### A Note About the Command Line

One public service announcement before we begin. In this training, I will be using the command line a lot, and if you are following along, so should you. I'm recording on a Mac, so you will see Linux‑style commands such as ls to list directories as opposed to dir in Windows and forward slashes in passed names rather than backslashes as in Windows. But if you're running a Windows environment, it should be easy for you to translate those commands to a Windows terminal. Most of the commands that we use barely even change from Linux to Windows. Now I know that many developers are a bit uncomfortable around the command line. They find it intimidating sometimes or just unpleasant. Maybe you prefer to use a graphical user interface tool for Git instead of the command line like the Git extension in Visual Studio Code, for example, or a standalone client like source 3 from a collection. And that's fine. I also like GUI tools for some operations. If I need to resolve a merge conflict or to compare two versions of a file, I use a GUI tool. So I'm not saying that you should use the command line all the time. However, I still recommend that you use the command line most of the time, especially when you are learning it simply because believe it or not, it's generally easier that way. A GUI tool will probably implement the most common operations and then fall short in those all‑important corner cases. And when that happens, you usually have to revert back to the command line anyway. I guess that what I'm trying to say is the command line is your friend. It makes it easier to learn Git because it gives you access to 100% of Git's features. And if you learn on the command line, then even if you switch to a GUI tool later on, you'll understand exactly what you're doing and you'll know that you can still fall back to the command line for those more advanced, more specific commands. That's it. Public service announcement done. Let's dive into the first part of our Git training and talk about the four areas of Git.

### Introducing the Four Areas

I was talking about the model earlier on. Here is the basic model that I'll use in this training. A Git project stores information in four separate storage areas. If you've been using Git for any amount of time, then you're probably familiar with at least three of these areas. The first of the four areas is the project directory on your file system, your working area. It's the place where you keep your current files and your current folders. The second area is the all‑important repository. This is arguably the main reason that you use Git in the first place. The repository contains the entire history of the project. When you commit stuff, it goes here. In between these two areas, there is another intermediate area called the index. It's the place where you put your files before a commit. If you want to master Git, then it's essential that you understand how to use the index. Finally, the fourth area sits a bit to the side. It's a temporary storage area called the stash. It's not nearly as important as the other three, but it's useful. And those are the four areas. Now, here is an important concept. If you want to really understand the Git command, then for most commands, you should ask two important questions. The first question is how does this command move data across the four areas? Does it copy data from the index to the repository, for example, from the repository to the working area? Does it delete any data from any of the areas? And so on. The second question you should ask is what does this command do to the repository specifically? The repository is the most important of the four areas. So, how does this command change the data in there? Does it create new commits? Does it move branches? Does it move the head reference and so on? It doesn't matter how confusing a command looks. If you can answer these two questions, then you will get it, at least broadly you will get it. And you can worry about the subtleties later. Let's recap. We have the working area, the repository, the index, and the stash. Actually, let's forget about the stash for now. We'll meet it again later. And let's take a deeper look at the other three areas.

### The Working Area

Let's talk about the working area and the repository first. These two areas are the subject of my previous training, How Git Works. So, this clip and the next one are essentially a super quick review of that entire training. The whole thing will take just a few minutes. Actually, if you haven't watched How Git Works, then you can take the next few minutes of video as a self‑evaluation test. If at any point, you are confused by what I'm saying here, especially when I talk about Git objects and branches, then maybe you need to understand a bit more about the internals of Git before you go on. No problem. Just go and watch How Git Works before you continue with this training. It won't take long. On the other hand, if this stuff sounds obvious to you, then just move forward. Okay, let's get started. The first of the four areas is your working area. That is the project's directory on your file system. This is where you work, you edit your files, you test your code and so on. The project I'm using for this training, a simple cookbook, has just a handful of files and folders. If I want to change anything in this project, then we probably start by editing a file or moving it or I don't know, creating it. All of these changes happen in the working area. So, the working area is all‑important to me. However, Git doesn't care as much about it. For Git, the working area is a very temporary place. Thankfully, Git will generally respect the working area. It won't go and destroy data in there. But in this training, we will see a few Git commands that do destroy data in the working area. So in general, don't assume that your data is safe until you have committed it. Once you commit your data, Git stores it in what it considers the really important area, the repository.

### The Repository

The repository is here in the .git folder. The most important data is in a directory that's called the objects database, here. There are a few different kinds of objects in the database. Let's talk about them. Some objects represent the content of a file at some point in the project's history. These objects are called blobs, and then there are other objects called trees that represent folders in the project. And also, there are commits. Whenever you do a Git commit, Git creates a commit. All of these objects are immutable. They can be created and deleted, but they can never be changed. These objects are linked together in a structure that represents your project's history. Each commit points to a graph of blobs and trees that represent your files and folders at the moment of that commit. For example, this commit is pointing at these blobs and trees, and this commit is pointing at these blobs and trees. So, each commit is like a snapshot of your working area at a certain point in time. Also, two commits can share the same objects. This means that these objects, in this case, for example, it's two files and a directory, these objects haven't changed between these two commits, and that is the way that Git stores changes to your files and directories. We will talk a lot about commits in this training, but we won't talk much about blobs and trees. So, let's keep blobs and trees off this picture and add a few more commits instead. In the history of a Git project, each commit is pointing to its parent commits. For example, this commit is pointing to this commit, which in turn is pointing to these commits, which are both pointing to this commit here. So, if you refer to the first commit in this chain, then you are also indirectly referring to all of these commits. Now remember that each commit is a snapshot, right? A phrase frame of your project's history, so to say. So, all of these commits taken together are a bunch of snapshots that is a slice of your project history. References to commits such as this orange arrow here are an important entity in Git. They are called branches. That's what a branch is, a reference to a commit. And because it references the commit and the commits are linked together to form a history, the branch is basically the entry point to a history of commits. And you can have multiple branches that are multiple slices of history. These are the commits in branch1, and these are the commits in branch2. A commit can belong to multiple branches. Finally, there is a special pointer called HEAD. There can only be one head. It's usually pointing to a branch, and that's the current branch. And the branch is pointing to a commit, so HEAD is indirectly pointing to a commit, right? And that's the current commit. If I move ahead, maybe because I switched to another branch like I'm doing here, then I switch to a different current commit. There is always one current commit. One last thing about commits, sometimes you can do operations that result in commits that cannot be reached from any branch. For example, if I delete branch2, then these two commits become unreachable. There is no branch pointing at them, either directly or indirectly. They are not part of any history anymore. So Git will eventually delete them, an operation that is called garbage collection. We will meet a few operations in this training that create unreachable commits. Wow! Okay, let me take a breath. That was it, our ludicrous speed or recap of the working area and the repository. As I said before, if you could follow this recap easily, then you definitely know everything that you need to move on with this training. If you couldn't, then consider stopping now and watching the previous training, How Git Works. And in that case, see you back in a couple of hours. Okay, so you're still here. Good. Let's move on.

### The Index

Now let's talk about the third storage area in Git, the index. The index is a very peculiar thing. Pretty much every version system out there has a working area and the repository, but the index is unique to Git or at least Git is the only version system that allows you to modify the index directly, as far as I know. You can visualize the index as something that stands between the working area and the repository. You generally don't move the data from the working area to the repository directly. You go through the index. That's why the index is also called the staging area. You stage the changes by adding them from the working area to the index, and then you commit the changes from the index to the repository. Let's look at this process in more detail. Okay, so we have these three areas. I will use the same cookbook project that I used in the previous training here. So the project already has a history of commits. But if you ask Git for the current status, right now there is nothing to commit and nothing new in the working area. Let's call this table situation the clean status, just to give it an easy name. In the clean status, the working area and the repository are aligned. They contain the same stuff. For example, let's go back to our project. There is a menu.txt file here and the recipes folder. And the folder contains more files, but let's ignore those. Let's make it simple. I will add this file and folder to the working area in the picture. The file and the folder have been committed some time in the past, so they are in the repository as well. There are two different kinds of objects. One is a blob, and the other is a tree, but that doesn't matter for us. Let's just say that they are two objects in the database. I will put them in the picture as well. Now I have to clarify one thing about the terminology I'm using. When I say that the file is in the repository, I usually mean something more specific. I mean that the file is in the current commit of the repository. Of course, the repository contains more stuff. It contains other commits and other files, the entire history of the project, in fact. So I'm playing a bit fast and loose with my language here. When I say the working area and the repository contain the same data, that's not exactly true. There is more data in the repository. I should say the working area and the current commit in the repository contain the same data. But to avoid repeating that every time, I will just say the repository instead of the current commit in the repository, it's fine as long as we understand each other. So, I will just visualize these two files, not the entire content of the repository. Okay, now what about the index? If you look at the content of the .git folder, you will see the index. It's this file here. It's a binary file. So we cannot just open it in a text editor, for example. What kind of data is in this binary file? Okay, this is where it gets a bit tricky. Right now, you probably think of the index as a transition area, a launch pad of sorts. In this mental model, the index is normally empty. Then you add the files from the working area to the index, you launch the files into the repository by committing them, and then the index is empty again. And in fact, that's pretty much how the index is implemented in Git. However, for the purpose of this training, I suggest that you ignore the implementation for once and you think of the index in a subtly different way. Think of it as just another area that holds everything, just like the working area of the repository, all your files and folders. So, when we say git status and we see this message, that doesn't mean that the index is empty. It means that the index contains the same files and folders as the repository. It's a small mental shift, but in many cases, it makes it easier to understand and visualize how data moves around in Git. So in this model, when we're in the clean status, the three areas are all aligned. They contain the same data. We can double‑check that with the git diff command. Git diff gives you the difference between two areas. One thing that is a bit counterintuitive about diff is if I use diff without any argument, then it's going to compare the working area with the index. Right now, they contain exactly the same data, so diff is empty. But usually, I don't use this diff style, not too often. The git status command is usually all I need to see the difference between what I have, that is my working area, and what I'm going to commit, the index. Instead, most of the times I want to compare the stuff I want to commit with the stuff I already committed. That is, I want to compare the index with the repository. For that, you can use git diff with the ‑‑cache option. Right now, the index and the latest commit in the repository are aligned, so this diff is also empty. Okay, we're all clean. And this closes our introduction to this training. Now that we've laid the groundwork, we're ready to start playing with the data. See you in the next module.

## Revisiting the Basic Workflow

### Revisiting the Basics

Welcome to the second module of Git Deep Dive. Good to see you again. In the previous module, I said that to understand the Git command, we must ask ourselves a couple of questions. How does this command move information across the four areas and how does it impact the repository in particular? In this second module, we're going to look at the basic Git workflow commands, the ones you already know, add, commit, switch, moving or renaming files, the works. Only we will look at these commands through the filter of those two questions. So even if you know those commands already, this might be a different way to look at them. Let's start straight away.

### Moving Data to the Right

You already know the basic workflow of Git, edit the file, stage the file, and commit the file. Let's look at those commands again, keeping the two questions in mind. To begin, I will edit data in my working area in the menu.txt file. I love Indian cuisine. I think it's amazing, so let's add something Indian or at least international Indian. There, we have Tikka Masala in the menu now. So I changed the menu file, and I will change its color in the diagram as well. And the status tells us that the file has been changed, but not staged yet. Okay. Now I want to copy the updated file from the working area to the index. You know the command that does that. It's git add. There you are. The file has been copied from working area to the index, overriding the previous version of the same file. Now git status tells me that the file is modified and staged, and git diff sees no difference. The working area and the index are aligned. But if I compare the index and the repository, then I can see the changes. They are ready to get into the next commit So let's do this, git commit. And as soon as we commit, the updated file is copied from the index to the repository, and now everything is aligned again. So in the course of these few operations, we moved data from left to right, so to say, from the working area to the index to the repository. And now as a result, the three areas contain the same data, no difference whatsoever. We are in the clean status again. Notice that the last command, commit, did more to the repository than just copying this file. It also created a new commit and other objects. It updated the current branch, all stuff that I'm not shown in this picture here. So commit moves data and also changes the repository. It's actually one of the most important commands that change the repository. By contrast, add just moves data and doesn't touch the repository.

### Moving Data to the Left

So, we've seen two commands that move data from left to right from the working area to the index, that was add, and from the index to the repository, that was commit. A quick question for you, can you think of a command that moves data in the other direction from the repository to the working area and the index? There are a few. One such command that comes to mind is switch, the command that you use to move to another branch. There are actually a couple of ways to move to another commit. You can also use git checkout, for example, to move to any commit, even if there is no branch on it. But if you want to move to another commit that is pointed at by a branch, then you generally use git switch. They work in a similar way, so I'll use a switch for this demo. Switch does two things essentially. In the repository, it moves the head reference, so it changes the current commit. And the second thing it does is it takes data from the new current commit, and it copies that data from the repository to the working area and the index. So it changes the repository first and moves data second. Okay, that was probably hard to follow. Let me show you how it happens, and everything should be clear. We have a few branches here. Let's look at the ideas branch. I can see the differences between that branch and our current main branch by diffing them. In ideas, the menu file is different, and other files in the recipe directory are also different. So what happens if we switch to the ideas branch? A couple of things just happened. The first thing, you probably already know about it, look at those files in the repository. There is a commit pointing at them, and it's the current commit. It's the current commit because there is a branch pointing at it, which is main, and HEAD is pointing at that branch. Well, the first thing that happened when we switched is that HEAD moved to the ideas branch like this. So the current commit changed. Where we are now, the files are different. I showed this in the diagram by using different colors for the files. As usual, I took a few shortcuts to make the diagram simpler. I ignored the many objects in the repository like the files in the recipes directory, a few merge commits and the like. I simplified the diagram to make it less busy. But otherwise, this is a good enough approximation to make my point. That is it's not that the data in the repository actually changed. It's just that HEAD changed, so the current commit changed, so we're seeing different data. So when we say the data in the repository changed, this is a shortcut to actually mean the current commit changed, so we're looking at different data in the repository. From now on, I will take these shortcuts for granted, both in my language and in my diagrams. And okay, that was the first thing that switch did. It moved HEAD. The second and last thing it did is it copied the current data from the repository to both the working area and the index like this. So now all the three areas have the same content, and we are in the clean status again. So commands like switch and also checkout change the repository because they move the HEAD pointer in there, and they copy data from the repository to the working area in the index. Okay, enough talking about switch. Let's go back to the main branch. So let's see. We have seen add, commit, and switch or checkout. You already knew these commands, but now we've been answering the two fundamental questions about each of them. First, how does this command move data across the areas? And second, what does this command do to the repository? So far, so good.

### Removing Files

Now let's talk about something less intuitive maybe, removing files in Git. We're starting from the clean status with a clean diagram. Let's ignore the existing files and directories. I will create a new file, a COPYRIGHT note for my cookbook project. Let's fill it in. I don't want anybody to steal my precious cookbook. There, now we have a new file in the working area. If I ask for a status, I see this file marked as untracked. That means that the file is in the working area, but not in the index or the repository. Git doesn't know what to do with it yet. I'm planning to commit this file. So I will use git add to copy it to the index. And now the status says that the file is new, which means that the file is in the working area and in the index, but it's not in the repository yet. It's not an official part of the project yet. If I commit it now, then the file would be copied to the repository, the three areas will be aligned, and I would be in the clean status again. But what if I change my mind instead? What if I want to remove the file from the index? Say I want to keep this file in the working area, but I don't want it in the index anymore. Maybe I want to commit something else first and get back to this file later. In other words, I want to go back to the situation I was in before the last add. I want the file in the working area, but not in the index. Now I use the add command to copy the file to the index. So you might think that I can use Git's remove command, rm, to uncopy the file from the index. Remove sounds like the opposite of add, right? Well unfortunately, that's not what remove does, or better, that's not the only thing that remove does. If I used it without any option, then I would try to delete the file from both the working area and the index. This could be pretty destructive. So remove has kind of a security feature built in, which you can see if I send this command. What happened is that Git noticed that the file I'm removing is not in the repository. So Git is essentially telling me, look, this file has changes that are not in the project's history. If I remove it from both the working area and the index, then it will be gone forever. Are you sure you want to do that? And it gives me a couple of options. I can force the removal, which is like saying, yeah, I know what I'm doing, just delete this file and forget about it. And I also have the option to remove the file from the index, but not the working area with ‑‑cached. That's what we wanted to do, so let's do it. There. The file is still in my working area, but I removed it from the index, that is I unstaged the file. So to recap, in Git, a remove without arguments is not the opposite of add. Add only changes your index while plain or remove changes both your index and your working area. You have to use the ‑‑cached option to make a remove work as the opposite of that. That's worth mentioning because it's a bit counterintuitive. Remove was pretty perplexing for me the first time I used it or the first few times actually. Now that I've told you about remove, let me delete the COPYRIGHT file for good. I thought about it, and I don't think that anybody will want to steal my cookbook anyway. There, I removed the file from the working area, and we're in the clean status again. Okay, that's it about removing files. Now let's take another baby step up to a slightly more complex topic.

### Renaming Files

We've seen what happens when you remove files. Now let's talk about moving and renaming files. Moving and renaming is actually the same thing, right? Renaming a file is just like moving it in place, if you wish, moving it to another name in the same directory. So I will show you what happens when you rename files here, and moving is exactly the same mechanism. Let's start from a clean status, as usual, with the three areas containing the same data. I'll only visualize one of our files, menu.txt. I want to rename this file and change its extension from txt to md. Md stands for markdown. It's a simple formatting language, and let's say that I'm planning to make this a markdown formatted file in the future. I will do the renaming in my working area first. There, so now the file in my working area has changed. I changed this color in the picture to show it changed. It's like a different file altogether. What happens if I ask for Git's status? The status is a bit confusing here. Git can see that there is a file in the working area that is not in the index, this file here. So it says this file is new. I don't know about it. It's untracked. And also, Git can see another file that is in the index, but not in the working area, this file here. So it says this file is in the index. It's not in the working area. You deleted it, right? Well no, not quite. How can I tell Git, look, this is actually the same file only with a different name? Well, the good news is I don't need to do that. I can just copy all the changes from the working area to the index. First, let me add the new file. There, it's been added, and now let's take care of the other file by adding that one as well. I will admit that this looks strange. I'm adding something from the working area to the index, but that something is not in the working area in the first place. Well, remember what add actually means. It means copy this data from the working area to the index. So if that data is nothing, as in this case, then Git will just override the data in the index with nothing, which means it will remove that piece of data from the index. So now we have the same data in the working area in the index. All of our changes are staged. And if we ask for the status, surprise! Look at this. Git already understood what's happening. It compared the content of the files in the working area and the index with the content of the files in the repository. And it noticed that the txt file and the md file have the same content, so they must be the same file with a different name. That's pretty smart, and it works both for renaming and for moving. It gets even better. Actually, Git is smarter than that. In most cases, it understands that you are renaming or moving a file, even if you change the content of the file at the same time. It just says these two files here look quite similar. It must be the same file. Now, of course, you can confuse Git if you try really hard, for example, if you move a file and you also change most of its content at the same time. But I would argue that if you get to that point that maybe you're doing too many things at once. Maybe consider moving and changing the file in two separate commits. For simple changes, together with moves, it just works. Oh, by the way, I didn't commit my changes yet. Let's do that. Now the file has been renamed in the repository, and we are in the clean status again. I just told you that Git tracks your moving and renaming automatically. So technically, you don't need a move operation, right? Still, there is such an operation, but it's just a convenience command. It basically does the same things that we did earlier. It moves the file in the working area and updates the index, only it does it in a single shot. See, I didn't need to rename the file first and that boots the old and the rename file to the index later. I did it all with a single git mv. Frankly, I personally don't even use git mv that much. I prefer to go through the steps manually and let Git figure out what happened. Just before leaving the module, let me commit what we have to revert the effect of the previous commit and leave everything in the clean status again. There.

### A Quick Summary

Let's summarize what we've seen in this module quickly. Here is how the basic commands impact the three main areas in Git. Add copious data from the working area to the index. It doesn't impact the repository. Commit copies data from the index to the repository, and it also creates additional objects in the repository, in particular, a new commit, and it moves the references in the repository. Switch and also checkout copy data from the repository to the working area in the index, and they move the HEAD reference in the repository. Remove delete files from both the working area and the index. It doesn't touch the repository at all. And mv, the move operation, moves a file in the working area and also updates the index. It doesn't touch the repository, and you can actually ignore it and do the same thing that it does with the other basic commands. There. We covered all the basics. Now let's move forward and deeper into advanced territory.

## Understanding git reset

### Understanding Reset

Welcome to the third module in Git Deep Dive. We're still going to talk about the four areas of Git, but now we're crossing that line that separates the occasional user of Git from the power user, so be ready. We'll talk about one of Git's most useful commands that also seems to be one of the hardest to understand, git reset. I have to admit that for the longest time I was nervous around a reset. I did use it for a couple of common use cases, but I never felt like I really understood what it was doing. I also knew that reset was a potentially destructive operation, which only made me more nervous around it. But like so many other things in Git, once you understand how this command works, you will wonder how it ever felt so hard in the first place. Why is reset such a confusing command, really? One reason is that before understanding reset, you have to understand a few other things about Git. You have to be familiar with the way branches work and the way the working area, the index, and the repository work. And if you are not familiar with those concepts, then you will have a hard time understanding reset. The good news is that you are familiar with that stuff now, so you're all set. The other reason that reset feels confusing is that it has many use cases, that is you can use it in different ways for different reasons. So if you look up for ways to do things with Git, you see a reset coming up again and again with slight variations, and the results look very different, depending on how exactly you apply it. So you end up thinking it must be a very complex command. Well, it's not. Let's see how it works. I will tell you what reset does first, and then we will see how those operations get kind of useful in practice. I will start our discussion on reset with a little quiz. How many Git commands do you know that move a branch? Think about it for a moment. You certainly know a few, four of them at the very least. Can you list them? Okay, your time is up. One operation that you know that moves the branch is commit. It creates a new commit, and then it moves the current branch to point at the new commit. Merge also creates a new commit in most cases, and it also moves the current branch to point at the new commit, so it's another command that moves the current branch. And rebase also does something similar. It creates new commits by coping existing commits in this case, and it moves the current branch to point at one of the new commits. And also git pull. It gets new commits from remote and updates the local and the remote branches. And maybe you know a few more. All of these commands move the current branch. They're all branch‑moving operations. However, none of them is a specialized command that only moves a branch. They move branches implicitly as a side effect of creating new commits or pulling them from remote. You might wonder, is there an operation that is more specialized, an operation that is all about moving a branch? Well, that's reset. The most important thing that reset does, the first step in a reset, so to say, is just that. It moves a branch. Generally the current branch, the branch that HEAD is pointed to. You pick a commit, say this commit here, and reset moves the current branch to that commit. So, that commit is now the current commit. Notice that a reset doesn't move HEAD. HEAD is still pointing to the same branch it was pointing at before, but the branch itself is moving, so HEAD is following along for the ride. If you only look at the repository, then that's all that reset does. It moves the branch to point at a specific commit. The part that you might find confusing, however, is not what a reset does to the repository. It's the second step. What a reset does to the other two main areas, the working area and the index. A reset does different things there, depending on its options. If you give the ‑‑hard option, then reset copies data from the new current commit to both the working area and the index. With the ‑‑mixed option, reset copies data from the new current commit to the index, but leaves the working area alone. This is the default option. So if you don't give any option to reset, then it will be a mixed reset. And finally, the ‑‑soft option means don't touch any of the areas, just move the branch and skip step two entirely. So this is what reset does. First, it moves the current branch, so it also changes the current commit. And second, optionally, it copies the files and directories from the new current commit to the working area and the index. That's all it does really. Okay, you might say, so how is that useful? How do you actually use a reset in practice? And that's where it gets interesting because there is no one single answer to that question. Remember when I told you that reset has many use cases. Indeed, depending on your use case, you might want to use reset with different options. Let's look at a couple of practical examples.

### A Reset Example

I will need to make a few changes to the repository before I show you how reset works. This time, I will draw the repository a bit bigger than the other two areas because we will focus on what's happening there first and foremost, so we need space in the picture. Okay, we are on the main branch in the clean status. So the three areas contain the same data. I will draw the menu.txt file and the recipes directory in each area. When it comes to the repository, however, let me draw a few more details as well. Here is the current commit. And I would also draw the current branch, main, which is pointing at the current commit. And finally, here is the HEAD reference, pointing at the current branch. There, now we have a pretty complete view of the repository. It's not really complete, right? As usual, I'm simplifying here. There are additional files in the recipes directory, which I didn't draw. I didn't draw the root project directory and so on. If I try to squeeze everything into this diagram, it will get really cluttered. But what we have here is the essential stuff that we need to follow this explanation on reset. Okay, so this is our starting situation. Now, imagine that I heard about this fashionable chef in Denmark who's having a lot of success with his new signature dish, squids with strawberry jam. I want to add this dish to the menu, so I will edit the menu. There, Strawberry Squids. Nice. So look at the diagram. I just changed the menu file in the working area. I changed its color to mean that the content of the file changed. Now let's stage it, and I just copied the updated file to the index, and now let's commit it. Look at what happens in the repository when I commit. Git creates a new commit. It moves the main branch to point at this new commit, and the new commit contains the new version of the menu file in the same old version of the recipes directory. Let's go through this entire process once more. In my cookbook project, whenever I add a new recipe, I'm also supposed to have a file in the recipes folder that contains the ingredients and other details about the recipe. So let me add a file with a couple of ingredients. Now this file is here in the working area and not drawing all the files, so I simplified and just showed the change to the recipes folder. I also changed its color, and I add it to the index, and I commit it. And again, Git creates a new commit, updates the main branch, and the new commit references the latest version of both menu.txt and the recipes directory. Now imagine that I get a phone call from a friend I trust who actually tried to cook squids with strawberry jam, and she guarantees that they are pretty terrible. She definitely does not recommend them. Great chefs sometimes go overboard with their experimentation. Well, long story short, I changed my mind. I don't want strawberry squids in my cookbook anymore. Unfortunately, by now, I have not one, but two commits that reference strawberry squids. I regret those commits. I would like that to go away now. How can I do that? Well, one way I can do that is by using reset. I have a commit here, two commits ago, this one. It still contains the older versions of my files before I started doing the squid thing. I would like this to be the latest commit in main, as if the last two commits never happened. That's where reset comes useful. I can take note of this commit's hash and ask Git to reset the current branch to it. This will move the branch back in time, so to speak. Of course, this should go without saying, moving a branch is an operation that changes the history of my project. So I should apply to this operation the same care that I apply to other history‑changing operations such as rebase. In particular, never change history that has been shared like history that I pushed somewhere on the center repository that is shared with my development team, for example. But in this case, these commits are only on my own machine, so I can change their history safely. What about the options to reset? Should this be a hard reset, a soft reset, or a mixed reset? Well, after the reset, I want to have the blue versions of the files, not just in the repository, but also in the index and the working area. I want to be aligned. I want to be clean again. So this must be a hard reset. Remember, ‑‑hard copies the files from the repository to the other two areas. Let's do this and see what happens. First, Git moves the current branch to the previous commit, which now becomes the new current commit, and HEAD follows along. Second, because this is a hard reset, Git copies the content of the new current commit, the blue versions of the files to both the working area and the index. And third, because the two squid‑related commits are unreachable now, they have no branch pointing at them, they will eventually be garbage collected. So we're back to where we were before this old strawberry squid fiasco, all with a quick reset.

### More Reset Examples

So we've just seen one way to use reset to move a branch, one specific use case. The use case was I want to revert the whole project to the state it was in a previous commit. We did that with a hard reset. But that is not by any means the only reason to use a reset. If I had to give you examples for all the reasons I can think of, then I'd have to talk about reset for an entire training. And I'm also certain that there are a few more reasons that I cannot think of, so I'll keep it short. I will just give you a few very quick examples. Here is one. Let's start from a clean status and focus on the menu file. Suppose that I've been experimenting with this file. For example, I edit it and add a new line for a BBQ recipe. I will stage this change. There we are. A change file in the working area, and it's also staged in the index. Now, what if I change my mind right now and I want to clean the stage file? Maybe I want to commit to something else first and get back to these changes later. In other words, I want to keep the changes in my working area, but I want to remove all changes from the index. In the index, I want the same version of the files that is in the repository. How do I do that? Early on, we've seen one way to do that, using the rm ‑‑cached command. That works, but it's not the only way to do it. Another way to do it is by using a reset, a specific kind of reset, what you can call a HEAD reset. This means that we are moving the current branch to the commit pointed at by HEAD, but the current branch is already pointing at that commit by definition. So in this case, the reset doesn't move the branch at all. If you wish, it moves it to the same place where it already is, so it doesn't move it. This is like skipping the first step of the reset altogether, the step where it moves the branch. What happens after that? Remember the second step of reset. Git moves data from the repository to the working area and the index. In this case, we didn't specify the kind of reset we want, so Git will go with the default, and the default is a mixed reset. Remember what a mixed reset does. It moves data from the current commit to the index, but not the working area. And the result is it unstages all the changes. Now, what if I decide to throw it all away? I don't want to commit this change at all anymore. After all, a barbecue is not really a recipe. I should have separate recipes for all the different stuff in a barbecue. So for now, I just want to wipe out my edits to the files in the working area and go back to the clean status. I can do that with a hard HEAD reset. A hard HEAD reset doesn't move the branch because it's still a HEAD reset, but it copies data from the repository to both the index and the working area. It overrides everything there. This is a popular command, but I use it with some care because it's a destructive command. In fact, it's one of the easiest ways to lose data in Git. You're saying explicitly, I don't care about all the stuff in my working area, please just override it with the content of HEAD. And if I send it, everything in the working area and the index gets overwritten. I could go on and on with examples of using reset, soft and hard resets, head resets, but I think I made my point. There are many different flavors of reset that you can use in many different circumstances. What you need to remember is the two steps. First, move a branch, possibly moving it in place, but in general, moving it to a specific commit. And second, copy data from the current commit to the index, the working area, both or neither, depending on which kind of reset you're doing. And since I don't want to rename this course Git Reset Deep Dive, I guess I'd better stop here, frankly. Enough about reset. Let's move on to the next module where we're going to talk about a few more advanced commands.

## Breaking into Advanced Workflow

### Stashing Data

We've got to the last module about the four areas and also the last module in this first half of the training, and it's a bit of a grab bag. We're going to take a look at a few tools that didn't find space in the previous modules. We'll look at topics like working with individual files or even only parts of a file and also the general philosophy of Git. But first, we're finally going to talk about the fourth area, the stash. At the beginning of this training, I told you that there are four areas where Git stores your data, but I only focused on three of them. I ignored the stash altogether. That's because the commits we've seen so far have some kind of effect on the working area, the index, and the repository, but none of them has any effect on the stash. Indeed, there is only one command that affects the stash, and it's you name it git stash. So if you want something to happen in the stash, you have to be very explicit about it. And this is actually the selling point of the stash. It's all yours, even more than the working areas. The stash doesn't change unless you explicitly ask for it to change. Let's see how you can use the stash. I will start from the clean status as usual. And for now, I will only draw the menu text file here. Now let's create a new recipe in the recipes directory, guacamole. This file is supposed to contain a recipe or at least a list of ingredients, but I'll think about that later. For now, I'm leaving it empty, just a placeholder, and stage it. In this project, whenever I add the recipe, I must also add it to the menu file. Let me do that. There. So now we have an updated menu in the working area and a new file in the working area and the index. Now imagine that while I'm working on this new recipe, I get interrupted for whatever reason, maybe I need to do some work on another branch. I want to focus on that other work, but I don't want my half‑baked guacamole recipe to get in the way. So this is a good time to use the stash. I can store all my changes in the stash, and it will stay there safely until I decide to return to the guacamole recipe. I store the current status with git stash save or just git stash. I usually use the abbreviated form, and I also use this option, include and tracked. It means also stash files that are still on track that have never been added, files that are entirely new in the working area. It doesn't make a difference in this case because we don't have untracked files. But by default, git stash just ignores some track files, and I personally don't like the default much, so I use this option without even thinking about it usually. And here is what happens. Git takes all the data from the working area and the index that is not in the current commit in the repository and copies all of the data to the stash, and then it also checks out the current commit. So now we are in line with the current commit. We are in the clean status again. Our changes are gone from both the working area and the index. They're not really gone, of course. They're still in the stash, and I can read the content of the stash with stash list. And there it is. All my half‑done work neatly packaged in a single stash element. Now we can see the point of the stash. The stash is like a clipboard for your project. It's the place where you store stuff that you need to set aside for some time. And it's a multiple clipboard. You can have as many elements as you want. Each element gets labeled with information about the latest commit to make it easier to identify, and it also gets a serial ID. Right now, we only have one element, so it's called the stash@{0}. The next one would be stash@{1}. And all these stash elements are strictly local. They're on your computer, they stay on your computer. They're not shared, the push, the pull or the like. It's your own clipboard‑like thing. Now that my stuff is stashed, I could switch to the other branch and do whatever in there, create new commits, anything. My half‑done work on the guacamole recipe stays in the stash. And after I'm done with that other work, I can retrieve the stuff I stashed. You use stash apply to move data from the stash to the working area in the index, and you can pass the name of a specific stash element here, but I won't do it. So the command applies to the master reset element by default, that is stash@{0}. And there we are. All our data in the working area and the index is back where it was when I stashed it. I can finish the job. I can stage the menu file, add a few ingredients to the guacamole recipe. Here, good vegetarian stuff. I'm getting hungry. And let's stage this update as well and commit the whole thing. There. Now that our data is safe in the repository, I don't need that data in the stash anymore. Let's clear the entire stash like this. And there you go. That was the basic stash‑related workflow. There is more to know about the stash if you want to dig deeper, but what you've just seen is the gist of it, a clipboard for your project.

### Working with Individual Files

Another feature that I'd like to tell you about is working with individual files as opposed to entire commits. I spend a lot of time in this training talking about commits, and indeed, we usually work with files only up to the point where we put those files in a commit. And from then on, we tend to work with commits all the time. When you reset, you reset to a commit. When you check out, you check out a commit and so on. But commits are pretty course in a way. You remember, a commit is like a snapshot of your entire project at some point in time. Sometimes you want to work with something smaller than your entire project, something like a single file or a single directory. Let's look at an example involving the menu file and the README file. I will edit the menu, and I will make everything in it all uppercase with a bit of Vim magic. And I will also make a similar change to the README file. And then, I stage all these changes. Now let's say that I'm not sure about these changes anymore. Let's say that I like the change to the README, but I'm not sure about the change to the menu just yet. So I want to unstage the menu file, but not the README file. In the previous module, we've see one way to unstage a file. You can do a HEAD reset. Last time we did it, we did it on the entire content of the index; however, it turns out that you can also do that on a single file, so let's do it. We didn't specify that this should be a hard head reset, so by default, it's a mixed head reset. And remember what a mixed head reset does. It doesn't move the current branch because it's a head reset, but it copies data from the current commit to the index. It doesn't change the working area. Normally, such a reset copies all the data from the current commit to the index. But in this case, we're saying that we only want to reset the menu file, so only the menu file gets copied. And the effect is that the menu file and only the menu file has been unstaged. The README is still staged. Brilliant! Exactly what we wanted, right? Now, let's go one step farther and remove the menu changes from the working area as well. So we want to discard the changes in the working area, and guess what? As usual in Git, there are multiple ways to do that. And now we're going to look at a few of those different ways, but just to be clear, I don't expect that you're going to remember these different ways of doing the same thing, absolutely don't. What I'd like to do here is to underscore this concept that in Git, there are often multiple ways to do the same thing. So don't start out and try to remember all this stuff. So we want to discard our changes to the working area, and we want to do that for this single file without touching other files. One way that you might think of doing that would be to use the same instruction we used earlier, a HEAD reset. Only this time, make it a hard HEAD reset, which makes perfect sense if you ask me. But apparently it does make sense to me, but Git disagrees on that. This command doesn't work. Git refuses to do a hard HEAD reset with the path. That does feel inconsistent in my humble opinion, but it's the way it is. Instead, there are other ways to discard changes in the working area that actually work. One is right here on the screen recommended by Git itself. It's a command called restore. So what you do is you say I want to restore this file. And when you do this, you discard the changes in your working directory. So that's probably the most common way to do it. However, the restore command is a relatively recent one. It's one of the commands that I mentioned in the very beginning that didn't exist until 2018 or so. It was introduced together with the switch command. Before then, in the ancient times of Git, people had another way to discard the changes to the working area, and it was based on the checkout command like this. Look at this weird checkout here. Normally, checkout moves the HEAD reference in the repository, usually to a branch, and then it copies all the files from the repository to the working area in the index. In this case, however, checkout is not going to move HEAD. Checkout is just going to copy stuff from the current commit in the repository to the working area index, and it's only going to do that for the one file we specified. So we just lost all the changes to that file. The menu is back to where it was in the latest commit. Exactly what we wanted. Note that checkout with a file or a path in general is not nearly as harmless as a regular checkout. It's actually one of the most destructive operations you can do in Git. We didn't even get a warning. We just swiftly and recoverably destroyed all of our changes to the menu file. So, handle this command with some care. So now the only file that I have left modified as staged is the README, and I can finally commit it. So we see a couple of things. One that I'm going to reconnect to in a few minutes is this idea that Git is super flexible. You can almost always do the same thing in different ways. And you might argue that this flexibility makes it a bit pointless to worry about learning all the different ways, all the more or less obscure options to all the commands, unless you really want to be the Git guru in your company. It's probably more productive to learn one way to do something and stick with it. And the other thing we learned in this clip that was more specifically the point of this discussion is that you can do operations in Git that are very granular down to a single file. But actually, you don't have to stop there. You can get even smaller than a single file. Let's talk about that.

### Committing Parts of a File

You've just seen how to do some operations that are smaller than a commit, operations on a single file. But Git goes farther than that. It can operate on something that's even smaller than a file. That concept feels a bit weird, I've got to tell you, because so far we acted as if the file and the directory are the fundamental units of the Git model, the atoms of Git. And in part, that's true. That's a good model to describe Git. It operates on files and directories. On the other hand, you can split that atom in Git if you wish. Let me show you on the terminal. I created a new branch called hunks to show you this picture. You'll see in a moment why I use this name. And on this branch, I will make a few changes to a project file, the menu file here. I will add the popular Italian dish right at the top, Spaghetti Bolognese. And I'll also make a small change to the Apple Pie and turn it into an Apple Piece because it's lighter on calories that way. And finally, I'll add another Indian recipe right at the end. Let's have Aloo Gobi. So I made a few changes. Let's save this file, and with git diff, we can see the changes I just made. Now here is a new scenario. Let's say that some of these changes I want to commit immediately and some I'd rather commit later in a separate commit, or maybe I won't even commit them. I'll decide later. But for now, I only want to commit some of the changes. But how can I do that because all the changes belong to the same file? There are a few different ways to do that in Git. The quickest is to use git add only with this option, ‑‑patch or just ‑p, and here is what happens. Git looked at the changes I made and divided them into sections called hunks. That's where I got the term from. And for each hunk, it gives me a number of options. They are somewhat hard to make sense of at first, but one them, the question mark, gives us documentation for the other options. So I press the question mark, and we get more details about the options available. You see? We can add the hunk to the index, skip the hunk without adding it to the index and move on to the next hunk, quit without adding anything else to the index and so on. However, in this specific case, I have a problem with my changes in that they're all very close together. This is a short file. So Git decided that all my changes are part of a single hunk. So if I want to decide which changes to commit on a line‑by‑line basis, I have to ask Git to split this hunk a bit more into smaller hunks. There is an option for that, s for split. So I press s, and now we have three hunks it says here. That's what we wanted. The first hunk is the one that adds Spaghetti Bolognese. Now, plot twist. If you ever spoke with an Italian about spaghetti bolognese, you might have heard that spaghetti bolognese is a fake dish. It doesn't actually exist in Italy. We just don't think that bolognese sauce and spaghetti fit well together. Look, if we ever meet in person, I can entertain you for half an hour on this topic, but you probably don't want me to do that now, so just know that I want to skip this hunk. I don't want to add it to the index. I must press n to skip this hunk, and skip it I will. And I'll probably just remove this change later on. Then there is a second hunk, the one about Apple Piece, which I would also like to skip. It's not that I don't like to have a piece of apple, but honestly, I'd rather have a full‑blown apple pie, calories or not. So, I'll press n again and skip to the third hunk that adds Aloo Gobi to the menu, a lovely dish. And this is the one change I'd like to add to the next commit. So I'll press y for yes, please do add this hunk to the index and the next commit. Let's recap what we just did. We use git add ‑‑patch to add some changes from the working area to the index, not on a file‑by‑file basis, but on a more granular hunk‑by‑hunk basis. We only added one change from one file, and we left the other changes on staged in the working area. And in fact, if I do git status now, I get this weird result that the menu file is both in the list of changes to be committed in the index and in the list of changes not staged for commit because part of it is staged and part is not. We can still see exactly what changes belong where by using git diff. Do you remember this? It means show me the changes between the working area and the index, that is give me the unstaged changes. And to see the staged changes, that is the changes from the index to the repository, I can do that with git diff ‑‑cached. And here is the change that will go into the next commit. This example involved a single file, but I can also use this hunking functionality over multiple changes across many files. And as it turns out, git add isn't the only command that has this patch option. Other commands have it like git restore, git stash, and even git reset. In all of those cases, you can say ‑‑patch, and it means let me do this operation, not on a file‑by‑file basis, but on a hunk‑by‑hunk basis. So you can decide which changes inside each file are involved in a restore or a stash or a reset. To be clear, that's not how most people usually work with Git. That's why I insisted that you should think of the Git model as a model based on files and directories because that is the easiest way to think of it. But just know that you can access this model granularity when it's useful. I don't know about you, but I think that this has been a lot of information in this module. So let's take a step back and talk for a minute about the philosophy of Git and its general approach.

### Git Is a Toolbox

Consider a toolbox, just a regular toolbox, screwdrivers, hammers, that kind of stuff. The tools are specialized in a way, right? Each tool has specific features, but at the same time, you can do many different jobs with each tool. You can use a hammer to plant a nail or to remove a nail or to straighten a piece of metal, for example. And you can also use different tools to do similar jobs if you are creative. If you need to remove a nail, you can do that with a hammer or you can use pliers, for example. So you have a bunch of tools and a lot of flexibility in how you use them. And that's the metaphor I have for you here. Git is a toolbox. It's not a single program that does stuff. It's not even a collection of utilities that neatly match your use cases one to one. It's a toolbox because you have generic tools like reset or checkout. And like in a toolbox, you can use the same tool for different jobs. We have seen a few examples of that. Think about git reset, for example. You can use it to remove the latest commits from your history, you can use it to unstage a file, you can use it to clean up your working directory and so on. Or git add, you can use that to tell Git about a new file, to put a modified file in the next commit or even to signal that you solved the conflict in a file during a merge, if you remember how you fix conflicts, right? And you can also use different tools for similar jobs. If you want to unstage a file, Git doesn't give you a command named, I don't know, unstage. There is no such command. To unstage a file, you need to understand how the index works. And once you do, you can unstage the file with reset, or you can use remove ‑‑cached. Remember that one. Or you can even use the restore command if you look up how it works. We mentioned all of these commands in this training, some in passing, some in detail. The point here, once again, is definitely not that you should learn all these options to all these commands. The point is to understand this toolbox idea. Git doesn't tell you here is this command, it's the command that you've got to use if you want to unstage a file. Git gives you the basic toolset, and then it's up to you to pick the right tool. And it also gives you a few precision tools like staging only some hunks, some parts of a file, for example. This is a very UNIX‑like approach, which is not surprising because Git was conceived by Linus Torvalds, who also made Linux. And this approach makes Git a bit more challenging than other versioning systems to get into. That's true, but it also makes it really, really powerful and flexible. On the other hand, this philosophy also means that it's probably not worth learning everything in Git. That's probably overkill for most users. Just learn the general approach, understand the model because that's essential, and look up the details when you need them. And with that little pep talk, we can wrap the first and longest part of this training. In these four modules, we explored the Git toolbox. And in particular, we focused on tools that move data across the four areas of Git. Some of those tools also have other effects, especially on the repository. Think of commands such as commit or reset. We described those effects as well, but our focus was on moving data across the areas. And now that you know about those data shuffling tools, we can shift to another set of tools, tools that focus almost exclusively on the repository and are particularly useful for larger repositories and big demanding projects.

## Exploring the Past

### Becoming a History Surgeon

Welcome back. We've reached the second part of this training. It's shorter than the first part, just three modules. And it's about techniques and approaches that are especially useful for large projects, but not only for large projects. They can come useful in any project of any size. For example, this module, Exploring the Past, is a brief discussion of some commands that you can use to dig into a project's history to make sense of the history. It comes without saying that if you have a large project with a long, complicated history, then these techniques might come very handy. But even on a regular‑sized project, you might still want to explore the history. So that's one example of what I mean when I say techniques that are especially useful for large projects, but not only for them. So let's talk about exploring history. I remember this training that I watched once about Vim, the text editor. And the teacher said using Vim feels like being a text surgeon. I think that's a nice metaphor, whether or not you like Vim in particular. Some technologies are like that. They are very precise and decorate. And when you use them, it feels a bit like doing surgery. I mean, I don't know how doing surgery feels actually, but that's how I imagine it. You do precisely what you want to do on the data you want, and you use tools that are a bit scary maybe. They cut easily, but they are also very powerful. Git also feels like that, especially when you're dealing with history, reading or writing it. Rewriting history is not part of this training. There is another training in the Git path that is all about that or you're writing Git history. But reading history, we can take a few minutes to talk about that. One note about this module. In the beginning, I told you that I wouldn't linger on technical details in this training, that I would focus on the model and the way of thinking of Git, not on the commands, the options and like. Well, just for this short module, I'm going to relax that approach a bit. Working with history is mostly a technical thing, so I'm going to talk a little bit more about the commands here. Let's dive in.

### Referencing Commits

When I say exploring your project history, I'm mostly talking about commits. So we will be referring to commits a lot in this module. And there are many ways to refer to a commit in Git. Let's take a few minutes to talk about that. Here is our cookbook repository again. I will switch to one of the existing branches, this branch called nogood. It's one of the branches from the previous training. It doesn't matter what it contains exactly. I'm just looking for any old bunch of commits here. I can use git log to look at the history. But the thing is, the default git log is not very useful when you're trying to make sense of a complex history because it squashes everything in a single list, so it's hard to make sense of branches and merges and understand what really happened. So I will use git log, but with a few options. The ‑‑graph option gives me nice graph‑like structure where I can see how the commits branch and merge. And the ‑‑decorate option shows the positional references like branches and head. By the way, recent versions of Git will have this ‑‑decorate option by default. But I still mention it in case you are using an old version of Git. And finally, I will format the log so that each commit takes only one line. There, beautiful! We can see the structure of the repository now, all the references, including HEAD, branches, the works. This command alone can almost replace a GUI tool for many of my needs. I'll copy this information to a diagram here so that it stays visible as I keep working in the command line. Now let's say that I want to see information about the current commit. I want to know which changes were introduced by this commit, date of the commit and so on. I can use the git show command to do that. Of course, I need to tell it which commit I want to look at, how can I refer to this commit? The obvious way is to use its hash like we did in the past or better, the first few characters of the hash like this. There it is, detailed information about this commit. However, using hashes is not always the easiest or the most practical way to refer to a commit. Another way to do that is to give Git the name of a reference that is pointing at the commit. The nogood branch, for example, is pointing at this commit. So I can use the branch to refer to the commit. Also, had this pointing at the branch that is pointing at the commit. This is the current commit at the moment. So I can also refer to it through HEAD. So far, so good and nothing particularly new, right? Now, what if we want to reference this other commit, the second to last one? There is no reference pointing at it, so it seems that our only option is to use its hash. However, there are other ways. I can start at HEAD and add the caret like this. The caret means the parent commit. So now I'm asking for the parent commit of HEAD. Bingo! And if I want to refer to this commit here, then I can use two carets. That means go to the parent of the parent of HEAD. And I can also say the exact same thing by using a tilde sign followed by a number. You can read this as go to HEAD and then go back to commits. This is useful, especially if you want to look at say the tens commit before HEAD and you don't want to type 10 carets. You can just say ~10. The syntax is fine if each commit has exactly one parent, but it breaks down as soon as you have commits with multiple parents like this merge commit here. What if I want to refer to this commit, for example? It's the second parent of the second commit before HEAD. In this case, I can address it with this syntax. Let's see, start from HEAD, then go back to commits, and then pick the second parent. Voila! There are other even more sophisticated ways to refactor commits, some of them surprisingly sophisticated, in fact. Just for a quick example, I just ask Git to show me where HEAD was one month ago. That's cool. So you have many ways to reference commits, including a few surprisingly powerful ones, but we learned enough about that for what we need here. Let's move on.

### Tracking Changes in History

Now that you know how to refer to a commit, let's finally talk about how you can explore those commits more in depth, look inside them. There are a few useful commands for that. One is git blame that shows you where the lines in a file are coming from. Let's see who changed the apple\_pie file and when. Here are all the lines in the file. For each line, you can see the latest commit where that line was changed. The caret here means in this case that this line is there in the file since the very first time the file was added to the project. And all the other lines were changed or added in subsequent commits. And here, I can see the dates and the orders of these changes. Another useful command is git diff. We already used git diff a few times in this training to compare the content of two areas like the repository and the index. But you can also use it to compare other things, for example, two commits. Let's see the differences between the current commit and two commits earlier. Okay, so this is the only file that changed between the two commits. One line was added, and another line was modified. And as usual, you can use branches to reference commits. So one common technique is using git diff to compare two branches. This shows all the changes between the two branches. Comparing branches is really useful, especially before merging stuff. We've seen git blame and git diff; however, the most important command by far when you're exploring history is a command that you already know. We just used it a few minutes ago, git log.

### Browsing the Log

Git log is the most useful command for exploring a project's history. We've already seen a few options for it, ‑‑graph, ‑‑decorate, and ‑‑oneline, but there are many more, a huge amount of options, in fact. Git log looks pretty harmless, but it's arguably the most complicated Git command. It's super powerful. I will give you a few examples. You certainly don't need to memorize these examples. See this as a quick demo of some things that git log can do. To begin with, you can get a detailed git for each commit in the log like this, so you can see exactly which changes were introduced in the commit. It's also very colorful. One more option, you can filter the commits. For example, I only want to see the commits that contain the string apples in their message, and there is only one of them. You can even ask for all the commits that added or removed the word apples from any file with ‑G uppercase. I will also use the ‑‑patch option here to show which lines exactly were impacted by these changes. There, there are one, two, three commits that either added or deleted this word, and I can also see the specific changes. And in all cases, they include the string, apples. As an alternative, there is also a git‑grep command that is useful for this kind of history‑wide text searches. Check it out, if you wish. Git log can also visualize a specific range of commits. The easiest way to do this is to say, for example, git log ‑n, which means only show me the latest n commits in the log. You can also use two dots to express a range like this. This means show me the commits starting from five commits before HEAD up to the parent of add. I used to find this range syntax mildly confusing because you specify the oldest commit first here. But in the default git log output, the commits are reversed. So you see the oldest commits at the bottom instead. I did get used to it eventually, kind of. This commit range feature is particularly useful if you need to compare two branches. This is a common scenario. It's not like a diff. It's a different kind of comparison. A diff can compare the files into branches. Here, we want to compare the histories of two branches. That's easier to understand with an example. For example, the current branch is the nogood branch. Let's say that I want to list the commits that are in the main branch, but not in the nogood branch. So you can read these as go from nogood to main and show me all the new commits that you see, and here they are. If we merge main and nogood right now, these are the commits that we would get. This branch comparison is really useful. I do it all the time. Okay, I guess you've got the idea by now. Git log has a huge amount of options to show your history in a number of different ways, to filter commits, to format them and so on. Again, I definitely do not expect that you will remember these features just because I spent a few seconds introducing them. But still, I don't want to spend more time going through all these options. By the end of it, you'd probably hate me. The point of this module is if you know that those features exist, then whenever you need them, they're just a quick Google search away. So let's recap. In this module, we've seen a few ways to refer to commits with special syntaxes such as the caret and the tilde. And we've seen a few examples of commands that you can use to explore history, git diff, git blame, and git log. Here in this list, I'm showing examples of some of the arguments we've been using. Oh, and let's not forget git show, and that was it. This module was quick and maybe not very exciting. Just a list of small features essentially. The next one is arguably more interesting. It's about the pretty impactful feature of Git called submodules.

## Using Submodules

### The Reason for Submodules

Hello again. Let's talk about a useful and confusing feature of Git, submodules. Submodules are not a great fit for a video training because they involve a lot of technical nitty‑gritties. So if I went into those details, this part of the training would become long and involved and honestly boring. Instead, I'll give you a higher level overview of submodules, why they exist, how they work certainly, and also how to look at them to avoid surprises. First, what are submodules for? Consider this common scenario. You have a project that depends on another project. For example, you are developing an application, and that application uses a library. So there is a hierarchy here, an outer project let's call it that has a dependency on an inner project. This example of an application in the library is just the obvious one. There are more. Hierarchies of projects with dependencies are the rule rather than the exception. In enterprise environments, especially it's common to have large projects that are split into subprojects that use each other, and they can be built and tested in isolation. So again, you have dependencies between projects, but the application library is probably the first example that comes to mind if you think about project dependencies. That's why we're using it here. So going on with this example, the library is an independent project. It stands on its own. So it might even have its own team, and that team wouldn't even need to know that the application exists necessarily. On the other hand, the people working on the application must be very aware of the library. If the library changes, it gets a bug fix or a new version, then the application might also have to change, otherwise it might break. And the application team might even want to contribute patches and bug fixes to the library. Let's say that you're part of the application team. How do you set up your project to manage your dependency on the library? The most basic solution that is almost never a good idea is to take the library source code and copy/paste it into the application project. So if the projects are JavaScript‑based, for example, you literally copied the JS files from the library to the application, and now you can use the library code from the application. But when the library is updated, there is no easy way to put the update in the application project. You basically have to copy the latest files again. A less naive way to organize the two projects would be to use the distributed version of the library via package manager. So if the library is distributed as a package, using any package manager system like pip for Python or npm for JavaScript, you have a file in the application that says I need that version of the library, and you use the package manager to install it. This is the most common way to manage project dependencies, and it makes it very easy to update the application to the latest version of the library. On the other hand, it's not as easy to contribute back to the library. If you're working on the application and you find a bug in the library and you fix the bug, there is no smooth way to send the patch back to the library project. Yet another way to solve the problem is have two entirely separate project directories on your disk, one for the application and one for the library and then reference the library directory from the application directory. Some programming ecosystems allow you to do that, to link a project in another directory. It's kind of like the package manager solution, only instead of referencing a packaged library on the internet, you reference the library source code somewhere on your disk, so you can work on either project or both. This looks like a great solution, but it does have a downside that it's kind of hard to keep the two projects in sync. Think about it, each specific version of the application works with a specific version of the library. So what happens, for example, when you check out an older commit in the application project or you switch to a different branch? That version of the application might or might not work with the current version of the library. Now you have to roll back the library to an older commit as well, and it might not be obvious which one exactly. So, this is a good solution until you start using your version control system to default less, and then things can get messy. So because none of this solution is perfect, Git gives you yet more ways to tackle dependencies between projects. One of those is nesting Git projects. The directory of one project is literally inside the directory of another project. So you put the library project right inside your application project. Now the library still works as a standalone project with its own history, and branches, and everything. But from the point of view of the application, the library is a subdirectory managed by Git. So each specific version of the application, each specific commit contains a specific commit of the library, and everything stays in sync. And it's also relatively easy for the application team to contribute to the library because they can look at the library as part of the application or as a separate project, whichever is more convenient for what they need to do. So they can change something in the library project and maybe push it to the library to share the repository if they have access to it or otherwise send a pull request. So this idea of nesting projects might work for you, but you can't just take the directories of two Git projects and put them inside each other. If you do, you'd have two .git directories inside the same directory tree, and Git doesn't like that. Instead, you have to use one of the tools that Git gives you for this purpose. The most commonly used such tool is submodules, nested Git projects that Git is aware of and can manage in sync. That sounds simple enough, but using submodules isn't always quite as simple. Let's see a demo.

### Using Submodules

You understand submodules better if you look at them from multiple perspectives, for example, the perspective of a developer working on the library and that of another developer working on the application that uses the library. So I prepared the demo like this. I created two projects on GitHub called app and lib, and I cloned them to two separate directories on my disk. Each project has only a readme file in it right now. This is the app project, and this is the lib project. Now, let's imagine that these are not two directories on my disk, but two different computers that belong to two developers. Apollo was working on the app, and Liberty was working on the lib. Did you get it? Apollo, Liberty. Never mind. Just pretend that there are these two developers on separate machines. Now, Liberty only cares about the library, but Apollo cares about both projects. So he decides to add the lib project to his own app project as a submodule. He does that with this command. It's basically a clone operation, but for a submodule, and this submodule behaves like a regular Git project. It includes the entire history of the lib project, in this case, a single commit. Okay, let's get back up to the root of the app project. Now, if we ask for the status of this outer project, we see that the directory of the inner project is already staged and ready to be committed. And the submodule command also generated a new hidden gitmodules file that tells Git that the lib directory is a submodule, and this file is also scheduled for the next commit. So, Apollo commits this change, the addition of the lib project as a submodule. In the meantime, Liberty has been working on a new feature for the lib project. I'll use this empty file to represent the new feature, and she commits the new feature to the library and pushes her changes. So now the library is updated in the center repo. Apollo notices that the library has been updated, and he wants the update. Maybe he wants to use the new library feature from the app, so you would imagine that all he has to do is all the changes. But if he tries to just pull from the app directory, nothing happens in the submodule. Instead, he has to go into the submodule first and pull from there, and now he has the new feature. And when Apollo steps back to the app project, he can see that the submodule has been updated, and he needs to stage it and commit it just like any regular subdirectory. And while he's there, he also updates the app to use the new feature from the library. I'll symbolize that with this file. Okay, now both the submodule and the new file are staged and he can commit. Now, let's do it again. Liberty adds another feature to the library, and she pushes the new feature. Now Apollo wants feature2. He could go like the last time. He could go into the submodule and pull again, but there is a quicker way. With this command, he can pull all the new stuff for all the submodules. It's a quality of life command, let's say. It makes it easier to just say go into each submodule, including submodules of submodules and pull new commits. We only have one submodule, but if you have dozens of libraries, then a command like this can be a lifesaver. There, and now the submodule is marked as modified. This time around, Apollo just wants to update the library. He's not looking to add features to the application. So he adds the submodule directory and commits it, and he also pushes it. Everything looks pretty smooth so far. However, if you look a bit deeper into submodules, you will find things that might surprise you. Here is one. If Apollo goes into the library submodule and he looks for the log and the current status, you can see that he's currently on the latest commit in the library, but not on a branch. He's in detached HEAD mode, even though he never asked to be. You may be expecting the submodule to be on the main branch at the moment because we've been pulling the main branch. And another thing that may be surprising is that if we step back into the outer project, the application, and we move back in time to a previous version of the application that used an earlier version of the library, you might expect that Git automatically moves the submodule to the version that matches this earlier commit. But no, this library is still in the latest version. Git knows that this is not the right committing lib for the current committing app. It even marks the lib submodule as modified from here, but it doesn't do anything about aligning the lib to the application. To move to the right commit in the submodule, we have to give a specific command, that is submodule update, and I can make it recursive for submodules of submodules if we had them. And now the lib was rolled back to the right commit. This kind of hands‑off behavior by Git also happens if we clone the project for the first time. Let's switch to the story of another developer who comes to the app project. If this person clones the project as it is, everything looks fine except that the submodule is completely empty. This new developer has to issue a sequence of commands to initialize the submodule first, that means activate the submodule in a way, and then update it. And only then he has the entire project, or otherwise this other developer could've used this option to the clone command, and then they would have had all the submodules initialized. But it's something that is easy to overlook, especially if you don't know up front that the project has some modules. So I guess that this demo left you with conflicting feelings about some modules. On one hand, they seem to be a very convenient feature. On the other, they can feel cumbersome. Let's take what we learn, their reason about it, and maybe note down a few reminders for ourselves when we deal with submodules.

### Submodules Made Easier

If I had to write down what we learned from the submodule demo, I'd say it's three things mainly. And you can see them as three things to remember when you deal with Git submodules. First, you can look at the submodule in different ways, as a nested project, as a subdirectory of your project and so on. But a good way to look at a submodule is as a pointer to a specific commit in another Git project. That's maybe the most useful mental representation of a submodule. So when we updated the lib submodule inside the app project, what we got was essentially that, a specific commit in the lib, not a project, but rather a specific commit. We also got the project history and so on. But think of the entire submodule as a pointer, and you won't be surprised when, for example, you find yourself in detached HEAD mode in a submodule. So for example, don't make assumptions on the fact that a submodule will be on a specific branch unless you go into the submodule and switch to that branch specifically. Speaking of that, the second rule I personally apply when dealing with submodules is don't assume that anything will happen automatically in a submodule. You check out an older commit in the outer project, don't assume that Git will check out the matching commit in the submodule. You clone the outer project, do not expect that Git will clone or even initialize the submodules. Submodules are static. They were designed to avoid anything magic happening behind the scenes. Git wants you to have explicit control, so submodules don't change unless you ask for them to change. You always have to be very explicit. Now this fact that you have to explicitly ask for things to happen inside submodules is probably a good design, but it can sound like bad news, like a burden. It makes submodules feel like a chore. What if you have many submodules or even multiple layers of submodules nested inside each other? Do you have to, for example, update each and every one of them every time you check out a null commit? Well, no because here comes the good news. Git has high‑level commands that allow you to be explicit in a very terse way. So if you want to clone a project and initialize and clone its submodules all in one shot, you have a command to do it. We mentioned it, git clone ‑‑recurse submodules. And if you want to update all the submodules with a single command, even submodules inside submodules, there is a command that does that. To be clear, we're not getting into all these commands here because they are very specific, and also, they can be different, depending on which version of Git you are using. So I really don't want to get into the weeds here. I promised you that I would stay at a pretty high level. The important takeaway is in general, you can find a Git command to take the pain out of all the important submodule operations, even if it's not always an obvious command. So first, find your submodule's workflow, decide what operations you need to perform, and then look up the Git commands that give you those operations with a minimum of fuss. Those commands might not be obvious commands. They might be different, depending on the version of Git you're using, but they probably exist. And these are three things to remember about submodules. While I'm here, I'll add the fourth point that is more general. Remember that submodules are just another tool in the Git toolbox and they are not mandatory. There are alternatives to submodules, simpler alternatives usually. We talked about some of those in the beginning of the module. You can use a package manager for your libraries. You can refer another project on the same disk from inside your code. And also, Git itself gives you other ways to do some of the things that submodules do. For example, Git has a feature called subtrees that is less powerful than submodules, but also simpler. While a submodule is a pointer to another project committing your repository, a subtree is essentially a copy of another project in your own repository. So long story short, you have plenty of options to manage project dependencies in a Git project. Submodules are probably the most powerful of those options, and that's why we cover them here, but you might get away with a simpler option. With that, say goodbye to submodules, in this training at least, and let's move on to another topic, LFS.

## Working with Git LFS

### The Problem with Large Binaries

This last module is about Git Large File Storage, or LFS reference. LFS is not a part of Git. It's a separate project, but it deserves a place in this training because it's a frequently asked topic. And the reason why people are so interested in LFS is that many Git projects have a problem, especially large projects, and LFS is a way to deal with that problem. So, let's talk about the problem first and then how LFS addresses it, then we'll discuss whether LFS is the right solution for you and your project. The problem in a nutshell is that Git is good at dealing with text files like source code, but not quite as good at dealing with binary files, especially if they are big. A concrete example, let's say that you work for this imaginary company called Globomantics. They make a system that parses images with artificial intelligence. It extracts information from images like what the image represents, what colors it uses and so on. That's the project you work on, and the GitHub repository for the project contains many, many files, a lot of C code and Python code, but also loads of image files in PNG format for things like functional testing or to train the AI. Those image files tend to be bigger than source code files, megabytes rather than kilobytes, and they don't compress as well, so they take up a lot of space. And the system is always evolving, so the images are also changing all the time. Now, that's what Git is for, right? Storing every version of every file, but there is a problem with that. Git never forgets. If you commit a file and then update it multiple times, and even if you end up deleting it, previous versions of the file are going to be in the repository forever. So all those versions of all those big image files, they add up. So the project's repository grows and grows. Now a very large repository wouldn't even be an issue if it stayed on the server, right? Just ask the company to buy a bigger disk. The second is, Git is distributed. So every person on the project clones the entire repository. Now that can spell trouble because a 10 GB repository is not 10 GB on some server somewhere. It's 10 GB on each developer's laptop. Also, consider what happens when a new developer joins the project and they clone the repository for the first time. Cloning a multi‑gigabyte repository can take hours. And even after cloning, the pain isn't over yet. Imagine that you do a Git pull and one of the files you pull is a 50 MB file that has been updated four times since the last time you pulled. In that case, you're going to get all four versions of that file, that's 200 MB, even if you only care about the most recent version. So a simple pull can take forever, and it breaks your flow, and it takes up space on your disk, all of that because Git wants you to have all the older versions of all those big files. And question, do you need those older versions? Maybe you do. Maybe you frequently go back in history and check out those old commits that include those old files, but maybe not, right? Maybe you rarely check out past commits at all and you only care about the latest version of each file. So why pay up front in terms of time and in terms of disk space for those irrelevant old versions? That is the problem that LFS is set to solve. Let's see how.

### Using LFS

Here's a very short demo on what it feels like to work on an LFS project. I created a tiny sample project and configured it to use LFS. Configuration was just a couple of commands. And then I put this project on Bitbucket. That is a Git hosting service like GitHub or GitLab. Some of these services give you LFS support right out of the box, and Bitbucket is one of them. So I'll clone the project to my local machine. This is the URL, and here it is, just a README and the fruit‑png file, an image. Let's have a look. There. It's a nice apple. But Git being Git, I didn't just get these two files. I also got the hidden .git directory. Let's put information about this directory on the screen. In particular, check out the size of .git. It's 2 MB, and .git contains the entire repository of the project. That's why I can check the log. And if I wish, I can check out any of these old commits, all right? Nothing new here. But here is the twist. You see, things don't really add up here. The fruit‑png file alone is nearly 2 MB. I read in the history it seems that this file has been updated multiple times already with different images. So how come the entire repo is pretty much as large as the current working directory if it contains all the history as well and that history includes multiple large images? If you are thinking compression, that cannot be the answer here because PNG is already a compressed format, so compressing PNG usually doesn't make it any smaller. The answer is LFS. It's been working its magic behind the curtains. You see, when I created this project, I told LFS to track my PNG files. You can see it with this command. There. By the way, you only have this Git LFS command if you have LFS installed. I installed it on this machine with my package manager. So LFS is tracking PNG files, and that means that those PNG files in older commits didn't actually get cloned on my computer. The commits are here on my computer, but the files themselves are not. With the exception of the files I need in my working area like the current version of fruit‑png, the older versions are not in my local GitHub repository. They on an LFS server somewhere. In this case, it's Bitbucket's own LFS server. But the selling point of LFS is that I don't really notice that those files are missing because as soon as I check out one of these older commits and at that point, I need the files in that commit, LFS will go out and download those older versions for me. I'll check out this older commit where apparently fruit‑png used to be a different image. And look what happens to the size of the repository. Boom! It nearly doubled the size because LFS quietly downloaded that older version of fruit‑png. And in fact, now there is an orange in there rather than an apple. And if we check out this version, the repository keeps growing, and we have a mango here in the image file. In a minute, we're going to see exactly how LFS is playing this trick. But the general idea is this, all these files that are tracked by LFS are downloaded to my computer on demand. And once they've been downloaded, they stay here. So now if I switch back to main, the size of the .git directory doesn't change because I already had the apple version of this image. So LFS didn't have to go and download it again. Essentially, what LFS is doing boils down to downloading some files lazily, only when I need them, if I need them. If I never check out these old commits, then I never pay the cost of downloading them. Of course, you need a network connection when you check out these old commits, otherwise you'll get an error. But if you're okay with that, then LFS can help you pay as you go instead of paying up front for downloading these files. And if you want to track some other file with LFS, you can say, for example, LFS, I want to track MPG files as well, and you got them. In the future, if some other developer pushes an MPG file to the project, that file will be downloaded in a lazy fashion. And it's just as easy to track, for example, a specific file or all the files in a certain directory. That's the minimum that you need to know to work on an LFS‑enabled project. But maybe you're looking for a deeper understanding of LFS, so let's open the hood and peek inside.

### Inside LFS

Just like Git, LFS is easier to use if you know how it works internally. So let's dive a bit deeper into it. First question, how does LFS know that it's supposed to track some files in this project and which files exactly? The answer is in this hidden file, .gitattributes. It's a standard in Git for storing information about paths, groups of files. Right now it says that these two extensions are tracked by LFS. When I asked LFS to track PNG files, .gitattributes did not exist yet, so LFS created it, and I added it to the project and committed it just like any other project file. And then we asked LFS to track a MPG file a minute ago, and it added that information here. So LFS will look in here to know what to track. But next question, how does LFS even get involved? For example, when you commit a file, how does it happen that LFS jumps onto the scene and maybe changes the way that the file is managed? That happens through a system called Git hooks. Maybe you know about hooks, but if you don't, then think of them as events in Git. They're short programs stored here in the .git/hooks directory. Git invokes these programs before or after certain operations based on the naming convention. For example, right after a commit, Git calls post‑commit if it exists. And when you start using LFS, for example, when you tell it to track PNG files, LFS goes into this folder and writes a few of these hooks so that they on LFS itself at the right time. So from now on, when you do, for example, a commit, Git calls post‑commit that calls LFS that does its thing. And now the most important question, what does LFS do exactly? How does this lazy loading thing work? To answer that, I'll dig a bit deeper right into Git's object database. Brace yourself. I'm going to look into the Git objects themselves and navigate from this commit to the fruit‑png file in the repository or rather to the blob that represents this file in the repository. If you watched the How Git Works training, then this object‑by‑object navigation will be business as usual. Otherwise, you might be puzzled for a minute, but no worries. Just ignore this low‑level stuff if you don't understand it. It's optional material. What matters is that you understand what LFS is doing, not exactly how it's doing it. So step one, I will print the content of this commit on the screen. You might remember this low‑level plumbing command to print a Git repository object on the screen. Like all commits, it points to a tree that is the root directory of the project. Let's look at the tree, and the tree is pointing to a blob that represents our image file. So, let's print this blob. Now if this were a regular non‑LFS project, you'd expect to see a bunch of garbage‑looking binary data on the screen because this is a binary file, all right? You expect the bytes of the image on the screen. But no, we get a small snippet of text data. What is happening here? What's happening is that behind the scenes, LFS quietly replaced this blob with a pointer of sorts. It switched the image file for a reference to that file. The file is stored somewhere else. Where exactly depends on how you configure your project. In this case, I'm using Bitbucket. That gives me its own LFS storage in the cloud, so that's where this image is stored. But LFS doesn't want to go to the internet every time you check out a commit. So the first time we fetched the file, it stored it in its own local cache. We talked a lot about the four areas in this training, right? Now think of the LFS cache as a fifth area that only exists if you use LFS. The cache is here, still in the .git directory, but outside the object directory that is outside the Git repository. A few minutes ago, we did download this file, right? So we should see it here in the cache. Let's hunt it down. LFS organizes its file in a way that is similar to Git. So again, if you know how Git works internally, you won't be very surprised. And if you don't, no worries. This is not something that you need to know. The gist of it is that LFS uses a naming convention based on the file's hash to store the file. Notice that this directory has the same name as the first two digits in the hash of this pointer. And the file in here has the same name as the hash, so it must be our fruit‑png file. One thing that LFS does differently from Git is it doesn't compress files. Because you never know with binary files, they might not compress well. So we can just rename this file with a png extension and open it. And here is our apple. To recap, LFS replaces files in Git with a pointer, a snippet of data that tells LFS where to locate the files on the remote. And it leaves the files themselves on the remote storage for as long as it can. When LFS eventually has to download a file like this PNG file in the lower‑right corner here, it downloads it to the local cache so that it doesn't have to download it again in the future. That's the right way to think of LFS, a cache that adds lazy downloading to Git. So now comes the big question, do you need this lazy downloading in your project?

### Do You Need LFS?

So should you use LFS or not? Well, it depends on your specific circumstances. So it's not like I can give you a straight yes or no answer, but I can give you guidelines. And I would say that there are two situations that suggest that you should consider LFS. First, if you have files in your repository that are unwieldy like big binaries and you or your teammates usually don't need those files in your working areas, you barely ever check out those files, that's the selling point of LFS, a cache between your working area and your repository. It's easy to misunderstand this point. Earlier today, I read a question from someone who said we have these big files in our project and we never update them, never delete them. They're just sitting there, we just add big files over time, and we're thinking of using LFS. Well, if you never update or delete those big files, then they're always going to be part of the current commit, right? They're right there in your working area all the time, so why bother replacing them with pointers in the first place? Basically, you have a cache, but you always need everything that could ever be in the cache anyway, so LFS doesn't buy you anything in that situation. On the other hand, if you know that those files are rarely ever showing up in your working area, then LFS can help you by replacing the files with pointers and downloading them lazily. A second reason to use LFS is you have your project on an online service like GitHub, GitLab, Bitbucket, one of those, and your project exceeds that service's limits. Online services put the limitations on the size of your repository, on the size of each file in the repository and so on. If you exceed those limitations, then you can upload the large files to some external LFS storage service. And using LFS with these services can be easy because they usually give you their own LFS storage with more space than regular Git storage. Note that in this case, you are not using LFS because it's a local cache on your computer. You're using it because it makes the size of your repo in the cloud smaller by putting big files outside the repo. So those are the two broad reasons to use LFS, as a local cache or as a way to keep your project within limits on services like GitHub. The irony is that LFS was conceived for the first case, but I suspect that most people these days use it for the second reason, to make GitHub happy basically. But if you're thinking of using LFS for the first reason as a downloaded cache, then I'd say think carefully whether you really need it. I mean, LFS does what it says on the box, and it's pretty seamless. Once you set it up, you barely notice it's there, but it's still another moving piece in your project, another thing that you have to worry about. And also, transitioning to LFS isn't always super easy. For example, if you have a large repository and you adopt LFS, then it might help you control the growth of that repository in the future, but it won't do any good for past commits unless you rewrite or history that some projects might not want to do, so your mileage may vary a lot. LFS could be a blessing or it could be insufficient or it could be overkill for you. So before jumping into LFS, also consider its main alternative, that is plain old Git. You see, Git has a few relatively obscure features that might just solve your problem. One example, if you have a large history and you don't want to clone it on each developer's machine, then you might want to look into a Git feature called a shallow clone. It's a simple concept. You clone the project, even a large one. This one has tens of thousands of commits. And you have this ‑‑depth option that says that you only want to get the last 1,000 or 100 or 12 commits. And now I only got the last 12 commits. So even if the full history is huge, I don't care. I only got the top slice of it. Of course, now I cannot get back in time or explore commits before this 12 because I don't have them in my repo, but I can change my mind later and make my clone deeper or even more shallow if need be. This is one example of a Git feature that might help you deal with large repos, but not the only one. Git has been getting better over the years at dealing with large files, especially since Microsoft has started using Git to develop Windows. That is a humongous project with millions of files and thousands of pushes per day, so they optimize Git a lot just for that. And they ended up contributing many of those optimizations back to Git itself. So even without LFS, regular Git is pretty darn efficient today, and it has some smart features to manage big binaries. For example, you can pull the project's history, but not the data in that history. The data itself is loaded lazily. That is called a partial clone, and it's kind of similar to what LFS does. And Git also has a feature called prefetching that fetches new data from the remote periodically in the background. So when you do a pull, you don't have to wait to download that data. It's already downloaded. So if you want LFS because you take a long time to pull data from the remote, this feature might save your day. And there are more features and optimizations, many of which are internal and sometimes invisible, but they may get faster at dealing with large files and large repositories. In general, Git is much more efficient today at managing big projects than it was when LFS was conceived. To be clear, that doesn't mean that those features and optimizations replace the need for LFS in all cases, but they do replace it in some cases, and it's worth checking where your project stands. So again, don't just use LFS straight away. First, make sure that your problem cannot be solved with plain vanilla Git. And with that, we're done talking about LFS, and in fact, we're done with the topics in this training. Let's recap it.