example, with the case of the bare repository, it could be write permissions on the folder. Anyway, let's talk about HTTPS. HTTPS Git authentication refers to the use of the HTTPS protocol to authenticate and secure the communication between a Git client and a server. If you use this method, it is required to generate a personal access token, which you can do from your Git hosting platform, for example, from GitHub. You first need to log in, authenticate, and then generate the token. For now, just remember that HTTPS is great for basic usage. What I mean is that it's very straightforward to use as there's no need to create and manage keys, just your personal access token. And since it uses port 443, that's the port used for HTTPS, it is more likely that it will work anywhere. The URL of a project that uses HTTPS will look something like this. This is the repo of a course that I'm hosting in GitHub, but it's a private repository. Very soon, I'll show you connecting to a public repository. Then SSH. SSH authentication refers to the use of the Secure Shell protocol to authenticate and establish secure connections between a Git client and a hosting platform. That's like GitHub. The key is that it takes place without requiring usernames and passwords. SSH allows remote logging and other network services to operate securely over unsecured networks. It helps identify trusted computers without involving username and passwords. A key pair is used for authentication. You need to take into account there are several steps required to configure. On the bright side, SSH is very convenient. Once the SSH keys are set up, users can authenticate without entering passwords for each interaction, providing a streamlined and convenient authentication experience. This is an example of an SSH URL to connect. Now, let's start with a couple of hosting platforms and get clients that are very easy to use.

### GitHub and GitHub Desktop

Let's talk about GitHub. Actually, have you heard about GitHub? Well, chances are that the answer is yes, but let's make sure that we are on the same page. GitHub is a Git‑based code hosting platform for collaboration and version control. In fewer words, remember how I said that Git is distributed? Well, in a nutshell, GitHub is just a place where you keep your remote repositories in a secure way, both private and public repositories. GitHub is so successful that it was acquired by Microsoft a few years ago. It has plenty of features that are quite useful from the basic ones like providing repository hosting and version control with issue tracking, which is quite useful as you can keep your code and a list of things to fix or implement right next to each other. Additionally, it provides continuous integration and deployment capabilities, for example, but not limited to GitHub actions. Also, you can create your documentation and wikis, and best of all, it has excellent collaboration functionality allowing you to build communities and with plenty of social features. In fact, it is where most open source projects are hosted. Also, it's worth mentioning that there are some tools available from GitHub like GitHub copilot, which has been trained on billions of lines of code to help you be a better programmer. Well, that's one tool that can be useful, however, it's out of scope of Git, so I'm not going to cover it now, just wanted to tell you about it. As a side note, before moving forward, if you want to learn the GitHub big picture, I recommend that you watch this course by Aaron Stewart. Aaron is an architect in GitHub, and he makes very engaging courses. You might want to check them out, but this one, in particular, is very useful if you want to learn about GitHub. Aside from copilot, the tool that's relevant to this training is GitHub Desktop, which is a desktop application that simplifies the interaction with Git repositories. It is very user friendly and it's built on top of Git, which means that most Git‑related functionality is available. That is if you use GitHub desktop, there's usually no need to use the command line. And as a bonus, it can work with other Git hosting platforms. Best of all, it's free. Some of the useful features that it has are repository management in a visual way. You can stage, stash. commit, and sync changes. It also provides branch management and collaboration features, including the ability to create pull requests, as well as many advanced features. Having said that, let me give you a quick demo on GitHub and GitHub Desktop. Here, I am in github.com. If you do not have an account, just click up the Sign up button and get started, it's easy and free. In my case, I do have an account, so I'll click on Sign in and provide my credentials. GitHub is very secure. Now, I get to the two‑factor authentication page. Given the sensitivity of some of the data that you'll be storing in GitHub, it is recommended that you enable two‑factor authentication. Your source code is going to be there after all. In my case, I'm going to use GitHub Mobile, and here I am. I am xmorera in GitHub. On the left, I can see the list of the repositories that I have created and the teams that I belong to. Also, remember that GitHub is pretty social, so I can see changes that have been done in other repositories by people that I work with or other repositories that I follow. Let me navigate to my profile. Here's where I can see all of the information that's relevant to my profile like the repositories that I created or those that I have starred. I can see also my personal information on the left. If you ask me, it's important that you keep this information up to date because some companies that are hiring may go over this. In fact, many companies are now checking github.com to see the contributions that you have made. If you are very active, that may help. In my case, I use GitHub very frequently. You can see that I have 1100 contributions just in the last year. This includes commits to private and public repositories. Anyway, I go back to the main page. I'm going to click on New because I want to create a new repository that I'm going to use for this course. I'm going to call it ps‑git‑demo, and I provide a description. Next, I can select if it's a public or private repository. This case, I'm going to leave it as public, and I'll initialize the repository with a README file. There are other options, for example, adding a .gitignore, which you use to indicate which files should not be committed to Git. And you can add a license. You can also include Azure Pipelines if you want your project to be built automatically. I'll just leave as‑is and click on Create repository. And just like that, I have the repository that I'm going to use in this training. As mentioned, once you create a repository, there's a lot of functionality that's going to be very useful. For example, the issues where you can keep track of bugs or issues in your project, pull requests, which I'll cover pretty soon, actions to build your project, and more. download it and install it if you haven't done it so far, once it's installed, the first thing that you need to do is to authenticate for which you select file options and you select to which account you're going to be logging into. You have GitHub dot com, which is the general platform and you have GitHub enterprise, which is a special type of GitHub that's installed in your company. I'm going to sign in using the browser I authenticate because I have already logged in to GitHub dot com and just like this, I've been authenticated. I am ready to start working with my repositories in GitHub dot com using GitHub desktop additionally. For this course, I'm going to be using Visual Studio Code which is my external editor of choice. In most cases, it's also the one that comes by default with GitHub desktop. Now, i'll click and save and it shows me the repositories that I have configured in this machine. If you just installed GitHub desktop, this will be blank. It is at this point that I can get started by selecting clone and specifying which is the repository that I want to clone locally. But there's another way if you go to GitHub dot com and you select a particular project in that code button, there's an option that says open with GitHub desktop. If you do this, you will get the option to clone right away. This is very convenient.

### BitBucket and SourceTree

Let's talk now about Bitbucket and SourceTree starting with Bitbucket, which let me ask you, have you used it before? Maybe you have, maybe you haven't. My recommendation is to at least try it. But what is Bitbucket? In a nutshell, it's a Git‑based hosting platform for version control from Atlassian, which are the makers of Jira and Confluence. This means that with Jira, you have one of the best experiences with issue tracking out there, and with Confluence, you get the documentation. Bitbucket has excellent integration with both of these tools. In a nutshell, Bitbucket is very similar to GitHub. You get the pipelines, collaboration, code review, and more. It's very similar. And aside from Git, it also provides support for Mercurial, which is another distributed version control system. And once you get started with Bitbucket, you can then use SourceTree, which I may say, it is similar to GitHub Desktop. And as mentioned, I'll be using source three in this course. And now let me give you a quick demo on bit bucket and source three. Here I am in bit bucket dot org. You can get started for free. And if you're already using it, just log in this case, I'm going to log in with my account. And here I am welcome to bit Bucket. If I scroll down, you can see that it's already integrated to my Jira instance. In this case, it is showing me the tickets or the clips that I'm currently recording. As I mentioned, I use Jira and GitHub to manage my course creation process. So each one of these clips has their own ticket in Jira. I also store all of my files in GitHub and large object storage. But anyway, this case, I'm just going to create a repository. This is a similar process to what I showed you in GitHub. It's a new project. First, I provide a project name, plurals site and then I provide the name PS git demo. I want it to be a public repository. I'll include a README with a tutorial for beginners. And the default branch name is going to be main. I click on create repository and just like this, I can get started. As you can see on the left, there's a whole lot of functionality that's somewhat similar to what I get in GitHub. I do believe that the Jira integration is a little bit better in bit bucket, which one to use bit bucket or GitHub is something that I'm going to leave to you as it's a personal decision. And then if you decide to use bit bucket, you can customize your repository in multiple different ways and you can clone pretty much in the same way as you did with GitHub, I click and clone. And it gives me the option to clone in source three or clone in VS Code and that's it. I'm going to leave this for now because I want to show you bit bucket to get bit bucket, navigate to source three app dot com. You can download, install and get started. Once you have source free, you can start by clicking on remote and adding an account. It can be a bit bucket account to connect to big bucket or if you wanted to, you can also connect to a different hosting platform. For example, let me select GitHub. In this case, I'm going to connect by using a token I authenticate in GitHub and there it is, I have been authenticated in sorcery. That was very easy, right? Let me uncheck show organization repo. Now I'm ready to clone my repository and get started, which is something that I'll cover in the next module.

### Other Options

GitHub or Bitbucket? Either one as a Git hosting platform works wonderfully. GitHub Desktop or SourceTree, either tool for working with Git using a GUI works fine as well. They are all great, but they're not the only options. In fact, there are plenty of options for both hosting platforms and clients. I can name as hosting platforms, CodeCommit from AWS, Beanstalk Perforce, Gerrit, Codebase, and SourceForge to name a few. For Git clients, you have TortoiseGit, GittyUp, GitAhead, SmartGit, Fork, and gitg, and there are more. I've used several, in fact, but there are two that I also like very much and that you might want to check out. They are GitLab as hosting platform. It has very powerful CI/CD capabilities, a slightly more generous free tier, and also a self‑hosted version called GitLab CE, that's Community Edition, which means you can deploy inside your organization to get started. You can migrate to Enterprise Edition later, if needed. And as an alternative Git with a GUI client, GitKracken is one that you could try as well, especially if you're using Linux. There's an official version that works pretty well. It's worth giving a shot if you have some time or you're using Linux. And now, let's do the takeaway for this module.

### Takeaway

As a takeaway for this module, we learned that Git was created by Linus Torvalds, the person who created Linux. Git is a source control system that's distributed, powerful, and flexible where each developer gets a local copy, can work, and then can push to remote repositories, which is what enables the powerful collaboration capabilities that Git provides. That is when you're done, you push your changes to remote repository and then you and other developers can fetch and pull changes from the remote repo. Git has integrity and performance and branches are Git's killer feature. The Git local repository stores all history, namely it keeps track of all changes over time and who made the changes. It does this, and it uses four different types of objects, the commit, blob, tree, and annotated tag. In Git, you have what's called a bare repository, which does not have a working directory, it can be used as a central repository, but there are many Git hosting platforms that are built around Git, for example, GitHub and Bitbucket, which provide multiple security mechanisms to secure your code, for example, HTTPS or SSH. Additionally, there are multiple Git GUI tools that allow you to work with Git using a GUI like GitHub Desktop and SorceTree, but there are plenty of other options like GitKracken. And with this, I invite you to join me in the next module, the mechanics of Git.

## Mechanics of Using Git (With a GUI)

### Mechanics of Using Git (With a GUI)

And so we arrive at the meat and potatoes of this training. These are the mechanics of using Git with a GUI. This is the must‑watch module if you're serious about using Git, especially if you're just getting started, even better, if you prefer to skip the command line. So here is what this module is going to cover. First, I'll show you how to get the code. I'll explain how to clone repositories, and then I'll talk about forks. I'll then cover pull, fetch, and checkout. This is all business as usual. I'll move on into how to commit for which I will explain staging, unstaging, funks, the index, and I will also get into conflicts. It will also include best practices like how to create a proper commit message, something that some developers take for granted, but that can really make a difference. Then what to do if you want to save code for later, also known as stashing, and because mistakes happen, we will learn how to discard changes. Also ignoring files is important, then merge, push, and pull requests, which is something you need to know very well if you're going to collaborate, which takes me to branches which in my humble opinion, is one of the best features of Git, they enhance and extend the possibilities of what can be done in software development, especially if you have a good workflow. And finally, some words on cleaning up. Let's get started.

### Quick Tour of SourceTree

Let's start with a quick tour of SourceTree. This is SourceTree. Hello SourceTree. When I open it for the first time, I may get prompted to authenticate with a Git hosting platform. Typically, I always dismiss this dialog boxes, but it's going to be important. I'll come back to it in just a few moments. Authentication aside, when I open SourceTree, I get this tab which shows me which are the local repositories that I have in my machine and that SourceTree is aware of. These are called bookmarks. And by the way, this is SourceTree for Windows. It may be possible that you're running in a Mac, and there's a slight difference in the Mac, namely that the repositories, that is the bookmarks, are shown in a separate window. In this case, I have one bookmark here. When I select it, that repo is shown in a separate screen. In this training, I'm going to be focusing on Windows, not on Mac, but the functionality is pretty equivalent. So let me go back to Windows. At the moment, I do not have any repositories, and there are several ways of getting a repo, which I'll show you soon. One thing that's really important and useful is the fact that SourceTree allows you to authenticate with a hosting platform like GitHub or Bitbucket, as I just mentioned, which then allows you to browse and perform actions with your repos directly from SourceTree. If you haven't authenticated yet, navigate to Tools, Options, and the Authentication tab, make sure that you have added the account that you're going to use. In my case, I'm logged in to GitHub, which is the Git hosting platform of choice for this course. Please also notice that depending on which authentication you use, be it HTTPS or if you authenticate in a separate way, for example, with SSL, then the URL to get a repository is going to be different. Anyway, once you have authenticated, you can select the Remote tab and browse your repositories from each one of the accounts that you have authenticated to. You can search for repos, both your personal repos, as well as organizational repos. This makes things a lot easier when getting repositories. Talking about repos, this is where you can see the repositories that SourceTree is aware of that are in your machine. As mentioned, it's called the Bookmarks area. When you get a repo with SourceTree, it is listed here. There are some important information shown, including the name of the repo, the folder where it's stored locally in your machine, and the branch. When you start adding multiple repos before it gets messy it's recommended that you group them logically by creating folders using this button right here like this. Here is the Pluralsight folder. Here's where I group all the repos that I use for building my Pluralsight courses. One great feature of SourceTree is that it has a tab UI, which means that you can open multiple repositories or have multiple bookmark tabs open at the same time and switch easily between them. Once you select the repo, it loads showing you several sections of interest starting with the side bar, which provides options for the repository's current workspace, available branches, remote branches, and tags, then the toolbar which provides the command that you would normally use in the command line. Those are the ones on the left, but you also get a few options on the right, for example, to initialize the repo, you use Git‑flow, open the remote, open a terminal, open an explorer and the repository settings. And this is at a high level, a tour of the different sections of SourceTree. In the upcoming videos, I'll show you some live demonstrations that go deeper into the different available functionalities that SourceTree has for you.

### Getting the Code: Clone and Fork

To get started, you need code, well, a repo, but how do you get your code? Well, there are several ways for which I would like to talk about clone and fork. Cloning is the primary mechanism used to get code. However, there are other two ways available from SourceTree that I would like to mention. Namely, add, which is used once you already cloned into your local machine, which is used once you already cloned. I'll tell you more in a few moments. And the third one is create. Let's start with clone. Cloning is one of the most fundamental operations in Git. It creates a local copy of the entire repository, including files that commit history, branches, and tags. You clone from an existing remote repository, which is usually called origin for which you require permissions, something that varies depending on the Git hosting platform that you use. To clone, you need the repo URL which you obtained from the hosting platform that you selected. Once you have cloned the repo, it is time to start working. It basically goes like this. You have your origin that's where your remote repo is stored. This usually lives in a Git hosting platform. You clone and then you get a local copy of the repository. Cloning provides a convenient way to get started with a project and is especially useful when collaborating with others as it allows multiple people to have their own local copies of the repository, which once they've finished their work, they can contribute it back to origin. Let me show you how to clone. Here I am in GitHub in the ps‑git‑demo repository that I created in the previous module. To get some code, I click on this green button which presents me several options to get my code. It all depends on which type of authentication I am using, for example, HTTPS. This is the URL that I would use if this is the authentication type that I selected, or if I configured my machine with SSH, this would be the URL, and I can also use the GitHub CLI, if needed. Additionally, I can open directly with GitHub Desktop, Open with Visual Studio, both of these options would still allow me to commit and push back, and there's the third one, the disconnected option where you just download a copy of the latest files. Let's copy the URL for SSH. And now, I go back to SourceTree and select Clone. Here is where I provide the URL of the remote repository. One thing to note is I selected SSH, but when I authenticate it in SourceTree, I use the personal token, which means that I need to actually select HTTPS. Let me show you what happens if you select the wrong URL type. Basically, SourceTree tells me that this is not a valid source path or URL. It is actually valid. The problem lies in that I do not have access, and I showed you this in case if you run into this issue, then you can check your authentication type. So I will go back to GitHub and select HTTPS, I'll copy this URL, I'll go back to SourceTree, and now that I've entered the HTTPS URL, SourceTree recognizes that this is a Git repository. At this point, if I click on Clone, I'll basically get a local copy of the repository in this location, and with this name, ps‑git‑demo. Additionally, there's an easier way to look for a remote repository. I simply select Remote, I'm going to uncheck Organization Repos, and I'll just put the name of my repository, git. Here it is, ps‑git‑demo, and there's a label here telling me it's from GitHub. If I had multiple hosting platforms configured, this label will tell me which is the source. So I click on Clone and I specify my local path. As convention, I always use the name of the hosting platform, then the repo owner, and then the repo name, so C:\github\xmorera\ps‑git‑demo. I can also select if I include it in a folder, and there are a few advanced options like which is the branch that I want to check out and the clone depth. Adding a 0 includes all the history. If you added a one, it would include only the latest commit, basically the head. Now, I'm going to click on Clone, and just like this, I have a local repository and here's the main branch with the initial commit on repository creation. Let's now look at the next functionality, Add, which as the name implies, adds an existing local repository in SourceTree. Notice that it adds the bookmark, not the repository. Why do you want to use add? Well, maybe you cloned using a different tool, for example, the CLI or another Git client, and now you intend to use in SourceTree. Well, add makes it very convenient. Also, it may be the case that you removed a bookmark. Well, you can add it again very easily. Here's how it works. Basically, I'm going to open a new tab which shows me the repositories that I currently have, and just to show you, I'm going to delete the bookmark for the ps‑git‑demo. Notice that you can also delete the repository, but in this case, I only want to remove the bookmark. Now I click an Add and here I select ps‑git‑demo, I select the folder, and just like that, I've added the repo. Once again, I have the bookmark. The third option is Create which creates a new local repository from scratch. It's pretty much the same idea as the process that I ran in GitHub and Bitbucket, but I am doing it locally. However, SourceTree allows us to create the repository and remote also. You just need to select the account and the owner, or you create locally and then you specify the remote repo and branch when you perform a push. Let me show you. For this, let me open a new tab and click on Create. Here, I'm going to specify the path where I want this repository to be created. Here it is, new‑git‑repo. That's going to be the name of the folder, and I can select if it's Git or Mercurial. Of course, I'm only interested in Git at this point. And here is where you specify if you want to create the repository in your remote. Here, it's going to be xmorera, that's my account. I'm going to be the owner, and here, I can optionally provide a description and indicate if it's private. I am not going to create it right now, so I unselect and click on Create, and just like this, a new repo has been created. And then finally, on this topic, let's talk about fork. In the context of GitHub, a fork is a new repository that shares code and visibility settings with the upstream repo. A fork is often used to make changes that are then proposed back to the upstream repository via pull requests. This is commonly used in open source projects, and it's very useful for experimenting, prototyping, and independent development. Let me show you an example. Here is the repository for SolrNet, which is an excellent API for working with Solr in .NET. It's an organizational repo, and it's open source. In this case, I can't work directly in this repository. So what I need to do is click on this button right here, Fork. Here, I specify the repository name for the fork, description, and whether I want to copy the main branch or others. I'll leave as‑is and click on Create fork, and this is just going to take a little bit of time. Once it's ready, I have my own repository that I've copied from the original repository. Now, I can work on this code, make all the changes that I please, and once I'm ready to contribute back to the original, I just create a pull request, which is something that I'll show you in just a few moments.

### Committing Code, the Index, and Staging

Once you cloned a repository, be it a new repo, an existing one, or a repo that you forked, the next step is working on your code. Well, Git is not only meant for code, so maybe I should say working on whatever it is that you stored in your repo. But let's say that you're storing some source code, and you made a few changes. What is the next step? Well, there are several possibilities. First, which I think is the desired outcome, is that you commit your code, that is you include your changes in the repository, although it might be possible that you want to stash your changes, discard them, or even ignore them. But for now, let's ignore those three last that I just mentioned. Let's focus on this particular one, commit. Commit, which is something that I briefly mentioned in the previous module is a snapshot of a repository at one point in time. It is one of the core building blocks of a Git project timeline. It captures changes made to the project and serves as checkpoint or milestone in the history of your code base. Each commit includes a snapshot of files, a commit message, the author and timestamp, as well as a unique identifier. For example, this is the initial commit that was made when I created the repo in GitHub. If you look right here, you can see the Commit ID, including the full one and the short version, the author, the date, and the committer. Notice that even though I'm the author, the committer is GitHub. This is because this commit was created in GitHub when I created the repository, and here's the initial message, as well as the file that was created. Now to understand the process used to commit, it is necessary to talk about the sections of a Git project. First of all, the working directory, which is your copy of the code, it is where you make changes. Then the staging area, which is also known as the index. Here is where you bundle up multiple changes to prepare for a commit. The index is a file located in the Git directory, but I'm not going to get into the Git internals right now. Then the Git directory, which is the folder that contains all the objects that are used to store all the history of your source code. Your usual workflow is you check out you stage and you commit. Now let's talk about the three states of files you have modified or untracked, staged, and committed. Modified are simply files that have been changed since the last commit. These can be text files or even binary files. Untracked are files that have never been committed. It's useful to note that Git detects changes by comparing current state of the file with the version in the repo. Then stage, which is the process of marking specific files to be included in the next commit. These files are added to the index which is also known as a staging area. An entire file can be staged or only a specific portion. This specific portion is known as a hunk. A hunk is a portion of a file that contains changes which are continuous block of lines, and this provides great granularity of which changes are going to be included in the next commit. And then committed, which represents files that have been added permanently to the repository. It takes a snapshot of the files and creates a new commit with a unique identifier. At this point, the commit becomes part of the repo's history. Let me show you a demo. I'm going to copy some source code. This is some Python code that just returns some information on the file system. It comes from another one of my courses about working with files in Python. At this point, I created a new file and added some code. So if I go to GitHub, I can see that I have a new file here in the unstaged files section. Also, I can see a question mark. This is what's telling me that it's an untracked file and at the top in the repos history. I can see that there are some uncommitted changes. Now, if I select the file, it shows me my code. Basically all this is new code. I could click on stage hunk to add this code into the index or I can use the plus sign. It's the same process. It is showing me the diff the diff is what shows you what has changed those plus signs and the green highlighting tells me that all this is new code. Now I click on commit and I need to provide a commit message. Let me now pause for a second because commit messages are really, really important because it allows you and other developers to know the exact purpose of a commit. And there are several recommendations. I'll call them rules right now that you might want to use. First of all, separate the subject from body with a blank line. Second limit the subject line to 50 characters, capitalize the subject line, do not end the subject line with a period. Use imperative mood in the subject line, wrap the body at 72 characters and use the body to explain what and why versus how these are the recommended rules by several people who know a lot about git more than I do. Here's an example of a great commit message. The subject is very clear, less than 50 characters, then the body less than 72. And it explains the why. Just always remember to read it like this. If applied this commit well, and then read the subject line. If it explains, sued what this commit does, then you have a great commit message. Also, there are multiple commit options. They're a little bit more advanced. So i'll cover them in a future module for now. I'm just going to click on commit and make sure that I do not have selected either one of these check boxes because I don't want to push to origin main and I do not want to amend latest commit. This is one option that might be tricky. I'll cover it pretty soon and just like that I committed. Now, let's add a couple of other changes. What I'm going to do is I'm going to go back to Visual Studio Code and now I'm going to make a couple of changes here. I'm going to expand so that this code will run in both Windows and mac and let me know the difference and I'll also modify it down here. There it is. I've made 2D different changes. Now, when I save when you're staging lines, if you want to make a change, that includes both what was added and removed. You need to select both. If you only select one. Like in this case, when you commit, you're going to have a duplicate line because you're not removing the older line, you're only including the new lines. OK? I'll just make a commit there. It is one more commit and I still have these two hunks at this point, I can stage one stage the other now and provide the commit message I commit. And just like that, I've made another commit with the other hunks. As you can see, we're starting to build a history in our git repository. Although at this point, everything is locally, I have not yet shared these changes with a remote repository, something that we will talk about very soon.

### Push, Pull, and Fetch

Up until now, you learned the steps required to commit, be it code or other type of file, but if you remember the diagram of your local and remote repos in Git, what I just showed you takes place over here in your local repository. You're working by yourself. You are not really collaborating at this point. To collaborate, you need to use the distributed part of Git for which you need to learn about push, pull, and fetch. Let's talk about each one of them. Push is the command that's used to transfer your commits from the local repository to the remote repository. It's what you do after working and committing your work. You push the remote repo. If something happens to your machine and you haven't pushed, then you may lose your work, but it's always important to push. You push to specific branches in the remote, which need to be preconfigured what's known as tracked, or if they do not exist, you can publish new branches. When you push, it compares commits in your local branch to commits in the remote repo, then it transfers this commits, and updates the remote's history. There are different available options. For example, you can force push, you can push tags, and you can also skip validations, what's known as hooks. Let me show you a quick demo on push. Here's SourceTree with the three commits that I did locally. If you see here, these labels tell us the latest commit that's right now in the origin, which is the initial commit, and these three are the commits that I have in my local machine. This label right here tells me which is the pointer to the latest commit in main, which is local. To confirm I can go to GitHub, and as you can see, there's only one commit. That's the entire history at this point in GitHub, the initial commit. Now, let's go back and click on Push. By default, it's going to push to origin and to the track branch, which in this case, it's main. It's currently tracked because I cloned from this repository, so everything is already set up. This is the repository where I'm going to push to origin. I can configure additional repositories and push to multiple remote repos, but at this point, I only have one, and in this branch, main, which is a tracked branch or my local main branch. Now, when I click on Push, just takes a few moments, and my commits have been transferred to the remote repository. You can confirm by seeing these labels. Main, my local branch and origin/main both point to the same commit. The histories are synchronized. To confirm, I go to GitHub, and now I refresh and it went from 1 to 4 commits. The three commits have been transferred successfully. I click to check and I can see my three commits right here. Now, if I go back to SourceTree, I can see on the left the branches, these are my local branches, I only have main at this point, and here in the remote, I have the origin main branch. Now, that's how you transfer your commits from your local repository to your local repository. Let's now go in the other direction which brings me to my next command, fetch. Fetch retrieves latest changes from a repository; however, it does not integrate them with your code. The integration part is what's known as merge, which means it allows you to review and decide when to merge, which is very useful. It is worth noting there are some tools that run a periodic fetch, which means that your repo is kept updated. Let me show you a sample of fetch. Here I am in my repository. I'm going to select README.md, and now I'm going to edit this file which I can do directly in GitHub. What I'll do is that I'm going to add one line, Fetch test, and then I'm going to commit. This is one way of making changes directly in GitHub, but there's another one. If you're in a file and you press period, it opens Visual Studio Code on the web, which is really useful for making many changes and then committing once it's done. You can view multiple files at once, but I already made my change, so I'm going to go back to my repository. This change that I just made is a commit that's only found in the remote repository. It's not in my local repository. Let me show you by going to SourceTree. Here I am, and as you can see, SourceTree still doesn't know about this commit. SourceTree thinks that my main and origin/main branches are synchronized. So, let me click on Fetch, and I'll leave the default options, but one that may be useful is to decide if you want to fetch from all remotes or just this one, and there it is. Now SourceTree knows that origin/main is one commit ahead of my local repository. You can see it because of these labels. Here's main, here's origin/main, and that's how fetch works, but to confirm, let me just show you for a second in Visual Studio Code my local version. Even though that commit has been transferred from the remote repository to my local repository, it has not been merged just yet. For this, let's talk about pull, which takes me to pull. Pull is a combination of fetch and merge, fetch being what I just showed you, merge being the combining of the commits. It's basically like running git fetch && git merge origin/master in the command line. Merge means combining changes from different branches, in this case, a remote branch and a local branch. By default, it pulls from tracked remote branch, but can pull from different branches, even other repositories just like what I showed you on fetch. However, one thing that you need to take into account is that you have to be very careful with conflicts. Let me show you a quick demo. Let's pick it up where we left off. Origin/main is one commit ahead of main. I have already fetched, but I haven't combined the changes for which I can click on pull. Let me show you pull in action for which I'm going to put SourceTree down here, and I'll show you the README.md, and now I'll click on Pull. I get the option to select from which remote I want to pull, in this case, its origin, and which branch to pull. Additionally, there are a few options. For example, I can indicate if I want to commit merge changes immediately, if I want to include messages from commits being merged in the merge commit, create a new commit, even if fast forward is possible, fast forward being something that we're going to talk about in a few minutes, and then whether I want to rebase, instead of merge, again, something that I'll mention soon, so I'll click on Pull. And now, main and origin/main are in sync, and if you noticed, the line that I modified in GitHub was updated here as well, Fetch test. Let's keep moving forward.

### Stash, Discard, and Ignore

In an earlier video, I showed you what you do after you work. Basically, you commit. However, there are a few other actions available that are useful in other scenarios. I'm talking about stash, discard, and ignore. Let's start with stash. Stash is used to save changes from your working directory, that is changes that you might intend to use, but you are not ready to commit just yet. When you stash, you remove these changes from your working copy, but it only works with files that you have already committed previously. You cannot stash new, that's on track, and ignored files. I'll come back to ignore in just a moment. Basically, stash is useful if you're in the middle of a change, but then you need to work on something else, though, you can stash your changes in other source control systems, the word shelf is used as well. But the point is that you store those changes for future use. You know, I've had a boss or two that likes to change my priorities. Stash is what I would use in that case. It's worth noting that stash is a local operation. Stashes are not pushed to the remote and you can have multiple stashes, you can even assign names to them. And then at a later point, it's possible to reapply these changes, that is reapply the stash, what's known as popping, and you can keep the stash or remove it after applying. If you keep it, it's useful to apply the stash to multiple branches. Let me show you a sample of how to use stash. Here I am in my repo, ps‑git‑demo. I'm going to switch to Visual Studio Code and then make a few changes. Nothing too complex, just print, and then I'll just create the main function and call it, which is very standard in Python just like this. Now, when I go back, I can see that there are some uncommitted changes, but I'm not ready to commit these changes just yet. So what I can do is click on stash, which will stash my current changes and return my working copy to a clean state. I can add the name for a stash and then I can indicate if you want to keep the stashed changes in my working copy or if I want to remove them, that's what this checkbox right here is for. Now, let me reposition SourceTree so that you see what happens. I am not keeping my stashed changes, so when I click on OK, a new stash is created, and as you can see, my code was returned to the working copy. The function definition was removed. And now if I expand the STASHES section, I can see that I have one stash, Adds main. Here it is. These are the code changes that I made, but did not commit. And now at any point, I can right‑click, and I get two options. I can delete the stash in case I don't need it anymore or I can apply the stash just like this, and yes, I'm ready to apply this stash the working copy, so I click on OK, and there it is. The stash has been reapplied and my code is ready to go. Just remember that it hasn't been committed yet. Next up, discarding changes. Basically, you can discard existing changes, that is maybe mistakes or errors that you made or maybe you are trying something different or just making a temporary change. By using discard changes, you can revert to the last committed state, which is needed if you're switching between branches and you do not have a clean working copy. By the way, if you want to keep those changes, that's when you apply stash, but if you don't, you can use discard. Also, it's useful for resolving conflicts, but it's a permanent destructive operation, so you have to be careful. Let me show you. In the interest of time, here are the changes from the stash that I just reapplied. As you can see, these changes have not been committed yet, so let's say that I did not want to make these changes. I tried this out, and I just want to discard them. So one thing that I can do is I can right‑click and select Discard. This is quite useful if you only want to discard one file in particular. The other option is to go right here at the top and click on Discard. There are two options. You can select which files you want to discard. In this case, I only have one, so it's pretty straightforward, or you can click on Reset All which abandons all local changes. Be careful when using this option because you might lose some work. Anyway, so what I want to do is I want to discard file changes, so I'm going to click this button, confirm that all is okay, and just like that, all changes have been discarded. I, again, have a clean working copy. And then finally, ignoring files. Why? Well, because not all files are meant to be committed. There may be some personal machine settings, build artifacts, compile binaries, output directories, or other types of files that you do not need or should not commit to a repository. In particular, never commit a key or a secret. Anything that's private should not be committed. In that case, you can specify which files should be ignored. For this, Git uses a gitignore file which has several scopes, it can be for one repository or for all repositories in your machine. There are several options for ignoring files, for example, by exact name, pattern, path, or extension. Let me show you. Here's my source code. Let me go ahead and create a personal machine setting. In this case, it's the one for running and debugging a Python application, so I click this button which creates a launch.json file. As you can see, it's right here inside a .vscode folder. This vscode folder typically has settings that are specific to my machine, but it's not something that should be committed, especially because if someone else pulls, it might overwrite their personal settings. So one option that I get is, and let me go to SourceTree, I can right‑click and select Ignore which gives me the options that I just showed you. It can ignore the exact file name, that's .vscode/launch.json. You can ignore all files with this extension, which you have to be careful to avoid ignoring files that are actually useful, or ignore everything beneath a particular path, in this case .vscode, or ignore a custom pattern. Additionally, you can ignore for this repository or for all repositories. I'll select exact file name, which what it does is it creates the .gitignore file with this file name. That's the pattern that right now is used to ignore this file. As you can see, launch.json is no longer in the onstage files. If I change the name, then it no longer matches, so as you can see now, launch.json is back in the list of onstage files. I can also use a pattern like this one. Now, everything underneath .vscode is ignored. Once you have created your .gitignore file, remember to commit it and push it. Now everything under .vscode in every one of our developer's machines will be ignored. To confirm, I go to GitHub and here it is ps‑git‑demo/.gitignore. And here it is, the gitignore file. Let's keep moving forward.

### Branch and Merge

When using Git, there are two things that you really need to understand to manage your code more effectively, branching and merging. Let's start with the definition of branch. Technically speaking, a branch is a lightweight movable pointer to a commit. That's kind of a mouthful, so let me put it in a few simpler words. In a nutshell, a branch is just an independent development line of work. It's basically a mechanism that's used to isolate changes which allows you to work on different features or bugs without affecting the main codebase, which is really, really important. Each branch has its own commit history, and the default branch is main, although main is kind of a newer word, it used to be master, but the convention has changed. With branches, developers can create new branches when they work. In fact, it's part of their daily workflow because changes made on one branch do not affect other branches until they are merged, which is something I'll show you in just a few moments. Let me show you first how to branch. Here I am in Visual Studio Code with my application. Here's my application, which I've been committing to this repository, but up until now, I've always done it on the main branch. I only have one branch which you can see right here. It's the main branch. Okay. Let's work on creating another branch. For this, I'm going to go to SourceTree and click on the Branch button. At this point, I provide a new branch name, for example, develop. The name of the branch depends on your workflow. For example, you can have here the name of a feature, maybe even your name. This is a Xavier's branch, so no one else should touch it. Additionally, and this is pretty important and we will get to this soon, you can also delete branches, but be careful when you do it. Anyway, let me create the develop branch. At this point, I need to decide if I want to create my branch from the working copy parent that is where my repository is right now or if I want to indicate a particular commit. This is very useful because, for example, I can go back in time and create a branch from a specific commit, which is quite useful, for example, for A/B testing or to try to reproduce a particular bug. Anyway, in my case, I'm going to use Working copy parent, and I'm going to check out the new branch. Checking out means that you're changing to the new branch. So I create the branch, and as you can see, here on the left, a new branch has been created, and it's the active branch. I know it's the active branch because develop is in bold. Now, when I go to GitHub, I'm going to look for my new branch, but it's not there. Do you know why? The answer is simple, because I have not yet published my branch. I have this branch only in my local repository. I'll publish it soon. Now, let's make a couple of changes because at this point develop and main, they both point to the same commit, this one here. So I come here, I create a new file, there's locate.py, and I paste some source code. I save, I go back to SourceTree, and I have my uncommitted changes. I stage the changes and commit. Here's the message. Please note that it says push changes immediately to and it doesn't have the name. That's because it has not been yet published. I'm not going to push it at this moment, so I leave unselected. Now I click on Commit, and immediately, I have a new commit and this branch is one commit ahead of main and origin/main, but still, we have only one history line that is a straight line of commits. That's because we have a single commit history. And now, I can push. At this point, I select develop and I click on Push, and just like that, a new branch has been published, origin/develop. Now, if I go to GitHub and I refresh, I can see that the develop branch has been published. Now, let me show you how easy it is to change between one branch and the other. Right now, I'm in develop that has the dirs\_locate.py file. To change between one branch and the other, that's the checkout, I simply double‑click on the other branch name, and just like this, I am now in main. As you can see that dirs\_locate file has disappeared because I switched from one branch to the other, and SourceTree selects to commit where I'm currently located. SourceTree removed the file. At any point, I can switch between one branch and the other. Up until now, we have a linear history, but what if I wanted to have independent lines of development. Well, in this case, I am right here in main which does not have the dirs\_locate.py file, and I'm going to create a new file, platform.py. I'll add some source code, I save, and then I'm going to commit. Here it is, Adds platform.py. I stage the change, push changes to origin/main, and commit. And now, I have two different histories, one in main, which has platform.py, and one in develop, which has dirs\_locate. Now, I have divergent histories, and both of my branches have been pushed to the repository. Now, let's talk about what happens next. Basically, merge. When you're going to work with branches, you have two options. You can create a branch or pick an existing branch. Picking an existing branch can even be picking someone else's branch that's in remote, you just need to check it out. Now, let's talk about merge. In my humble opinion. Merging is the unsung hero of Git. As I mentioned earlier, Git's secret weapon is branching. Well, not so secret, maybe killer feature, but the thing is that branching would be nothing if merging wasn't as easy and effective as it is. Merging means combining changes from one branch to another. Basically, it takes the commits from the source branch and integrates them into the target branch. This is how new features and bug fixes are added. It is what enables collaboration. It's possible to join multiple branches. However, there are some limitations with the default UI functionality. When it comes to merging, there are multiple merge strategies. The most common ones are the standard merge, which is the one that SourceTree typically uses. It performs a three‑way merge and creates a new merge commit when merging branches. It automatically detects and handles conflicts, which allows you to review and resolve the conflicts within the SourceTree interface. The other option is fast forward merge, which is if the commit history of the branch is being merged forms a direct line, SourceTree may automatically perform a fast forward merge. This happens when the source branch is based on the latest commit of the destination branch. In a fast forward merge, no new merge commit is created, and the destination branch is simply updated to include the commits from the source branch. That was quite a mouthful. Basically, just think of it like taking all the commits and placing them first in line in the destination branch. Let me show you. I need to do is to merge, develop into main for this. I can go to SourceTree and let me just put it like this so that you can see the changes. Here's main, here's develop. This is the file that I want to move to begin a merge, first select the target branch. This case is main, then click on merge. Now here is where you pick the commit that you want to merge into your current branch. So I'm going to go here the one in develop. Now when I click, OK, any commits that are part of the developed branch are going to be merged into main there error, multiple options here I can indicate if you want to commit, merge immediately. That's if no conflicts. I can also include messages from commits being merged into the merge commit. That's if you want to have a larger commit message and I can create a new commit even if fast forward is possible. That's what I was telling you about taking the history and just putting it first in line in the main, I can also get the option of re basing which I'll explain pretty soon. And I have the option to detect renames with a high similarity. Anyway, I'm going to leave SS and click an OK. And just like this, my histories have been merged locally Maine now also has the commits from develop. But locally I know this because I can see the labels origin main is in this commit while Main in this one. And I have a little number telling me that there are two commits that need to be pushed before I do that. Let me show you here. I am in Maine which has both files. There's locate and platform dot pi if I change to develop, I only have theirs locate. So let me switch back to main, i'll click on push and just like that origin main now includes the commits from develop. If I navigate to GitHub, I can confirm in this branch, you can see the two files while develop still only has the des locate dot pi. As you can see the merging applies in one direction, develop is still intact.

### Pull Requests

Now, let's go to a very important part of working with Git that you need to learn, pull requests. Pull request is a feature that allows proposing changes to a project's codebase, but here's something that's very important. It's not a native Git feature. Instead, it's provided by a Git hosting platform, for example, GitHub. It is the method of choice to contribute in open source projects, but it's also common in commercial development. It's an essential part of a collaborative development process. Here's how it works. After you make a change and you push to your branch, you open a pull request, then developers can review and discuss changes, and once you open a pull request, additional changes can be added. Then at some point, your pull request can be accepted or rejected by a maintainer, that is someone who has enough rights. If accepted, a merge takes place. In a nutshell, it's just like saying, hey, here's my code. Please take it. Let me show you. Here's my application. I created a new file, simple\_differences.py and added a little bit of code. As you can see, I have uncommitted change. If I just do a commit and then perform another change, I add a few more lines of code. I commit again, and now, I use this commit option, Create Pull Request. I select it, then provide the message, and when I commit, SourceTree is going to automatically redirect me to my Git hosting platform to the screen where I can create the pull request like this, I just click on create pull request and just like that I have a new pull request for all the developers to come in review comment, add more changes or at the end someone can accept the pull request in a pull request. You can see the conversation that commits that make up this pull request. There's two right now, you can also add some rules, you can set up continuous integration so that only changes that don't break the code can be merged and then you can add comments. Also, if you're not intending on merging this pull request, you can just click on close pull request additionally, I can see which files have changed. In this case, only one file, simple differences dot pi. Let me go back to the conversation and add a comment in here. So we typically work together on improving the code before merging. Let me just show you something in source three. As noted, I created a pull request but source three has no idea of this pull request for source three. There's only a couple of branches with several commits. Now let me go back to GitHub once ready I or someone that has enough permissions just clicks on confirm merge and just like that, the pull request was successfully merged and closed. Now, if I go back to source three, I can see the merge right here merch. Pull request number one from X Morera develop. Now when I click on pull, my main branch has been synchronized and now it has the changes that were included in develop in a nutshell. A pull request is just like a merch just that it's done in a collaborative way and only certain people may be able to take care of the pull request. One note is that when you're working with open source, you typically first fork the repository and then open a pull request. When that pull request is accepted, the commits from your repository are transferred to the main repository, the upstream repo what it's called. And that is how pull requests work.

### Conflicts

Now, let's talk about merge conflicts. I'm going to save stage and commit. But please note that I'm not pushing the change just yet. There it is. I committed locally. My main branch is one commit, thead of Origin Maine, but I'm going to go to Origin Maine. Select the same file. Now, origin main has one commit and it happens to be in the same line as I commit it locally. As you can see, first three tells me that there's one commit that I need to pull and that there's one commit that I need to push the problem is that both of those commits modify the same line. So how does sorcery or know what to do? Well, when I pull, it doesn't really know what to do. It tells you that you have a conflict that you need to resolve this one right here. As you can see, here are the less than markers with head which tells you that that was the change that you did locally. Then you can see the greater den markers with the hash those are the changes that were made in that commit, which is in the repository. If I go to Visual Studio Code, I can see the conflict and there are two options right here. I could sort this out manually or I can use the GUI functionality. Visual Studio Code provides this functionality, you accept current change or accept incoming change or both or compare them. But I'm showing you in sorcery. So when I go to sorcery, what I need to do is I select the file with the conflict and select resolve conflicts. I have several options using mine is my commit bears is the commit in the repo or I can restart the marriage mark is resolved or I have a few other options. But typically mine and theirs are the ones that you need to use. Unless you want to resolve manually. I'm going to use mine which is going to overwrite the change in the other repo. And here's the message for the commit, merge, branch main and the conflict I click on commit. And now when I push, I can see that I have resolved the conflict using my changes. As mentioned, there are different ways of resolving conflict, whichever one you use, always make sure that you don't lose any existing functionality.

### Cleaning Up

Now, just a few final notes on some best practices that I would like to share regarding the mechanics of Git with a GUI. First of all, it's recommended that you delete merged branches, that is after merging a feature or bug fix branch into the main branch, it's a good practice to delete the merged branch. Next, it's recommended to prune remote branches that remote branches that have been deleted on the remote repository may still exist locally as stale references, so remember to delete these branches as well. Additionally, discard uncommitted changes, that is if you have made local changes that you no longer need, you should discard them, but be careful when you discard changes as you can't recover the changes. And finally, keep a clean commit history. Basically, what this means is that you should try to maintain a clean and organized commit history by making logical and atomic commits. Remember what I said, commit early, commit often, and each commit should represent a single logical change and should have a descriptive message. Avoid including unnecessary changes or unrelated modifications into the same commit. Having said that, let's do the takeaway for this module.

### Takeaway

In this module, I covered the mechanics of Git, which are a small number of actions which are the ones that you use most of the time. Most of these actions are available in SourceTree, but a few like pull requests take place in your Git hosting platform. These actions include a clone, which is how you get code from a remote repo, and SourceTree also allows you to add an existing local repository or create a repo, even with the option to create the repo in the remote. Then you have fork, which is a way to create a copy of a remote repo in the Git hosting platform for you to work on, something that's very common with open source projects. Then we talked about the actions starting with commit, which is arguably one of the most important ones. A commit is a snapshot of a repository at a point in time. It's basically how you save your work. To understand commits, you need to know about the sections of a Git project which include the working directory, staging area, and the .git directory, and there are different states for the files, they could be modified or untracked, staged, and committed. And always remember, when you're committing, always write good commit messages, very descriptive and concise, and always, always commit early, commit often. Then I talked about push, which is how you transfer your changes to the remote repository. Fetch is how you get changes from a remote repository, but you don't merge these changes. Then pull, which is how you get changes and merge them. Next up, I talked about stash, which is a way to save changes locally for future usage. You just apply the stash when required. Then discard, which is a way to revert to the last committed state. And ignore, which provides a way to specify which files should not be tracked. Basically, you provide a pattern to follow. Then I talked about branch, which is an independent development line of work, it's how you isolate work or features, and as mentioned, it's Git's killer feature. Then merge, which is how you combine commits from different branches. Then I talked about the pull request, which is a feature to propose changes that's not native to Git. If a pull request is accepted, then a merge takes place. This is commonly used in the open source development world. Then I talked about conflicts, which is what happens when changes cannot be merged automatically by Git. You can resolve a conflict manually or using a GUI tool. And finally, always remember to clean up after you're done. And with this, we complete this module. I invite you to join me in the next module to take Git to the next level with a GUI.

## Git: To the Next Level (With a GUI)

### Git to the Next Level (with a GUI)

Welcome to the next module of this training, Git to the Next Level with a GUI. In this module, I will explain a few actions that may not be used on a daily basis, but they can be very helpful in certain scenarios and may even save you when you're in trouble, they're not terribly hard nor advanced. This is not a Git internals module, but I promise that what you're going to learn in this module will be useful pretty quickly. Let's do this.

### Merging and Rebasing

Once you have completed your work, it is necessary to merge it back. That's how you collaborate. However, there are several different ways of merging branches in Git. Which merge strategy to use depends on your desired outcome and specific scenario. So, let's talk about merging versus rebasing. Just in case, let me remind you of a few things. When you merge, you integrate changes from one branch into another, which creates a new commit that has two parent commits. The commit represents the merge and it preserves the history while combining independent lines of development. This is what we've used so far, but there's another alternative, rebasing which is also a type of merge, but what it does is that it applies a branch changes onto a different base commit. It does not create a merge commit, which means it does not create a merge commit and it modifies the commit history, which means that the hash that identifies the commit. The SHA might change, making it so that it appears as changes were made onto the latest commit of the branch that you're rebasing onto. Here's a question, why is it called rebase? Well, the answer is pretty simple. You select a new base commit to serve as the new starting point as the new base. Let's talk about the pros and cons of merging and rebasing, starting here. Merging is simple to understand and separates the commits. I'll talk about fast forward in just a few moments, but for now, let's just think about the regular merge that we have, and you keep the original context. On the other side, you have the pros for rebasing, namely it simplifies history, it's intuitive and clutter free, and helps you combine commits from multiple developers. Now, let's see the disadvantages of merging. If merging comes as a need because there are multiple people working in parallel in the same branch, then there will be no useful historic purpose, and it will create some clutter. On the other hand, with rebasing, it's slightly more complex, especially in terms of conflicts, and it's confusing if you push your commits. More than confusing, it can cause some trouble because the hashes will change. Let's do a demo on merging versus rebasing. I'll start with the normal merge, which is the one that I've shown you before. Here, I have two branches, main and develop, and please note that I have not pushed my changes just yet. With merge, this is not an issue, but with rebasing it is. In this case, I'm going to merge from develop into main. So, from the main branch, I click on Merge, I select develop, and I just check that rebase is not selected, it is not, so I click on OK. And there it is, regular merge. Now I'm going to go back in time so that my repository is in the exact same state. Well, I didn't go back in time. I deleted the merge commit, which I'll show you how in just a few moments. And now, let's do rebasing, but remember the golden rule of rebasing. Never rebase on public branches. This means basically that for rebasing, you need to make sure that you haven't pushed your changes yet. I'm going to follow the same process. I am in my main branch, I select Merge, and I select which is the branch that I'm going to merge, develop, in this case. And now, I select Rebase instead of merge. Look at the warning. Make sure you haven't pushed your changes. I click on OK, and now, as you can see, there's no merge commit. My changes have been applied linearly. This is easier to understand and less clutter‑free. And now if I wanted to, I can now push my changes. Just to show you, let me put them side‑by‑side. Here on the left, I have the regular merge, and on the right, is where I rebased. As you can see with rebasing, it looks a lot better, but you have to be careful.

### Fast Forward Merge

Let's now talk about the fast forward merge, which is a special type of merge. Namely, when the branch being merged has no additional commits since the branching point. Like in this case, the pink one is the main branch. The purple one is developed. In main, there are no new commits since the diverging point. In this case, Git moves the branch pointer of the target branch forward. In this case, it would be main to the latest commit of the branch that is being merged. This causes the target branch to fast forward like this which creates a linear history. Let me show you with a demo. In here, I have a branch called data‑feature that has a couple of commits after its diverging point from main. Now, let me switch to main, and I'm going to merge data‑feature into main, so I click on Merge and I select data‑feature. And now, let me first show you what a regular merge looks like. In this case, SourceTree is going to try to do a fast forward merge by default, but I can modify its behavior. So I click on Create a new commit, and when I select it, now I have a merge commit, merge commit being the default behavior, except in a scenario like this. Now, let me go back to the state where I was, data‑feature two commits ahead of main. Now I'm going to click on Merge, and when I select, I'm not going to click on the Create new commit, which means that SourceTree is going to do a fast forward by default. When I select this, I can see that the main pointer is just moved forward, which means I successfully did a fast forward merge. And now, as usual, I can push. Let me put them side‑by‑side. On the left, I can see where I specified that I wanted to create a new merge commit. On the right, it was a fast forward merge. As you can see, it's much clearer to do a fast forward merge because it creates a linear history.

### Detached Head

Next up, let's talk about the detached head, and I agree that's not a very nice name, but let me explain. The head in Git refers to a symbolic reference to the currently checked out branch or commit. It represents the tip of the branch or the specific commit that you have currently selected as your working state. It is how Git keeps track of what's considered the latest commit in your repo, and a detached head is a specific scenario where the head pointer is pointing directly to a specific commit rather than a branch reference. In this state, you are no longer working on a branch, but rather on a specific commit in the repository's commit history, and it can occur if you check out a commit, a tag, or a remote branch that's not associated with a local branch. When in a detached head state, any new commits will not be associated with an existing branch, which means that changes are lost if you switch to another branch. This is important because you do not want to lose commits. The good news is that it's possible to recover from a detached head state by either creating or switching to a branch. Let me show you with a demo. Here I am looking at my repository's history. I'm going to move SourceTree a little bit to the right to show you the files in my working directory. This is main and head is pointing to this particular commit as well. Now, if I double‑click on a specific commit, I will check out this commit which creates a detached head, and SourceTree tells me about this scenario. It even gives me the option to discard all current changes. Again, this is potentially troublesome as you may lose some work. Anyway, I'm going to click on OK, and as you can see, it looks like I traveled back in time as I have a lot less files in my working directory, and you can see here that the head pointer is pointing to this commit, which as mentioned, is not in any branch. If I go up, I can see that I have a few uncommitted changes because there's a file that I ignored in the next commit. As you can see, SourceTree lets me know that these uncommitted changes are in a detached head, which means that if I commit a change and then I switch to a branch, I will lose this change. To recover from the detached head, I can just pick a branch and a change, and as you can see, my files are back to its original state. So here's the question, why did you want to go back to that specific commit? Maybe that you want to create a new branch and add some files? Well, instead of a detatched head, what you need to do is you need to create a new branch, so you select the commit, you click on Branch, give it a name, and you select which commit you want to create a new branch from. I select this particular commit, I click on OK, and now I do have a branch. Now I can start working on this branch, and my changes will not be lost.

### Blame and Search

There are two functionalities that are very useful from time‑to‑time which can get you out of difficult situations. I am talking about blame and search. I'll start with blame which shows what revision an author made certain modifications to a particular file. In fact, it annotates each line in a given file. However, it does not show deleted or replaced files. With blame, you can determine who made changes which may help you understand the why. Let me show you. I am looking at my history, and in this commit, there's a particular file that I want to see who made some changes, so I select it, I right‑click, and I select Blame, which gives me the history of the changes made to this file. If there's one change in particular that calls my attention, I can select it, double‑click, and it will open the log which shows me the changes that I can understand by looking at the diff and then search. It is almost definitely certain that you're going to need to find older commits, file changes, or work done by authors for which you use search, which as mentioned, helps you with commit messages, file changes, and authors. Let me show you. At this point, I have History selected, but on my WORKSPACE section on the left, I can click on Search, and now I can specify what is it that I'm looking? I type data. This is the commit where I added the data to my repository. In this case, I'm looking for the commit messages, but I can also search for file changes and authors. For example, let's look for any file changes for else. These are all the changes where I committed a file change with an else statement. As I always say, search is one of the most useful functionalities in every system, and this is how you search.

### Cherry Pick

There's a phrase that I hear all the time, "Small commits allow for big wins". Short phrase, but extremely accurate. Let me tell you why. If you create a commit that implements two features or performs two different fixes or combines different tasks, then it might be hard to identify what went wrong if there's an issue, or if you're maintaining several versions and you need to move a particular feature or bug fix from one branch to the other, again, it's not easy. On the other hand, if you create small atomic commits, then taking your fix or new feature and moving it to a new branch is much easier. Basically, you can move specific commits without the need of a full merge. That is what's known as cherry pick, which allows you to move a single or a specific commit or commits from one branch to another, but maybe you're wondering why not simply a merge? Well, maybe you don't want all features and fixes found in a specific branch. Even worse, there may be incompatibilities or the code is in an out‑of‑date branch. Let me show you a demo on cherry pick. Let's start as usual in the history, and I'll show you the files on the left. I'm going to switch to this early branch, the one that I created in the detached head video which only has one code file. I'm going to remove this file right here to have a clean working directory. Okay. Now, here I am in the early branch, and I want to add a specific file, this one right here, the pandasimport.py, so I click on it and select Cherry Pick, and what this does is that it's going to apply that specific commit onto my checked out branch. I click on OK, and now I can see that this commit has been applied. It's also worth noting that the hash ID is different as it is a new commit, and this is how you cherry pick to apply one commit onto a different branch.

### Rewriting History Rebasing Interactively

Let's talk about a very important topic, Rewriting History in Git. As I mentioned before, Git is designed to never lose a change, and on top of this, with the caveat that you usually only add, you don't modify anything that you already added in Git. However, given its power, it's also possible to rewrite history, which you only do usually on your own branch to prepare for a push. To rewrite history, you would use what's known as rebase interactively. As expected, SourceTree will run underneath all Git commands by presenting to you an easy‑to‑use UI where you can do things simply with drag and drop or a few clicks. When you rebase interactively, you can modify and reorganize commits within a branch history. You can combine, edit, delete, or reorder commits, which allows you to create a more coherent and polished commit history. And don't rebase on public branches because it might create issues for other people that are working in your repository. Also. rebase interactively is not recommended as a normal action. In the upcoming videos, I'll show you how to use rebase interactively with SourceTree to delete commits, edit commit comments, amending commit contents, and reorder commits. The one thing that you need to remember is that when you're rebasing interactively, you do not rebase on an existing commit. You rebase children of an existing commit. Let me show you.

### Deleting Commits

Before I show you how to delete commits, remember that in Git, you typically only add new commits. However, it's possible to delete commits using the reset subcommand, which you can do individually or you can do using rebase interactively. In this video, I'll show you how to delete directly, but remember that it's something that you should avoid unless absolutely necessary because you're rewriting history, and do not delete when you already pushed your changes as it may cause plenty of confusion for other developers that are working on your repository. If you need to delete something that you already committed, you may use revert instead, which I'll show you soon. Let's now talking about deleting a commit with the reset command. There are several options. First, the soft delete, hard delete, and then mixed. What's the difference? With the reset soft, what happens is that the commit or commits are deleted, but the changes will be left into index ready for a new commit. Then you have hard, which means that the commit or commits are deleted, and all files are taken back to their original state. It is a destructive operation. And then mixed means that the commits or changes are deleted, and your files will be left unstaged. Let me show you with a demo. Okay. Before deleting a commit, I'll start by deleting a branch, which is another way of deleting a commit. I'll simply delete the branch, and there it is. I am now in main, and what I'm going to do is I'm going to show you the three different types of delete. For this, I'm going to add two new changes because if I do a delete right now, I'm going to do it in a remote branch. I'm going to do it in a local branch. I'm going to add these two files, close.py and delete.py, and I'm going to commit them. There you go, adds close and delete code. Now, I'm going to show you the files on the left. Here, you can see those two files. Now, how do I delete? Well, you cannot select the commit that you want to delete. As you can see here, there's no option to delete, but if I select a previous delete, for example, this one, I right‑click and I'm going to select Reset current branch to this commit, and I get this dialog with the three options, soft, mixed, and hard. I'm going to start with soft, and just like that, the commit has been deleted. My history has been reset to the latest commit that I have pushed, but close.py and delete.py are still there, and they are staged. Now, I can commit them again, Adds close and delete code. Now, I'm going to do the same thing. I go to the previous commit and select Reset current branch to this commit, and I'm going to do a mixed reset, which as you can see, the files are still there, close.py and delete.py, but they are unstaged. Now, I'm going to commit them again, and I'm going to do it one last time. This time, it's going to be a hard reset. I select hard, click on OK, and the commit has been deleted, and as you can see, the files have been removed as well. As I mentioned, remember that this is a destructive operation, so use it with care. And finally, don't do it on branches that you have already pushed because a developer may have their tip of their branch in the commit that you deleted and this will wreak havoc.

### Editing and Amending Commits

Next up, let's talk about editing and amending commits, which are to frequently use functionalities when you need to rewrite history. Let's start with editing a commit comment. There may be times where you need to change a commit comment. With SourceTree, it's easy to rewrite history, but as mentioned, do not edit unpushed commits, and aside from being able to edit the comment, you can also change the content that's a snapshot of a commit by adding or removing files, but here's a disclaimer, because you're changing the contents, it's going to change the commit identifier, the SHA, which you have to use it with here. To amend the commit, you select from the commit options and check Amend latest commit. Let me show you both of these functionalities together. Here's my repository. Here's the last commit that I have not pushed Adds running procs code. I just realized that I had to add another file over here. This one, psutils, so I copy it over here, and now, I have an uncommitted change. I stage this file, and from the commit options, I select Amend latest commit. Now, what's going to happen is that psutils is going to be added to the previous commit. Actually, a new commit is going to be created that contains the contents of the previous commit and this one. Let me show you. I click on Commit and there you go. Now psutil\_files and running\_procs have been both added, but I forgot to change the message, but I don't get any option here to edit the message. How do I do it? Well, I select a previous commit and select rebase children interactively. There you go. I select the commit, click on Edit Message, modify the message, click on OK, and just like that, I have amended a commit and modified the commit message.

### Reordering Commits

From time to time, you may need to reorder your commits, which is something that you can do in a very straightforward way from sorcery. Basically by dragging and dropping from the Rebase interactively interface, just take into account that any changes are not done until you click. OK. Let me show you with a demo. And of course, remember, don't reorder commits if you have pushed your changes. Anyway, here I am. My main branch is two commits ahead, which means that I can select the previous commit and select Rebase interactively. Here it is, I can drag and drop or I can use the buttons down here right now. Open code is the latest commit. But when I click this, it's next to last. Now I click. OK? And immediately the commits have been reordered. Just one thing. Take a look at the hash I DS they changed. And that is one of the main reasons why you don't reorder commits on changes that you have already pushed.

### Squash Commits

Next up, squashing commits, which I think the name suits it perfectly. As I mentioned earlier, a very good piece of advice is to commit early, commit often, but this may mean that you end up with lots of commits, many of them being work in progress, so you can squash multiple commits to create a cleaner history, which you do from the rebase interactively interface, and the important part is that you provide a commit message that gives an overview of all the commits included in the new squash commit, for example, includes completed slides for module three, and well, some people prefer to squash, others not to squash. I leave this decision up to you, but let me give you my point of view. When you do a squash commit, the whole point of squashing is that most people will not care about file revisions. They care about the overview what the bundle of commit does like which bug is fixed by applying this commit or what feature gets done. However, other people prefer not to squash, as I mentioned, up to you. Now, let's see a demo on squash commits. Here, I have three commits that I have not yet pushed Mod1, Mod2, Mod3. Yes, the commit messages are not very good, but I was in a hurry to complete this feature. Now, here's what I'm going to do. I go to the commit previous to this ones and I select Rebase children interactively. Now I'm going to select Squash with previous once, and then I'm going to drag and drop once more. Now, I have three commits that have been squashed into one. I can edit the message if I want it to provide a better message, and when I'm ready, I'm going click on OK. And just like this, I have one new commit that contains the three previous commits, and that is how you squash commits.

### Reverse Commit

Now this is something that you need to learn, reverse commit. Why? Well, mistakes can be made that may require reverting changes, and maybe when you think about a mistake, your first impulse is to delete, but as mentioned in Git, it's better to avoid deleting commits because it rewrites history. What you should do is to reverse commit instead, which is a safe method. It undoes a committed snapshot by creating a new commit. Let me show you. Here, I have my latest squashed commit, which I have not yet pushed. Let me put the files right here. Down below, you can see the modules.py. Now, I'm going to right‑click and select Reverse commit. Are you sure you want to create a new commit reversing all the changes in the select commit? Well, yes. I select this and a new revert has created, which as you can see, deletes this file. It basically undid all the changes that was done by that particular commit.

### Tagging

Let's talk about tagging in Git. Tags, as I mentioned earlier, are objects that are used to label specific points in the commit history of a repository. Using tags, you can mark important milestones, releases, or versions. They provide a way to reference specific commits for easier retrieval and identification. There are two types annotated and lightweight. Lightweight tags are like an immutable branch. They're a pointer to a specific commit, while annotated tags are full objects. They have a hash, tagger name, email, date, and message, and can be signed and verified. It is the recommended method. It is possible to tag anywhere and you can also push your tags to the remote repository. Let me show you a demo. Here's my local repository. I'm going to select a specific commit, right‑click and select tag. I then provide a tag name, for example, data version, and in this case, I selected a specific commit, but you can also select the working copy parent. If you need to change to a different commit, simply select here and select where you intend to create the tag. I'll click on Cancel. Also, you can push your tag be it to the origin or another remote repository. This, in particular, is an annotated tag. Within the advanced options, you can create a lightweight tag, and you can also specify if you want to move an existing tag. I'll click on Add tag, and just like this, my tag has been created. You can see the list of tags here on the left, and in this case, I specified that I wanted to push my tag, which I can see over here in the releases. I can select the tag and here it is, data version. If I select it, I can even download the source code at this point in time. Tags, it's what usually is used when you want to create releases. And just like this, I have created a tag.

### Custom Actions

There may be times where you need to perform a particular action, but not all actions are available in SourceTree. Because of this, SourceTree has with some functionality that allows you to extend the range of actions which is achieved by creating scripts or commands. You specify which information is required in the form of variables, and then you create a custom action. All you need to do is open the custom actions dialog and indicate which executable or script you want to run, specifying the parameters. Custom actions gives you all the flexibility that you need in case your particular action is not available in SourceTree.

### Takeaway

As a takeaway for this module, there are many actions that are not commonly used, but are quite useful in certain scenarios for which I started talking about rebasing, which is a way to create a cleaner history, a linear history. When should you use rebasing? Well, basically when you have not pushed your changes yet. Don't rebase when you have already pushed your changes as that may cause problems. Also, the fast forward merge where commits are applied linearly when merging. You can do this when no new commits have been added since the branching point, else it will be rebasing. Next, we talked about the detached head, which happens when the head pointer points directly to a commit rather than a branch reference. If this is the scenario, new commits will be lost if added, then we talked about blame, which is a way to show revisions and who made changes to particular files. This excludes deleted and untracked files. Then I showed you how to search or commit messages, file changes, or authors. I then showed you cherry pick which is a way to move a commit or many commits to a different branch. Then I talked about rewriting history, which is possible, but should be done with care, and you never rewrite history on public branches. That is those that you have already pushed. I showed you then how to rebase interactively, which allowed you to edit commit messages, amend commit contents, reorder commits, and squash commits. Then I showed you how to delete commits using reset of which there are three types, hard, soft, or mixed and show you what you should use instead of deleting the reverse commit, which creates a new commit that reverts changes. I then showed you how to tag, which is used to mark important milestones and told you of custom actions which is a way to perform Git actions that are not included in SourceTree. And with this, we complete this module. I invite you to join me in the next module to talk about branching strategies.

## Branching Strategies for Every Occasion

### Branching Strategies for Every Occasion

And now that you're aware of the mechanics of Git and learn about a few more advanced commands, but without getting into the low‑level stuff, it's time to talk about branching strategies and how important it is to have one, and you may be wondering why is it important to have a branching strategy? Actually, what is a branching strategy? A branching strategy is nothing more than a set of rules and guidelines that define how branches are used in a project. This strategy defines how branches are created, named, merged, and managed throughout the development lifecycle. Using branches and having the right branching strategy is important because it allows for better collaboration, while at the same time, providing a way for your team to do development in parallel and allowing isolation of work. I know all of these sound like they are completely opposite and different things, and yes, they are, but the point is that branches allow you to do things in different ways as needed. Also, branches make it easy to experiment and create prototypes. If it didn't work, then don't merge. If it does, well, you merge, and branches provide an excellent way to do release management and perform rollbacks and bug fixes. You can maintain and update older versions easily. Anyway, check out what I have for you in this module.

### Centralized Workflow

Let me start with the centralized workflow. In this workflow, you use Git as a centralized repository. This does not change the fact that Git is distributed. However, you'll use Git in the same way as you used other source control systems that were centralized, which means that all team members make changes directly to the main branch. No other branches are used, which is a strategy that works very well for small teams because it's easy to communicate and coordinate among team members. The centralized workflow is the easiest way to get started when transitioning from other source control systems. It gives developers that are not used to Git with enough time to get used to it. The way it works is that developers, they just do their thing normally. When their work is ready, they stage and commit locally and then push the commits to origin, and there are just a few things to take into account. First, they need to get all changes before pushing, that is pull the changes. They need to resolve any conflicts if they arise, and Git will not allow overriding commits, which means no work is going to be lost. Then they rebase, and they push their commits. And I don't know if you noticed, but I just showed you the secret to making it all work. Rebasing is required. Why? Because rebasing keeps the history linear. If you just do a merge commit, it will create branches. And that's not what we want with the centralized workflow. Using the centralized workflow is easy to get started. There are a few disadvantages of using it. First of all, is the code in main always ready for deployment? Not usually. Also, how do you manage work in progress on a feature? Or even worse, how do you collaborate on a feature without having your system not being stable? How do you prevent broken code in main? That's a pretty hard one if you have a linear history, and these caveats are also present in other source control systems. They're not specific to Git, so they can be resolved, but it's not the best scenario. Anyway, let's say that you're working with a centralized workflow and you have everything under control, but Git can do more. Let me tell you about the other workflows.

### Feature Branch Workflow

Next up, the feature branch workflow. The feature branch workflow, which is very straightforward, works like this. Whenever there's a new feature that's required, a dedicated branch is created, then one or many developers can work on this specific branch, which in fact encapsulates the work. It provides a clear picture of what is being worked on and does not affect the main branch. It also allows for work to take place in parallel, that is if a user is working on a feature and a really urgent request comes in perhaps the VP of development requested a change to be done right now, then the developer can create a new branch, complete the bug fix, push it, and then come back to the original feature branch. In this way, the feature branch workflow enhances collaboration. Once the feature is ready, then it could be merged or a pull request can be created, and when the work has been reviewed, everything is merged into main. At this point, the feature is complete. In summary, let's say that new work comes in, so a new branch is created for this particular feature, so then everybody that's involved work and commits when done, a few actions take place which involve pushing, pulling, and merging and then work keeps happening. And of course, multiple feature branches can be created and worked at any point. Let's summarize how it works. One thing that's important at this point is to talk about the distinction between merge and pull requests. Well, the distinction and similarities. When you merge, you incorporate changes from your current feature branch into your main branch. At this point, it is required that the feature branch is up to date, which may involve pulling from the main branch to your feature branch. This includes resolving any conflicts, and once it's ready you merge commit or even you can fast forward merge. With a pull request, you do a request to merge where some permissions may be required to perform the merge. Notifications are sent to those that are in charge of reviewing and merging, some collaboration takes place, which means usually a code review or you check the new changes and then potentially discussion of the proposed feature. If everything is good, then it's approved and now the merge takes place. And here's a very important recommendation. Always use continuous integration and automated tests to validate changes in feature branches before merging. This can avoid you a headache or two.

### Gitflow Workflow

Every once in a while, someone sits down to share with the world something that can change the way things are done. Well, this happened in 2010 when Vincent Driessen shared with the world his blog post, a successful Git branching model, thus creating GitFlow workflow. I encourage you to read it if you have a chance. In my humble opinion, it's brilliant. Let me tell you why. The GitFlow workflow is great for complex projects, but also works great for small project, it organizes your workstream in a brilliant way by simplifying complex source code management by having separate branches or features, hot fixes, and releases. This allows you to work in parallel. And one of the most important points being that it makes a distinction between development and production ready. Thus, you always have stable versions to release when the need arises. My only question now is how did I live without it? And even better, SourceTree supports it out‑of‑the‑box. Let's talk about the main branches. There are two, main and develop. Origin/main is the production branch. It is always production ready and it's a stable branch. Origin/develop, it's what's called the integration branch. It is where the latest development changes are merged into. If you do them, nightly builds are done from this branch. The nightly build is the automatic build done preferably daily with all recent changes that are being added. Usually, it is deployed and tested either automatically or manually, and the key is that only completed features are merged. It is supposed to be pretty stable, but not production‑ready like main, and once you're ready to start a new production release, you merge as stable/develop into main via a release branch, and then you tag. I'll explain release branches in just a second. These two branches continue in an infinite lifetime. Neither one is deleted. They're just used as integration and production branches. As mentioned, these are the three supporting branches. They're not special, they're just like any branch. There are some rules to follow to make this work like a charm. They are named after the way they should be used. Feature branches are used to work in a specific feature or bug. Release branches are used to prepare a specific release, and hot fix, which are fixes to the current production releases. The advantage is that teams can work in parallel without disruption by using this model. Let's learn about feature branches. Well, it's very straightforward. Each feature branch or bug has their own branch. Every branch always starts from develop and exists while development takes place on the specific feature or bug. It is usually recommended to have the branch in the local repo, but perfectly fine to upload to origin as a backup and for working with other developers, you know, collaboration. Once you're ready, you merge. Feature branches can also be called topic branches, and the rules are that you start by branching off from develop and merge back into develop once development has been completed. They can be named anything, except main, develop, release\* and hotfix\*. Then the release branches, they are used to support preparation of a production release. They are created when production code is almost ready and as stable as possible, they're used for last minute fixes without changing code base, and you can update your metadata, for example, a release number. While you have a release branch, work can continue undeveloped, the integration branch, and when all is ready, this means you have a stable release, then you merge back into main to take back any minor last‑minute changes. You tag the release, and finally, you delete the branch. The rules are that they are branched from develop and must merge back into develop and main. The naming convention is release‑\*, that is the star represents the name of the release. And then hotfix branches, they're used to prepare a release, but an unplanned release, basically you're reacting to fix a bug or a few while the team keeps working on feature fixes or features. And once you have fixed the bugs and you're ready, you update metadata and you merge into master and develop, except if it originated from a release branch, in which case, you merge back into this release branch. And finally, you delete this branch. Let's talk about the rules. Hotfix may branch from main in which case, it must branch back into develop and main or potentially into the release that it originated from. And the naming convention is hotfix‑\*. The same principle applies. The star will be the desired name if you're hotfix while you're working. Again, this is a convention, no harm done if you just simply give it a name. In smaller projects, it might be totally okay. In bigger projects, it's much better that you follow the convention. Here I am in SourceTree. I created a new repository, it's called ps‑gitflow. It also exists in GitHub. Okay, so now, let me direct your attention here where it says Git‑Flow. SourceTree has the GitFlow functionality fully integrated, but when I click on this, this will initialize the repository for GitFlow. If I wanted to, I can change some of the names. For example, my production branch is going to be main. Now, I'm going to click on OK, and my repository has been initialized. At this point, this only means creating a branch from develop, the integration branch, which is where I am right now. Now, whenever I'm ready to work, I just click on this button and I have several options. At this point, these are my available options. I can Start New Feature, Start New Release, Start New Hotfix, or other action. I'm going to start a new feature, going call it pycode, and there you go. A new feature branch has been created which you can see over here. Okay. I've just added this file. I'm going to commit it and OK, that's the first change in this feature branch. In my case, I'm going to finish the feature. So I click the button and then click an OK? And now my work has been merged into develop. Please note that I didn't have to merge my work source street did this for me. Now, at any point, I can start new features, new hot fixes or in this case, I'm going to start a new release. I give it a name and a new release branch has been created this point. I can do things like update the metadata version 0.1 and I commit all good. Now my feature branch is ready. I have updated my metadata for which I click the gate flow button and finish release. I also create a tag. So I'm going to finish the release. I give a name for the tag and now I click an OK? And just like that I have finished the release. As you can see here, Maine has five changes that need to be pushed once I push them, I have completed this release.

### GitHub Flow

Next up, the GitHub Flow, which is the branching strategy used by GitHub themselves. It's a simplified version of the GitFlow workflow. It's a straightforward approach with fewer branches. Here's how it works. The main branch is always production‑ready code with the feature branch being created for specific features or tasks, it branches off from main, and once it's completed, it's merged back. There's no releases branch. So it goes like this, you create a branch, you add your commits, you open a pull request, discuss, and review, and finally, merge and deploy. If GitHub themselves uses it, why don't you?

### Forking Workflow

Let me now mention a few things about the Forking Workflow. Forks being something that I already mentioned. The Forking Workflow is commonly used in open source projects where contributors clone the original repository, that's they create their own fork, the fork being a copy of the original repository under their own account, and then contributors do their work in their own repository. Once it's ready, they propose changes back to the original repository via pull requests. This provides a structured and collaborative approach to contribute to a project. Let me show you a quick example. Here's an open source project, the Roslyn .NET compiler. In this case, if you look at the top, there are almost 4,000 forks. If you wanted to create your own fork, as I mentioned, you can do it like this. Anyway, I'm not going to create a fork right now. What I want to show you is the pull requests. Each one of these pull requests is a contribution by someone. For example, let me open this one, and in this pull request, I can see that it's from this fork. This person basically worked on making a few changes. Once these changes were ready, the contributor opened a pull request with these changes. Once the pull request is reviewed and approved, it can be merged. The Forking Workflow, as mentioned, is typically used in open source. However, it can also be used in your company if the need arises. And now, let's do the takeaway for this module.

### Takeaway

As a takeaway for this module, there are multiple branching strategies, and you should use the branching strategy that fits your team best. The centralized workflow being the easiest way to get started, especially if you come from other source control system as it has the smoothest learning curve. Once you learn the ropes with Git, you can evolve into using the feature branch which starts to take advantage of many of Git's features. It is quite a nice improvement over the centralized workflow. Then, Gitflow Workflow, which is brilliant as it allows to have a great source code management of your project by separating development for production branches and providing a very organized and clear management way for the use of release, hotfix, and feature supporting branches. Then GitHub flow, which is similar to Gitflow Workflow, but has fewer branches, and it's the branching strategy used by GitHub. And finally, the Forking Workflow which differentiates itself by having multiple server‑side repositories registered under different usernames, having them all originated from a single repo. This is what's commonly used in the open source, but can also be used by companies where multiple departments share common code, but have slightly different requirements. And now, let's do the final takeaway of this course.