## Getting to Know Git

### Welcome to Git

Welcome to Git! In case you never use Git and don't know what it is, let's start with the definition. Git is an open source distributed version control system that, wait, now sorry, this sounds so complicated. Let me scrap this slide and start with a simpler definition. Git is a tool that tracks changes to source code. And it's very basic. That's what it does. It tracks changes to any file, actually, but it's mostly used for source code files. And if you put it like this, it doesn't sound like much. You wouldn't expect something that tracks changes to source code to change your life or the life of a software team, but it does. For most software people, using Git has a lot of interesting consequences that we'll explore in the rest of these big picture training. More specifically, here is what we'll talk about. This is a beginner course, so we'll start from scratch. In a minute, we'll have a quick demo of Git. You see how the most basic features of Git work in practice so that you have something concrete to anchor the following discussion. Then in the next model, we'll get the broader picture of version control. That is this idea of tracking changes to source code. Git isn't the only version control tool around, even though it's the most popular one. So in the following module, we'll see what makes Git special and different from other similar tools. If you already have experience with version control, then this is where the important information starts for you. And finally, we'll have a module about the Git ecosystem, in particular about online services like GitHub that are so important these days. So, that's what you'll get in these four modules. All modules are less than 20 minutes, sometimes much less, so this is a quick training. You can binge watch it in a single sitting if you wish. So make yourself comfortable, and let's start with that demo of Git's basic features.

### Putting a Project on Git

So, Git is a tool that tracks changes to source code. That is a version control system. Let's see a short demo of the very basics of Git, and I'll give you that demo in a terminal window. I have it here. I'll use the command line. Maybe you don't normally use the command line in your job. In that case, no worries. I'm using the command line here because I want you to see the commands and get used to the vocabulary of Git, but you certainly don't need to memorize the commands. It's even possible that you'll never use Git from the command line. Maybe you'll always use it through a graphical interface integrated in Visual Studio, for example. So don't feel like you have to memorize this stuff, just follow along. And to make things more clear, I'll also visualize what I'm doing here on the right side of the screen. So let's say that I'm in this directory that contains an open source software project. In this case, the project happens to be a popular JavaScript framework, but that doesn't really matter here. I just went to the internet and downloaded the source code of the first open source project I could think of just to have a real example. And I'll throw three files here on the right side of the screen to represent the files in this project. I'll just throw you three, but it's actually many more, and main directories, too. Let's see. On the operating system I'm using, this command shows me the entire tree of files and directories, and see, there are thousands of directories and files in this project. Some files, are source code, some are documentation, some are configuration, maybe, like any non‑trivial software project. Now, I want to use Git to manage this code. I already have Git installed on this system, so the first thing I'll do is to give this command, git in it. That means put this project under Git. In practice, I just created something called a repository or, a repo, for short. That's a Git storage area. For now, don't worry about the exact location of the repository. We'll talk about it later. Let's say it's somewhere on my computer, and it's still empty because I just created it. And what the repo does is it can contain snapshots of my project. Let's see how

### Creating Snapshots

Let's create our first snapshot of this project. That's something that you do in two steps. First, we must tell Git which files from this project will go into the snapshot. I'll do that with this command, git add. And because this is the first snapshot that we create, it's a good idea to include all the files from the project in it. So I'll use this dot. That means this directory where I am now. So I'm telling Git to include the current directory in the snapshot and all its files and subdirectories, the entire tree. I sent this command, but note that I didn't create this snapshot yet. This was just the first step. Let's use this metaphor. Let's say that we have a launchpad to the repository, and what we just did is we put some files on the launchpad. This launchpad is also called the index in Git. In this case, we put all the files in the project in the index. And now, step two, we can send the content of the index straight to the repository with this command called commit. So, Git commit means create a snapshot that includes all the files that I put on the launchpad earlier, and we can give a message to the commit to describe what's in it, say, "First commit with all the files." And when I send this command, fire. The files get to the repository, and they become the first snapshot of the project inside the repository. So we just created a snapshot of the entire project. And if I use this command git log, then we see the history of the project so far that right now it is only this single snapshot, this single commit in the language of Git. Now, this commit isn't very useful until we have at least another commit. So let's imagine that we have to do something that happens all the time in software development, we have to fix a bug. I'll use my text editor to edit a file in this project, let's say this file. And let's change this line here to fix the bug. Actually, I'm just doing a random change. Most likely I just created a bug instead of fixing it. But let's just pretend that I made this change to fix a bug. Now this file has been changed. It's been modified since the last time we took a snapshot, a commit of the project. And in fact, if I ask Git for the current status of the project, it tells me, look, this file has been modified. It shows the modified file in red. This is already a mildly useful thing, right? This fact that Git tells me which files have been modified since the last commit. So let's create another commit, but this time this commit only includes the changes since the previous commit. That is, this file. So, once again, two steps. First add the file. Let's say it's this blue file in the diagram. Put it on the launchpad, and if we do git status again, Git shows me this modified file in green to mean this file is ready to go in the next commit. And second, we create the commit with a message 3, 2, 1, lift off. And now we have a second commit, another snapshot that includes this changed file. In fact, git status shows nothing because everything in the project is aligned with the latest commit. And if I ask for the log, you can see two commits, one from a few seconds ago that includes the latest changes, and an earlier one that includes all the original files in the project. And it also shows us who created these commits. That's me. So, we are already tracking the history of the project at this point. For example, we can ask Git, okay, so what changed between these two snapshots? That's a command named git diff that shows the difference between two commits. This command needs the names of two commits, and one way we identify commits in Git is by using their unique code, their version, if you wish. These long hexadecimal codes, Git generates these codes automatically when we create a commit. We actually don't need to use the entire code. We can get away with just the first few digits. Like, give me the diff between these, I'll copy paste it here, and this. And this command shows that the only thing that changed between the two commits is that in this file, this line changed. So now we can find out what changed in the project and who changed it and when exactly. And that's already a very useful concept, right? Especially in a large project. But that's only the beginning; it gets better. Let's see!

### Traveling Back and Forth in Time

One of the most important reasons to use version control is time travel. Let's say that for whatever reason, I want to go back in time to an earlier version of the project. There are many possible reasons we might want to do that, but let's say, for example, that we got yet another bug report, and we realized that maybe my bug fix that is in this commit here had side effects and it caused another problem, so we want to track down this new bug. And to do that, we want to go back to an earlier version of the project before the bug fix. Now, in this demo, I only changed one line, so we could roll back the change by hand, but imagine a real project where a single commit might involve changes to dozens or hundreds of lines of code. It would be unfeasible to revert the fix by hand. Instead, we can use Git to do this with a single command that says hey, Git, take me back in time to this version of the project. That command is called git checkout. And checkout takes the version of the project that we want to go to, this commit. So I'll copy paste this commit. And when I do this, Git takes the content of that commit and puts it in our file system. If we only look at this terminal, it seems that nothing happened, but if I edit this file again, you see that my earlier change is gone. The file is back to its original state. We roll back time to the earlier version of the project. And now I can do all the testing I want. And when I'm done, I can check out the latest commit again and roll time forward again to the latest version of the project. And if we check the file again, our earlier change is back in. We're back on the latest version of the code. So thank you for staying with me. That was the demo of the basics of Git. I'll introduce other feature of Git later on, but those require a bit more context. So let's stop here and recap what we've seen in this first module.

### Recapping the Demo

In this module, we've gone through a demo of the basic version control functionality of Git. We took a software project and we put it on Git, so to say, by creating a Git repository. Then we created two snapshots of the project, two commits, to use the Git term. Each commit is a different version of the project. One contained all the files in the project, and one contained only one change to a specific file. And we saw that Git can show us these two commits additional information on them, like the name of the author and the differences between them. And even better, we traveled back and forth in time moving freely between these two versions of the project. Those were the basics of version control with Git. And if Git did nothing else for us, I think it would already be worth using just to check the history of the project and travel to any point in that history. It would already carry its weight. But it does more than that, much more, in fact. Version control is a game changer in software development, and in the next module, we'll see why.

## Understanding Version Control

### Tracking a Project's History

Welcome back. In this module, we'll look at the big picture of version control. We've seen that version control is basically about creating snapshots of files, but that simple idea has far‑reaching consequences, and we're going to talk about them. Public service announcement, we're going to talk about version control in general, not specifically Git. So if you are already familiar with any version control system, you might decide to skip to the next module, if you wish. Maybe just watch the clip called managing multiple versions of a project because I will mention a few Git‑specific commands in there, and feel free to skip the rest if you're in a hurry. Are you still here? Good. So let's see what makes version control so essential to software developers. The most obvious benefit of version control is that it gives you a detailed history of your project. As an example, I have a long‑running open source project here, and here is its history. See, we have all the changes, their messages, the authors, and dates, everything that happened. This can be useful for any kind of project, not just software projects. I used version control to produce this training, for example. In software projects, however, this history is even more important because source code is so complicated and hard to deal with. So a history like this can help you debug your code, for example, or to revert broken changes. It's like having an infinite undo or redo function. In fact, sometimes I wonder how did we ever live without version control? Well, most people did something like this. This is a very old project of mine, a computer game. Back then, I didn't use version control, so I periodically copy‑pasted the project folder to this backup directory. See, some of these directories are named with a date or with a version of sorts. And another thing I did was sometimes I didn't delete code, even useless code, because I was afraid I would need it again later. So I just commented it out, like I did here. And as you can imagine, the result was a big mess. I was not 100% sure which of these folders contained what, my code was littered with commented out functions, and ironically, sometimes I still lost important information because I got confused by all these copies. It wasn't a great development experience. These days, I put all my projects on version control. For example, I told you that I put the material for this training on Git. Here it is. If I want to recover an old slide or a file that I deleted, I can just go back in time, and there it is. If I'm wondering where a change happened, I can ask Git. It's just so convenient. Just, to be clear, version control is not a backup, at least not by itself. For example, yes, I put this training in a Git repository, but that repo is still on my laptop, so if my disk fails, then I lose both the project and the repo. So I have a separate backup strategy to keep this project safe. Version control can be useful to recover from a disaster in some cases. But in general, it's not like you put a project on Git and you stop worrying about backups. It depends on where that Git repository is exactly. I just had to make myself clear on that. So, tracking your project's history, that's the first and most obvious reason to use version control, but there are more.

### Managing Multiple Versions of a Project

A second benefit of version control is that you can use it to juggle multiple versions of a project at the same time. Why would you want to do that? Here is a quick example. Imagine that you are developing a smartphone app called Marathon Trout. It's a running tracker for fish. You put this project on version control and commit new features. Each orange circle here is a commit. And every few days, maybe you release this app for free on some app store. Then at some point, you decide to have a second paid version of the app with premium features. So now you have two streams of development, a free version and a premium version, and they share most of their code and they usually change together, except that sometimes they don't. So, for example, these orange commits are the same in both streams. There are features or bug fixes that are common to both versions. But these green commits are premium features, and they're only in the premium version, not in the free version. Without version control, this is a complicated situation. I guess you could maintain two completely separate projects and copy‑paste code from one to the other, but that's a lot of copy‑pasting. It would be easy to make mistakes, almost guaranteed, actually. But version control systems have a solution for this. It's called branching. Let's go back to the moment when you decided to have a premium version of the app. At this point, you have one stream of development. In Git, this main stream is usually called the main branch. It can have any name, but main is one of the most common ones. What you can do is to branch out another stream from the main branch. There is a git branch command that does that. So if you create a second branch, let's say a branch called premium, and work on the two branches in parallel, after a while you have a situation like this. I'm visualizing the commits on the premium branch in green to make them easier to tell from the orange commits on the main branch. So now the main branch contains the free version of the app and the premium branch contains the premium version, and you can switch from one branch to the other with a command that's called git switch. So if you say git switch premium, you move to the premium branch. And with git switch main, you move to the main branch. That contains the free version of the app. So this is how you maintain two streams of development, but that doesn't really solve the problem yet, right? Yes, now you have two branches, but how do you share code between them? For example, let's say that these commits in the free version of the app are bug fixes, and they also need to go into the premium version. How do you do that? What you can do is to use the counterpart to branching that's called merging. You do it like this. You move to the premium branch, and then you say git merge main, and see what happens to the history. Now, the history of the premium branch includes everything, including all the green premium commits, and also the orange or regular commits, while the history of the free app on the main branch includes the free commits, but not the premium commits. So that's what branching and merging looks like. Just a very quick introduction because this feature deserves its own training. But even if that's all you know about branch and merging, if you think about it, you'll already realize one thing, that is, branching is easy, but merging can be hard. For example, what if when you merge these two branches, you find out that you have two commits into two branches, say, these two commits, that change the exact same line of code, but in different ways? Now you have a conflict, two conflicting changes that need to be reconciled somehow. Git cannot do it on its own. Only you can because you understand what those changes mean. And in fact, as you learn how to use Git, you'll see that merge conflicts do happen frequently. But even then, it's way easier to solve those conflicts in a version control system than by just comparing two directories. It's actually one of the reasons to use version control to make this kind of conflict a bit more manageable. So that was one example of why you might want to maintain multiple versions of the same project. But in practice, there are many other situations that call for that. You can probably think of a few from your own experience. There are always many opportunities to use branching and merging in a project. In fact, branching and merging aren't useful only to maintain multiple versions of a project. They actually enable what I think is the primary reason to use version control. Let's talk about that.

### Sharing Code Amongst Developers

Writing software is hard, and writing software in a team is even harder. Actually, sometimes I think it's a miracle that people manage to work together on the same code at all. As you know, code is finicky. One little mistake and you get a bug. And it's all too easy to make those mistakes even when you have full control of your own code. So, how can multiple people work together on the same code without stomping over each other's toes all the time? And indeed, before version control became widespread, people tried really hard not to work on the same code. They did that in a few ways that usually involved splitting the project into multiple parts, separate modules. For example, a team writing software for inventory management might have split the project into the code to access the database, the user interface code, the administration interface code, and so on. You tended to have one developer for each piece of the project, and they worked pretty much in isolation until it was finally time to put all the pieces together. That was the infamous integration phase, and it was usually a bloodbath. It wasn't uncommon to see a large project stuck in the integration phase for many months as people tried desperately to make each piece work with each other's piece. These days, almost all teams use version control, and that makes it much easier to work on the same code, mostly because version control gives you the repository, and the repository is a shared authoritative version of the project that you can use at any time to get the latest changes and integrate your own changes. Once the developers have a share of the repo, they don't have to wait for a final large integration, they can integrate much more frequently. Different teams integrate their code in different ways, but most teams use some form of branching and merging convention to do that. For example, one common technique is to create a temporary branch for each new feature or bug fix. Let's say that we have two developers, Michelle and Aaron. Michelle is writing code for a new feature, so she creates a new branch for that from the main branch, and she starts committing code to this branch. And let's say that Aaron is fixing a bug, so he also creates a branch for that and commits code to that branch. But let's say Aaron's job was quick, so he's already done, and he merges his changes back into the main branch. And then Michelle finishes her feature, and she also merges her new feature into the main branch. Now, if you look at this diagram, you might notice that there is a moment when there might be a conflict between the changes of Michelle and Aaron. And that's here, where Michelle commits her own changes. If she and Aaron happen to change the same area of the code, then she might have to solve conflicts at this point, to put our code and Aaron's code together. So, it's like she's going through a mini integration step when she merges. And you might think, if integration is so hard, then what's the point of doing it so often, almost every time a developer commits changes to a shared branch? If anything, shouldn't we try to do integration as little as possible? But it turns out that if you integrate more often, yes, you might still get conflicts, but those conflicts are likely to be small and easy to fix. Martin Fowler put it well by saying, "If it hurts, do it more often." Integration is often painful, even with version control, but as a rule of thumb, the more frequently you do it, the less painful it is, and version control allows you to do it very frequently. Some teams integrate their code many times per day, even many times per hour. So that's maybe the most important benefit of version control. It makes it much easier to integrate code. That ultimately means that it allows modern software teams to work together. In the modern world, you usually don't have the luxury of waiting for a long integration phase to complete. Many projects deliver new versions all the time, and that kind of frequent release is made possible by version control. And finally, there is also another facet of coordinating teamwork with version control, and it's important enough that it deserves its own clip.

### Coordinating Teamwork

So we said that once you have a repository, all the developers can use it as a source of truth to get the current state of the project. But that's not only limited to developers; it extends to the entire team, managers, testers, system administrators, and even all the tools that the team needs to keep the project under control. For example, most software teams use a bug tracker to manage a list of bugs or changes that need to be done. In these days, bug trackers integrate directly with the source code in the repository. So, for example, if a tester finds a bug, he can link the bug to the commit where it happened, and the developer fixing the bug can link the bug to the commit where she fixed it. And the same kind of integration happens with tools that, for example, help you manage the backlog of work that needs to be done, or maybe allow a team leader to allocate work to team members, like Jira, for example, that's a popular tool to track work on a team. You can link a task from these tools to the repository, and that makes it so much easier to understand what's happening in a project. One essential tool in many projects today is the build machine, also called the automation server sometimes. For example, there is a system called Jenkins that is very popular for that. The build machine also connects to the repository, and it typically springs into action whenever the repository changes to do many useful things. First, it typically builds the project. It compiles the code if it's written in a compiled language. It packages the code for distribution, maybe deploys the system on some test machine, and so on. And also it can monitor that the code is healthy. It runs metrics, and, above all, automated tests. It's common for software teams today to have many automated tests, and running all those tests can take a long time. So the build machine takes care of that, and it raises a flag if your latest commit breaks something in the system. That's an extremely useful thing. Some teams even go farther than that and use their build machine to trigger what is called continuous deployment. That means that the build machine not only checks the code or runs the test and so on, it literally rolls out the system to production. So every time you change anything in the source code, that change gets to production automatically. Only a small percentage of teams are able to pull that trick, but some do. It requires a lot of automated tests, of course, but that's an example of the power of this concept of a build machine. And all those things can only happen because the build machine knows every change to the system. They can happen thanks to version control. So, that's the final benefit of version control that I wanted to tell you about. It's not just the center of human communication in a team, it's the focus of all development these days. It's crucial to coordinate teamwork.

### Recapping Version Control

This module explained why version control is so essential to modern software development. First, it allows you to track the history of the project and move back and forth in time. That feature alone makes your life so much easier. You don't need to maintain copies of a project by hand, you can delete obsolete code with confidence. And if you get confused, you can sort out what happened in your project and when. A second way that version control improves your life is that it allows you to manage multiple versions of the same project. We made an example with an app that has a free version and a premium version. But there are many reasons why you might want to have multiple streams of development in a project. And a version control system gives you branching and merging to deal with those multiple streams. Maybe the most important reason to have version control is that it allows you to share code amongst developers and to coordinate the effort of teamwork. And when you start doing that, version control becomes the center of your project, not only for developers, but for everybody working on the project. It becomes the authoritative source of the current state of the project for all the people and tools that gravitate around the project. Developers, testers, managers, back trackers, and the all‑important build machine. So, I hope I convinced you that version control is important. But Git isn't the only version control system on the market, so why do so many teams today use git specifically? What makes it special? That's the subject of the next module.

## Making Sense of Git

### A Few Reasons to Like Git

What makes Git special? Why do so many teams use it? In this module, we'll go through a few benefits of Git and a few shortcomings as well. In particular, we'll talk about distributed version control. That's the killer feature of Git. To start with Git's strong points, one of the first things you'll notice when using Git is it's fast. Most Git operations are so swift, they feel instantaneous, branching, checking out a commit. I won't claim that Git is the fastest version control system out there, but it's certainly very fast. And another strong point of Git is that it's pretty smart in many practical cases that are traditionally hard in version control. For example, if you rename a file or move it to another directory, many version control systems might get confused by that. Not Git. It just keeps tracking the file's history without a flinch. Git is also flexible and powerful. This is something that developers love. You almost never have to worry, oh, maybe I cannot do this. You can do almost anything with your Git repository. It's yours. If you follow the more advanced Git trainings here on Pluralsight, you'll find that you can do crazy stuff, like refactor a project's history to clean it up. How does that sound? Git is a power tool. And finally, I want to mention, Git is safe. If your data is in a Git repo and if that repo is backed up, of course, then that data is safe both from accidental corruption and from intentional corruption. For example, a hacker who wants to sneak malicious code into your repository. So Git is both safe and secure. It's rock solid in that respect. So those are a few benefits of Git, but they still don't explain why so many organizations, even large companies that are traditionally conservative, are flocking to Git. I would say that for most people, migrating to Git means migrating to a different way to do version control, from client‑server version control to distributed version control. Let's see what that means exactly starting with an overview of client server version control.

### Client Server Version Control

Before Git came onto the scene, most people use the style of version control that you can call client‑server version control. For example, one of the most popular client‑server version control systems is called subversion. I still use it sometimes. I'll show you an example of the typical subversion workflow. As usual, you shouldn't feel like you have to remember all this stuff, but it is important that you get a feeling of the core ideas of this way of working. Meet Sonia. She's a developer, and she just joined the team that's using subversion. So the first thing that Sonia does after joining the project is she connects to the project's subversion repository. Client‑server version control, the repository is on the server somewhere in the building, or maybe in the cloud, and all the members of the team connect to it, and also the build machine and the other entities that need access to the project. There is one repository server to rule them all. So, Sonia connects to this repository, and she copies all the files from the project to her computer into her local working area. That's how it's called, her local copy of the project's files. As usual, I'll show only three files for simplicity, but imagine a typical project with many files. And she starts working on those files. Let's say that she needs to fix a bug and she changes one file to fix it. I colored it orange to mean that it's been changed. And now Sonia wants to commit this file, send it to the server. Not so fast, so because that's the downside of working in a team, it's possible that while Sonia is fixing the bug, one of our team members changes this same file on the repo. For example, maybe Frank edited this same file for some other reason, maybe to add a new feature. So now the file under repo isn't the same file that Sonia edited, and she cannot just send this file to the server; otherwise, she'd override the changes from Frank. She's got to fix this conflict. So what Sonia has to do is first get the latest changes from the server to her computer. Second, fix the conflict, that is, merge her changes with the changes from Frank, resulting in a file that has both her changes and Frank's changes. And third, she can finally send the entire thing to the server and create a commit. So it's a 3‑steps process. So that's the basic workflow of client‑server version control. Of course, there might be complications and variations and branches and all this stuff, but you've just seen the basic loop. And I call it client‑server version control because there is a server that contains the repository, and many clients that are the developers' workstations, and also to build machine, and so on. Now, let's see how distributed version control works.

### Distributed Version Control

So, distributed version control. This isn't an exclusive feature of Git. There are other distributed version control systems around, but Git is by far the most popular, and also the one that made this way working mainstream, so to say. So most people identify distributed version control with Git. Let's see what Sonia has been up to. She just moved to another project in the same company and this other project use Git. What does she do differently this time? The first difference between working with subversion and working with Git comes immediately the moment that Sonia connects to the project's repository, in that she doesn't just connect to the repository, it's that she does something called the clone. There is a git clone command for that, and what that means is that Sonia copies the entire repository to her machine, all of it, project history and all, including, of course, the latest commits that is the current files in the project. In fact, if you want to see the repository of a Git project, you usually don't have to look very far. Here is that project from the demo I gave you earlier on. Back then I told you not to worry about the repository, that it was somewhere. As it turns out, it was right under our noses. If I give this command that on my operating system is show me the content of this directory, including the hidden files and directories, you see that there is a hidden director here called .git. And this directory contains the Git repository, including the entire history of the project. So, more often than not, a project's Git repository is right there inside the project directory. So now that Sonia has her own repo, she can do anything she wants on it. She can create commits, branches, check out earlier versions of the project, anything she wants. And at some point, she probably want to synchronize her changes with the changes from the rest of her team. How does that work in a distributed Git project? What happens is that even though she has her own repo, Sonia maintains a link to the shared repo. That's called a remote in Git's terminology. And she has additional commands to get the latest commits from the remote repo to the local repo. That's called the pull. Or the other way around, that's called the push. And just like what happens in client‑server version control, Sonia might have to solve conflicts during these operations. In that case, she'll do a [ull first to get all the shared changes, then merge those changes with our local changes, and finally push her local changes to the shared team repo. Just to be clear, even if the team has a shared repo, that doesn't mean that Sonia's repo is a second‑class citizen. A Git environment is a peer‑to‑peer system, where there are usually multiple repositories that are all technically equally good, except that, of course, there are conventions, and most teams agree by convention that there is a copy of this repo on a shared server, and that's the important one that is visible to the entire team. And Sonia's own copy is less important, and maybe it doesn't get shared to other developers. But otherwise, they're just multiple copies of the same project history. In theory, you might not even have a shared server. You might just have a network of developer's repositories that are each other's remotes without any server that's officially the center of our repo. I've never seen people do that in practice, but it could be done. So, that's the idea of distributed version control, this idea of having multiple repos that are essentially peers and sometimes are connected as each other's remotes. And when I learned Git for the first time, I was a bit confused. I was like, okay, so I get the idea, but how is that better than just having one center of our repo? Doesn't, it just makes things more complicated? And it took me some time to understand the advantages of distributive version control. One of those advantages is pretty clear, that is, if Sonia has her own repo, she can still work with version control, even when she disconnects from the central server. She cannot share code with the rest of the team, but she can still create commits or check out branches, move back and forth in history, and so on. That's a nice feature. It plays well with the remote working, and it also removes the single point of failure that is the center repository. But I would say that the greatest benefit of distributed version control goes above and beyond this idea, working while disconnected from the remote server. It's a benefit that is at the same time more subtle and more powerful. And it's all about the flexibility to work in ways that are different from the traditional workflow of client‑server development. Sometimes a bit different, sometimes radically different. In the next and last module of this training, you'll see what I mean by that. We'll talk about the workflow used by most open source projects today that would be simply impossible with client‑server version control. But before we move into that, let's close this module with a few reasons not to use Git.

### A Few Reasons to Dislike Git

I gave you plenty of reasons to love Git, so you might wonder, are there any reasons not to love it? Any downsides? And of course, there are. First, let me get rid of a couple of reasons to avoid Git that in my opinion aren't very valid. They are more like myths than valid concerns, in most cases. First, some companies are wary of distributed version control because they believe that you cannot control access in a distributed system. That is, once you have distributed the repos, every developer can change any file. You cannot say, for example, this file is important, only the team leader can change it. And also you cannot lock files. Locking is a feature of some version control systems where you try to prevent merge conflicts by owning files, by saying this file is mine for a while, I'm working on it, and nobody else can edit it. You cannot do that with a distributed system. And that lack of access control, some people believe it's a recipe for chaos. Well, I don't think that's a valid concern. Plain and simple. Yes, distributed version control is different. It has different ways to deal with access. But I think that those different ways are generally more efficient and more effective than the traditional access control model. For example, with Git, you can empower a team leader to accept or reject any changes, like open source projects do. We'll see more details in the next module. And yes, you cannot lock a file and you cannot say nobody edits this file but me, but that's a good thing, in my opinion, because file locking often gets in the way of people working. And in practice, it's almost always better to solve conflicts when they arise than try to prevent conflicts by locking files. So I don't think that these fears of Git not giving you access control are justified, frankly. And while we're talking about myths of distributive version control, another thing that scares people away from Git is this idea of having a copy of the project's history on each developer's computer. I've heard people say, hey, we have a huge project in this company with a long‑running history; it would be madness to copy that history to all the development workstations. And what I found is most of the time, that turns out not to be a problem in practice. If you have a huge project with a long history and don't want to copy that history to your repo, Git gives you a few ways to deal with that. You can break down the project into multiple smaller repositories, or you can use a feature called a shallow clone that allows you to distribute only a partial history of a project, like the last few commits. So in practice, that's really a big problem. After all, the Linux kernel uses Git. The Microsoft Windows project uses a modified version of Git these days. That's over 3 million files, thousands of developers. So I'd say if Git is good enough for them, then it's probably good enough for most large projects. Those are the myths, but there are also a couple of good reasons to be wary of Git, real shortcomings. One is Git is great at dealing with text files like source code, but pretty bad at dealing with large binary files, like video or big data files. If your project includes many of those, then you might want to keep them off the repo, or just use something other than Git. Otherwise, they might slow down your repo and take up a lot of disk space. To be fair, that's not only Git; most version control systems aren't that great a dealing with large binary files. But Git is maybe worse than most. There are a few version control systems that specialize in binary files, and some of them are even built upon Git. But I wouldn't use plain vanilla Git to store many large binary files, let's say that. And finally, in my opinion, the most important reason to avoid Git, it's not a technical reason, it's a people reason. That is, Git is pretty tough to get into. First of all, distributed version control is harder to understand than client‑server version control by its nature. And on top of that, Git wasn't exactly designed to be user‑friendly. It reflects the personality of its creator, Linus Torvalds, the same guy who created the Linux operating system. Torvalds is a contrarian; he likes to do things his own way, so he got to the extreme of using the same words that other version control systems were using with a different meaning. For example, the words commit and checkout mean different things in Git than they mean in subversion, and that can be really confusing as you move from subversion to Git. In general, Git takes some getting used to. You're not going to master it in an afternoon, and it won't give you warm fuzzies as you learn it. That being said, if you're willing to give Git a try anyway, once you get over that initial hump, I think that Git is worth it. Let me put it like this. I found Git hard to learn, but now that I learned it, I find it easy to use. I do realize this is a controversial statement. I think that Git is one of those tools that look really complicated, but once you understand how it works, everything clicks, and it suddenly looks clean and consistent, and it gives you this great feeling of power of mastery. I have a training here on Pluralsight called How Git Works that's all about that, and as soon as you gain some confidence with Git, I hope you'll watch that training. And once you get there, I bet you will fall in love with Git. I did!

### Recapping Git's Pros and Cons

In this module, we went through a few benefits of Git. It's fast, it's smart in the sense that it makes some common operations like renaming files, for example, as easy as they can be. It's very flexible and powerful, and it's solid, it keeps your data safe. Then we went into a discussion of Git's killer feature, distributed version control. We compared it with traditional client‑server version control. Distributed version control has far‑reaching consequences, so we can't explore all of them in this training. But in the next module, you'll see an example of a popular distributed workflow. And finally, we looked at Git's shortcomings, a couple that I think are not very relevant, they are more myths than real concerns for most projects. And we also mentioned two limitations that I think are more worth considering. The fact that Git isn't very good at storing large binary files, and the fact that Git isn't very friendly at first. It requires some training. In the next module, we'll look at the Git ecosystem, and in particular, we'll see a concrete example of the power of distributed version control. That's essential information.o see you there

## Entering the Git Ecosystem

### Integrating Git

There is an entire ecosystem of technologies revolving around Git. Other tools that communicate with Git, graphical interfaces, online services, and so on. We're going to talk about that ecosystem, but actually that's going to take us a couple of minutes, tops, except for one specific item in the Git ecosystem, and that is Git‑based services, like GitHub. Because they are a great example of the power of distributed version control, so we'll mostly talk about that in this module. So let's get the rest of the ecosystem out of the way first because that's going to be a quick discussion. It can be wrapped up as if you need to integrate Git in your current tools set. Chances are you won't have many problems doing that. Git is one of those happy situations where one of the best technologies is also one of the most popular, so it enjoys all the support you might need. For example, starting with the tools that a developer uses daily. If you want to use Git in a development environment like Visual Studio or Eclipse, no problem, those come with Git integration right out of the box. And if you prefer a modern text editor, like Visual Studio Code, Vim, or Sublime, they integrate with Git as well. And also, if you want to use Git on its own, but you're not exactly jumping at the chance to use the command line, there are plenty of standalone graphical interfaces to Git, like SourceTree from Atlassian, and just name the wrapping. You know, there are just so many tools that integrate with Git or provide a graphical interface to Git. And also Git jives well with all the team‑oriented tools like the automation servers, like Jenkins or Travis CI, that is the build machines, the issue trackers like Jira, communication tools like Slack and Microsoft Teams, and so on. And then there are the online services, all the big cloud providers support Git, Microsoft Azure, Amazon AWS, Heroku, you name it. Git is a welcome citizen in pretty much any environment these days. And finally, there are the Git‑based services like GitHub, GitLab, and Bitbucket. And now that's when I should stop just enumerating logos and get into a more in depth discussion.

### A Quick Look at GitHub

There are a few Git‑based online services. They tend to work in a similar way, so I'll use GitHub as an example for all of them. What I'll tell you about GitHub is also true for other services, like GitLab and Bitbucket. GitHub is such a popular tool these days because it's useful in many ways. The first and most obvious is its hosting service. For example, here is my GitHub homepage. Let's say that I need the Git repository for a new project. I click here. I give a name to my project, and boom. I have a Git repository in the cloud. And I can clone this repository to my computer. This is a public repository that GitHub gives me for free. If I want a private repo, for example, for a commercial code that shouldn't be visible to everybody on the internet, then it depends on the service. Some services also make that free with some limitations. Others require me to pay for it, you know, your standard hosting service policy stuff. GitHub also provides project management tools, like issue tracking. So here is a popular open source project, and there is an Issues tab here. This project has plenty of issues, and here anybody from inside the project or even a user of the project can open an issue and people can discuss these issues. And GitHub gives you the means to connect this discussion to the project's history. For example, you can refer to a specific commit. And it gives you many tools like this to track issues, to integrate with external services, for example, build machines, code metrics systems, and so on. In fact, these communication features in GitHub are so powerful that you can almost describe it as a social network, only center don't code more than people. If you look at my GitHub home page, you can see that these here are my friends or colleagues. These are their activities, contributions to projects, discussions they're having, and so on. So, GitHub is a repository hosting service, a set of project management tools, and even somehow a social network that focuses discussions around projects and code. And those are all the reasons why it's popular. But the most important reason why people use GitHub is that it builds upon Git to give you a specific workflow for collaborating on software projects. That workflow is even more important to many projects than the hosting and issue tracking and integrations. Let's talk about it.

### The Open Source Workflow

Let's look at this workflow that is very popular these days. You can call it the open source workflow, even though it's not just for open source, as we'll see later. My usual disclaimer, I don't expect that you'll memorize this information. I won't go into the technical details in this training. I just like to give you a concrete example of a way of working that's only possible with distributed version control. You'll probably have many opportunities to learn this stuff by doing it as you gain more experience with Git. For now, just relax and let's look at this way of working. Look who's back, Sonia, our developer friend. She's a passionate developer, so she wants to contribute to open source projects over the weekend. And she found a project on GitHub that could use a few bug fixes, so she plans to contribute to that. Let's say that the project belongs to a GitHub user named Pluralsight and it's called marathonfish. The GitHub convention is that our repo is identified by the name of the user or organization that owns the project/the name of the project. So it's pluralsight/marathonfish. So, Sonia creates a GitHub account. Let's say her GitHub name is nusco. That's actually my GitHub account. I'm using it here because I want to be sure that I'm not stealing some other user's name and dragging innocent people into my example. So, Sonia, or nusco, to use her GitHub name, wants to contribute to this project. The first thing she needs is her own copy of this repository so she can work on it. And maybe you remember from the previous module that she could clone this repo to her local machine, and she'd get a complete copy of the repo, history and all. But instead she does something slightly different. Let me show what and then I'll explain why. Instead of getting a local clone of the repo, Sonia asks GitHub to give her a remote clone inside the GitHub. This operation is called the fork in GitHub's terms. Every project you visit in GitHub has a button to fork it. It's this one. So when Sonia forks pluralsight/marathonfish, she creates her own copy of the project. So, nusco/marathonfish. And then she also clones this repo to her local computer to work on it, and she keeps nusco/marathonfish on GitHub as her remote. In a minute, I'll show you why this complication, why Sonia didn't just clone the original repo. So, now Sonia works on her contributions to the project. She fixes a bug and creates a new commit or a few commits in our local repo, although the commit says this little patch here. Then she'd like to push the commit to the original project, right? She wants her changes incorporated here. But that's when things get a bit more complicated because Sonia doesn't have permission to push to this project. It belongs to another user, Pluralsight. I mean, open source projects are public and old, but that certainly doesn't mean that anybody can just push code to any project. Instead, open source projects have a maintainer, a user, or an organization that decides which contributions get into the project. In this case, the maintainer is Pluralsight. So the problem here becomes as follows. Sonia has a new commit for the pluralsight/marathonfish repository, but she doesn't have the right access to that repo. So she'd like to submit her contribution to the Pluralsight user so that this user can hopefully roll it into the original project. But Sonia's contribution is on their private computer. So how can Pluralsight see those changes or even know they exist? And now we can see why Sonia created a fork of the original project. Because this fork can be read by anybody, but it can only be written by her, it's her copy of the project on GitHub. So she can push the changes to the fork, nusco/marathonfish. And then she can use the defining feature of GitHub that's called the pull request. That is, she sends a message to the Pluralsight user that says, look, I have a contribution here on this fork of your project, so why don't you come and get it? And Pluralsight can check the changes and maybe review them, maybe discuss them with Sonia, run the project's automated tests on Sonia's changes, and hopefully say, okay, I like these changes, and pull these changes from Sonia's fork into the original project. All those things can happen inside the GitHub interface. The entire process from the sending of the pull request to accepting it can be done in GitHub. And that's the open source workflow. It's actually a bit more complicated than this, in the general case. For example, Sonia might have to keep the original project as a secondary remote, as well as her own fork, so that if the original project changes, she can get the latest commits and solve many conflicts before she signs a pull request. So there are a few more wrinkles involved. But what you've seen is the core idea of this way of working. You fork a project, you clone it to your local computer, you commit your changes to it, push those changes to the fork, and then send a pull request to ask the maintainer of the original project to pull your changes. And hopefully they do. And if you think about it, this workflow can only exist in a distributed version control system. You cannot have pull requests in a client‑server system. You might remember, in the previous module I told you distributed version control has its own ways to control access, and this is one, and it works great. Some open source projects have thousands of contributors from all over the world, from core stable contributors to casual contributors who might send a single patch and then disappear. And even with all those people waltzing around the project, they still maintain tight control over what code gets into the project, arguably more controlled than the typical internal company project. And, in fact, many companies these days are adopting this way of working based on pull requests for their internal projects. So, I called it the open source workflow, but it's becoming more and more common for closed source projects as well. And that was my last example in this training. I hope it gave you a glimpse of the power of git and distributed version control.