## Git Is Not What You Think

### Version Check

A version check before we start. This training was produced on this version of Git, but if you're using a different version, an older version, or even a later version, you'll most likely be okay. I'm going to be more specific about Git versions in literally a few minutes. For now, I just want to reassure you up front that you are not watching obsolete information. Again, more about Git versions in a couple of minutes. Okay? Let's begin.

### Introduction

Welcome! In this training, we're going to talk about how Git works internally under the hood. That might sound like a weird choice. Why is that important? Well, of course, there is some geeky pleasure in understanding how things work, but that's not the most important reason to know this stuff. Give me 1 minute to tell you why we're talking about the internals of Git. When you think about Git, you probably think about the high‑level user commands, the so‑called "porcelain" commands. You're probably familiar with the basic ones, like add and commit. And if you work with a remote repository, then you probably also use the push and pull. And if you work with branches, you use the branch, switch probably, merge, maybe even rebase. The list goes on. If you have a lot of Git experience, then maybe you got even deeper than these into the low‑level commands, the so‑called "plumbing" commands. For example, cat‑file, hash‑object, and a few more. These are the basic building bricks that the porcelain commands are built upon. Maybe you never even heard of these commands, and that's okay. You might never need to use the plumbing commands unless you're doing some advanced Git scripting or the like. Now, understanding all these commands can be hard. Some of them can be confusing. However, here is a key point. You could argue that the secret to Git is not about knowing the commands, either porcelain or plumbing; instead, the secret to Git is about knowing the conceptual model behind Git. If you want to use Git safely and unleash all of its power and not get in trouble, then don't look at the commands. Look at the model instead. Once you do, the complexity of the Git commands kind of fades away. Suddenly, Git looks simple, even elegant, I promise. You don't get stuck anymore. So if you really want to become a Git master, then you should understand the model, and then you will also understand the commands much more deeply after you understand the model. And this is what I want to talk about in this training, the model. So let's get started. But first, let's take a minute to check whether this stuff works on the version of Git you're using.

### More About Git's Versions

Git has been around for a while, so you might wonder whether this training converts to the version of Git that you're using, and it probably does. Almost everything we'll say here works the same on even much older versions of Git, with a couple of minor exceptions, maybe. The most noticeable exception by far is in this training, we'll use the git switch command to move to another branch, and that command is relatively recent. A few years ago, you used git checkout instead of switch to move to another branch. So if you see that switch doesn't work on your version of Git, use checkout, and you're golden. For the purpose of this training, they work the same way. Anyway, I'll remind you of this when we get there, so you don't need to take note about this. And if a new version of Git ever changes anything that we say in this training, then I'll update the training as soon as I can. That doesn't happen often because we'll cover fundamental features of Git that change regularly. Last time it took over 5 years before I had to tweak the information in this training. But if you do notice anything obsolete, please let me know. Post a comment on the discussion board, and I'll be on it.

### Git Is an Onion

So, we want to wrap our head around Git. Let's start by talking about what Git really is. It's not necessarily what you think. Imagine that Git is layered, like an onion. We won't try to understand the whole onion at once. That would be very ambitious to eat the whole onion. Instead, we will peel off the layers of the onion until we reach Git's conceptual core. One common definition of Git is a distributed revision control system. That's a mouthful. Not only Git does what other revision control systems do, it does that in a distributed way that's harder to understand than traditional client server revision control systems, like subversion. So let's make things easier for ourselves and peel off one layer. Let's remove distribution. In this first part of this training, imagine that Git is not distributed at all. If you can, imagine that there is only one computer in the world, and there is a repository on that computer. That's all you want to think about for the moment. So, Git becomes just a revision control system, no distribution. However, a revision control system is still a complex beast. It includes things such as history, branches, merges, and these features make things more complicated. So again, let's simplify things. Let's remove one more layer. What happens if you forget about the branches, history, and the like? Now we have a smaller onion. You can call it a stupid content tracker because that's all it does; it tracks content, files, or directories. And if you look at Git's documentation, you will see that this is actually Git's definition of itself, Git, the stupid content tracker. If you look at it as a content tracker, then Git is easier to understand. But let's take this one step farther. Forget even about tracking files. Forget about the notion of a commit. Forget about versioning. Let's look at the very core of the onion, the basic idea behind Git. And let us say that at its core, Git is just a map, a simple structure that maps keys to values. And this structure is persistent. It's stored on your disk. Now we reach the core of Git. During this training, we will rebuild the onion from the inside out and understand each layer in depth. In this module of the training, we'll talk about the first two layers of the onion. This information will be pretty technical. You might even wonder why I will go into so many details. But this is the groundwork we're laying here. Be assured that by the time we get to the second module and we get to the outer layers of the onion, you will be surprised by how concretely useful this deep understanding turns out to be. Even things you've been doing every day with Git might look different and simpler.

### Values and Hashes

I just said that at its core Git is a map. That means that it's a table with keys and values. What are the keys and what are the values? Well, the values are just sequences of bytes. For example, the content of a text file or even a binary file. Any sequence of bytes can be a value. You can give a value to Git and it will calculate a key for it. A hash. Git calculates hashes with an algorithm called SHA1. This is one of those technical names that nobody seems to agree how to pronounce. You might hear people call it SHA1 or "Sha 1." Other people call it "SHAN" that I think is nice because it kind of rolls off the tongue. So I'll call it "SHAN." Every piece of content has its own SHA1 hash. For example, let's take a piece of content, the string "Apple Pie." If you ask Git to generate a hash out of this string, then you will get this hash, exactly this one. There is only one hash for this string. SHA1 hashes are 20 bytes in hexadecimal format, so they are a sequence of 40 hex digits. This will be Git's key to store this content in the map. We can also calculate the hash on the command line. To do this, we need a command that you might never have heard about because it's a low‑level plumbing command, git hash‑object. So let's pass our piece of content to hash‑object. I wish I could do it like this. It would be simple, but I can't. Hash object is not very user‑friendly. It's a plumbing command. So if you do what I just did, Git will think that Apple Pie is the name of a file. Instead, I can use the echo command to output this content and then pipe the result into hash‑object, like this. I also need to tell hash‑object to get its content from standard input, not very intuitive. By the way, if you're using Windows, then you will use slightly different shell commands. But don't worry too much about that. I'm not asking you to send this command yourself necessarily. It's enough that you understand what it does. It points out the hash for this piece of content. And here is the result. This is the hash for the string Apple Pie. This is the same hash I showed you on the previous slide. We have the same content, so we get the same hash. If you change anything in the content, a single letter, for example, I will add a new line character at the end, like this. Then you get a completely different hash. Speaking of Windows, one thing you might find confusing is if you do send the equivalent command under Windows, then you'll get a different hash. But that isn't because there is any difference in git hash‑object or the hashing algorithm. It's just that the echo command behaves differently under Windows in a few ways. In particular, it doesn't strip away these quotes. It considers the quotes a part of the string. And it also adds a new line at the end of the string. So, long story short, you're actually hashing a different sequence of characters under Windows. But that doesn't matter in general. It only matters for this specific example because we're using echo to generate the sequence. In general, if you pass the same sequence of bytes to hash‑object, then you get the exact same hash every time on every operating system, and each object in Git repository has a SHA1 hash like this. If you put the string Apple Pie in a file and store this file in Git, then the hash we just generated will identify the file. And as we'll see later, directories also have their own hash, as do commits, and so on. With so many hashes around, you might wonder what happens if they collide. After all, the number of possible hashes is it's large, sure, but it's not infinite. What if I have two different pieces of content and just by chance they happen to have the same hash? Wouldn't that make a mess of my project and cause me to lose my data? Well, yes, it would. But it's unlikely to happen. Let's see just how unlikely it is just because it's fun. Think of the U.S. Powerball lottery. How many chances do you have of winning the lottery jackpot? Google tells me that the chances that a particular combination of numbers wins the jackpot are about 1 in 175 million. This is a large number, so let's try to visualize it. Imagine printing one ticket for every possible combination of numbers in the lottery. You get 175 million tickets. Now put all these tickets in a line, 1 every 25 centimeters, about 10 inches. That's a very long line of tickets, long enough to span the equator. Now, imagine starting somewhere on the equator and taking a walk around the world. It's a long walk, and also quite a bit of swimming, so it's going to take a while. You are walking along this very long line of lottery tickets, remember, and just once during your trip, you are allowed to pick up a single ticket. And if you're really lucky, that's the one ticket that wins the jackpot. Congratulations! That's how hard it is to win the jackpot. Now, imagine that you enjoyed winning the jackpot so you want to try it again. You take a second trip around the world. You once again pick up a single ticket along your way, and you win again. Now, that's really, really good luck. Winning the jackpot twice in a row is almost miraculous, as a matter of fact. Now, imagine doing it a third time, and amazingly you win again, and again, and again, six times in a row. Now, winning the jackpot six times in a row is extremely unlikely, you will agree. Well, going back to Git, these are about the same chances of getting the same hash for two different pieces of content. It's just not likely to ever happen to you or to anybody by chance. So for all practical purposes, SHA1 hashes are unique. Not just unique in your project; you can think of them as if they were unique in the universe. You could put all of the data you will ever write in your life in the same Git repository, and Git would assign a different hash to each version of each file in each folder. That's a lot of data. You might get some performance problems. Okay, you probably will. But still no collisions. Later in this training when we talk about distribution, this notion that hashes are unique in the universe will come useful. For now, I'm only mentioning it to say if you have ever worried that two hashes might collide in your Git project, then no need to worry about that.

### Storing Things

So, we have seen that Git is a map where the keys are hashes and the values are pieces of content. But they also said that Git is not just a map, it's a persistent map. Where does persistence come from? Let's go back to the git‑hash object command we used a few minutes ago. If I want the Apple Pie content to be persistent, I can add the ‑w argument to this command; ‑w stands for write. So now besides generating the hash, Git will also save this piece of content in its repository. However, now we don't have a repository yet, so if I try this command straightaway, Git complains. We're not in a Git project. We don't have a repository. I don't know where to save the content. So, let's turn this director into a Git project. There is a command for that, and you probably used it already. It's a high‑level porcelain command, git init. There. So, we created a new repository, and Git is saying, look, the new empty repository is in this directory .git. That's a hidden directory. On this operating system, I can visualize hidden directories with the ‑a switch. So, now Git has the .git directory, a place to save stuff, and if you are on the hash object command again, with ‑w, we get the hash, and we also save the content. Let's see where exactly. Let's peek inside the .git directory. There are a few files and folders here, but for now, just look at this directory here, objects. This is called the object database. It's the place where Git saves all its objects, like the string Apple Pie we just saved. Let's peek inside. Ignore these two, the info and pack subdirectories for now. We'll come to this later. Instead, look at this subdirectory here. Its name is 23, and these are the first 2 hexadecimal digits of the hash of the content we just saved. And if we look inside 23, there is a file in here, and the name of this file is the remaining digits of the hash. Git uses this scheme to organize content and spread it over multiple directories. It's just a trick to avoid piling up all the content into a single huge clutter directory. Our original string Apple Pie is inside this file. This is what Git calls a blob of data. A blob is a generic piece of content. However, the original string has been mangled a bit inside the file. Git added a small header and compressed the content to save space. So we cannot just open the file and read it. But we can use another low‑level plumbing command to look at the content. It's called git cat‑file. Once again, don't worry if you don't remember this command, it's rarely used. I'm using it now just because I want to show you how Git saves content. Git cat‑file takes the hash of an object and an argument. If we run it with the ‑t argument, it stands for type, Git tells us what this piece of content is. It's a blob. And if we run it again with the ‑p for printing, then Git unzips the object, removes the header, and it prints out the actual content of the blob. And here it is, the string Apple Pie there. So, so far we have seen that Git can take any piece of content, generates the key for it, a SHA1 hash, and then persists the content into the repository as a blob, a persistent map. This is the very basic of the Git model. Let's build on this and move on to the next layer of the onion.

### First Commit!

We have seen that Git is a persistent map, but you probably don't see it as a map. You see it as something more than that, something that tracks your files and your directories, a content tracker. Let's see what that means. We need an example project, so I built a very simple one, a cookbook. In the root of the project, there is a file named menu.txt. This is supposed to be a menu, a list of all the recipes in the cookbook. Right now it only contains a single recipe, Apple Pie. Then we have a recipes directory that contains a README that tells you that you are supposed to add one separate file for each recipe here. And indeed, we have one file here with the recipe of the apple pie. This file is supposed to contain the entire recipe. For now, it's just a placeholder, actually, and it contains the string Apple Pie. I'm using this string a lot here. I like apple pie. We'll fill in the real recipe later. So we have three files, one in the root and two in the recipes folder. It's a very simple project, but that's what we want for now. We want to understand how Git stores these files and folders, so it's better if we start simple. Let's make this a Git project with git init. There, now, we have a .git directory here. And because it's a brand‑new project, the object database in the objects folder here is empty, apart from the info and pack subdirectories. We can ignore those as usual. Now that we have a project, let's create our first commit for this project. Let's use the git status command to see the files and folders in the project root. You've probably used the git status before. I configured my Git installation to use core, so we can see that both the menu text and the recipes directory are red because they are on track. That is, Git doesn't yet know what to do with them. You know that to commit a file, I have to put it in the so‑called staging area first. It's like a launch pad; whatever is in the staging area, we'll get into the next commit. We can add these files to the staging area with the git add command. Let's add the menu text, and then the recipes folder, and all of its content. Now the files are green. It means that they have been staged. Let's commit them. I will use the ‑m argument to Git commit so that I can give a commit message right here. There, now the staging area is clean, and we can use another popular command, git log, to look at the list of existing commits. There is only one, and this hash starts with these digits. Okay, good to know. So far, this was business as usual for any Git user. Now let's go deeper. Let's open the hood and look inside the Git object database. This is going to be short, but intense, so hold on. If you look in the .git directory under objects, you will see that we have a bunch of subdirectories in here now. One of these is named with the first two digits of the commit. And here are the remaining digits, so this file must be the commit. A commit is compressed, just like a blob, but by now we know how to pick inside compressed files, and we can use git cat‑file for that. I will git cat‑file the commits shown with the ‑p so that it prints the content of the commit. And here it is. So, what's a commit? It's a simple and very short piece of text. Nothing else. It's truly as simple as this. Git generates this text and then it stores it pretty much the same way it stores a blob. It adds a small letter to this text to say this is a commit, it generates its hash, it compresses the text, and it stores the result in a file in the object database. The commit text contains all the metadata about the commit, the name of the author, the committer, both are myself, the date of the commit, and the message. And then it contains something more, the hash of a tree. What's a tree? Well, just like a blob is the content of a file stored in Git, a tree is a directory stored in Git. The commit is pointing at the root directory of the project. That's what this tree is, the root of the project. If you look in the object database, you will see a directory named with the first two digits of the tree's hash, and inside it is the tree, a file named with the remaining digits of the hash, as usual. It's just like a commit, see, a piece of content that is generated by Git and then hashed and stored in the object database. So what's inside this tree? What does it look like? Let's cat‑file it. Just like a commit, a tree is a tiny piece of text. That's all it is. And it contains a list of the content of the directory, a list of hashes, actually. In this case, we have a blob and another tree with their names. The blob is the menu text file that's in the root, and the tree is the recipes directory that's also in the root. There is also some additional data for the files and directories to access permissions, but otherwise, that is it. That's all it takes for Git to store a directory. Now, if you have great memory for hexadecimal numbers, I don't really, then you might find the hash of this blob familiar. It's the same hash as the Apple Pie string that we've seen earlier. Let me prove it for you. I will use cat‑file ‑p as usual, pass it the hash of the blob, and there it is, the string Apple Pie. That's what's inside menu text. So to recap, the commit points to a tree, the root, and this tree points to a blob, menu.txt, and another tree, recipes. And the blob is just a piece of content, the string Apple Pie. Now let's finish the job. Let's look at these other three and see what's in there. Let's use cat‑file again to peek inside the recipes tree, and there you are, two blobs. One of these blobs is the README file. I will cat‑file it. Here, there it is, the content of the README. The other blob, well, this one looks familiar even to me now, even if I can't remember numbers because it's the same hash as the menu text blob. That's because these two files have exactly the same content, so Git will not create two separate objects for them. It will just reuse the existing object that is already in the database. So to be picky, a blob is not really a file. A blob is the content of a file. The file name and the file permissions are not stored in the blob, they are stored in the tree that points to the blob. You will see in a few minutes why this is a good thing. For now, let's update our diagram of the object database. The recipes tree is pointing at the blob with the content of the README file, and it's also pointing at the blob with the content of apple\_pie.txt, which is the same content as the menu.txt file, so it's actually the same blob. And there you are, the whole object database. All of it. One small note about this. If you try building this exact same project and you use the exact same comments that I used here, then you will see that you get the same hashes for all the trees and all the blobs. However, the hash of the commit, that one will be different because you have different data in your commit, a different user, and a different commit date. The important thing to understand here is that there is no magic behind the SHA1 hashes. If you have the same content I do, then you get the same hash. A commit is also just a piece of content, and your commit has different content than mine, so you get a different hash. It's as simple as that.

### Versioning Made Easy

Now brace yourself. We're going to talk about versioning. We're going to see how it works. You might think that versioning is a big deal and complicated, but now that you know about the object model, you will see it's not that complicated at all. First, let's change a file. I will edit the menu.txt file. I will add the name of another recipe to it, Cheesecake. I'm in a cakes mood. Let's save the file, and now git status tells us that the file has changed. So let's stage it with git add and create a new commit. There, now our working area is aligned again, and if we look at the log, we can see both commits. Let's use the now familiar cat‑file to peek inside this second commit. There. This commit has something more than the first one. It has a parent. The parent is the previous commit, our first commit. Commits are linked, and most commits have a parent. The very first commit is an exception. So, the commits are linked like these. Also, if you look at the hash of the tree that this second commit is pointing at, you will see that this is a brand‑new tree. It's not the same tree that the first commit was pointing at. It's like a different root. You'll see why in a minute. But for now, let's just draw a new tree here. Let's look at the content of this tree. You know how to do it by now, so I will go fast. We'll cat‑file it. We can also use just the first few digits of the hash, and Git will automatically iterate the whole hash from the database, unless there are multiple hashes starting with these first few digits. So now we can see that the tree contains another tree, the recipes folder, and the blob menu.txt. Note that this is not the same menu.txt blob we had in the previous commit. It's a brand‑new blob because this file has changed. So if we cat‑file it, we can see it has the new content of the file, all of it, including both the apple pie and the cheesecake. However, the tree here that lists the content of the recipes directory, this one is the same object that we already had in the database since the first commit because the contents of this directory haven't changed. So there is no reason to create a new object. Git can just use the object that was already in the database. So here is the file structure of the object database after our second commit. The new commit is pointing to a new tree, which is pointing to a new blob, and into the same tree as the first commit. Now it's clear why this tree must be new. This blob has changed, so the content of this tree must be different because it's pointing to a different blob. As usual, if you change anything in a piece of content, then you get a whole new object with a new hash. This tree, however, it hasn't changed because nothing inside the directory changed, so Git can reuse the same object. That's one of the reasons why Git is so efficient; it doesn't store things more than once. We changed the single file, so Git stored the new blob, and in our case, a new tree and a new commit, because they are ultimately pointing at that new file, so they had to change. The hashes they contain have changed. Trees and commits are really small, so even if they change, it's not a big deal, the process is still quite efficient. If you count the number of objects in this diagram, it's two commits plus six trees and blobs, eight objects total. This is the current number of objects in the object database. Let's double‑check it. The database itself is getting a bit crowded. So instead of counting the files, let's use one of those seldom‑used plumbing commands, git count‑objects. And there you are, eight objects, and they take a very small amount of disk space. Speaking of efficiency, you might be surprised that Git stores a new blob every time you change a file. What if I have a huge file and I only change a single line? Will Git store an entire new blob in this case and duplicate the rest of the file? Well, not really. Git also has another layer of optimizations to save more space. For example, as you keep working at adding content to the repository, Git might decide to store only the differences between the two files, or even compress multiple objects in the same physical file. By the way, that kind of stuff is the reason for those mysterious info and pack directories in the database. However, we are really talking about implementation details, and usually you can ignore them. To understand the Git model, it's good enough to think of each commit blob or tree as just files, separate files, that are hashed and stored in the database. At a commence level, this is how Git actually works. And then it has another layer of optimizations that are probably not interesting to you unless you're working on the Git source code. Just know this, when it comes to being efficient, you can assume that Git always does the right thing.

### One More Thing: Annotated Tags

So, let's recap. In the Git object database, you have blobs, arbitrary content, trees, the equivalent of directories, and commits. There is also a fourth kind of object in the database. It's something called annotated tags. Maybe you know what a tag is. It's like a label that you put on a commit. For example, you might tag a commit to mark a release. That's a typical reason to use a tag. And there is also a special kind of tag called an annotated tag that carries additional data, like a message, and the name of the person who created the tag, and the date of the tag. Now, we won't go into details about tags for now; we'll do it later in this training because by then it will be easier to understand how tags work. For now, just note that an annotated tag like this one is a small object in the database that contains the data, the messages, and so on, and it points to a commit. So it's another type of database object, very similar to a commit in a way. Again, the details will come later, but I wanted to at least mention annotated text for completeness because there is nothing else in the database, just these four types of objects. So if you know about blobs, trees, commits, and annotated tags, then congratulations! Now, you know the entire Git object model.

### What Git Really Is

Take one last look at this object model because there is something interesting to say about it. Look at the whole model from an abstract point of view. What do we have here? Well, we have a structure where some things contain data, blobs, and then there are other things called trees that contain blobs and other trees. So the entire structure is recursive. And the names of the blobs and trees, they are not in the objects themselves. Instead, they are stored in the containing tree. So you can have the same object, say, the same blob, or the same tree, pointed at by different trees with different names. Does this structure remind you of anything? Well, to me, it looks an awful lot like a file system. Just like in a file system, you have content, files, or blobs, and nested containers, directories, or trees, in this case. And you can have a links. The same file or directory can be reached from different places with different names. It's like links in Linux or shortcuts in Windows. In fact, you might argue that that's what Git is. It's a high‑level file system built on top of your native file system. This shouldn't surprise us. After all, Git was written by Linus Torvalds, who wrote Linux. He is an operating system kind of person, so when he built a version control system, he built it like a file system. That's just the way he thinks. It's a version file system, of course, because it also has commits, which have a version. And that's what we mean when we say that Git is a content tracker. So, we have seen that Git is a persistent map at its core, and layered on top of that is a stupid content tracker that looks a lot like a version file system. In the next module, we will put this theory to work to understand the all‑important next layer of the onion. You will see how easy Git branches are and how easy branch‑related operations are once you know the basic model of Git. See you in the next module!

## Branches Demystified

### Introduction

Welcome to How Git Works, module 2. This is where things really get interesting. So far, we laid the groundwork. Now we're going to make that information concretely useful. In the previous module, we said that Git is a stupid content tracker. We had this metaphor of an onion. Now we can move on to the next layer of the Git onion and look at the features that turn Git into a full‑fledged revision control system, features like branches and merges. I'm assuming that you already have a basic idea of what a branch is. Maybe you even use Git branches every day. But after this module, you might end up looking at them in a different light.

### What Branches Really Are

Let's go back to our cookbook project. For now, it's still just a handful of files and a couple of commits. We haven't created any branches yet. But you probably know that as soon as you have a Git project, you also have a branch. Git creates this branch for us when we do our first commit. Let's look at the list of branches in the project with a git branch like this without any argument. And there it is, our default branch that in this project is called main. By now, we're used to looking inside the .git directory, so let's do it again, looking for branches this time. What's a branch? This main branch must have some kind of concrete representation in this folder. What does it look like? Well, Git normally puts branches here in a directory called refs and a subdirectory called heads. Ignore the other subdirectory for now. And there it is, a small 41‑bytes file called main. This is our main branch. What's inside this file? You could probably guess it. But you don't have to guess, it's not compressed, so I can just print its content to the screen. And there you are. The file contains a single line, a hash. And as you probably expect, it's the hash of the current commit, this commit here. To recap, we have two linked commits in this project, and we also have a main branch, and the branch is a reference, a pointer to a commit essentially. That's why the directory that contains branches is called refs, references. And note that this main branch actually has no special status in Git. Yeah, Git created it for us, but otherwise, it's just a branch like any other. And in general, we conventionally decided that it's the main branch of our project, unless we decide otherwise. And just like any other branch, all there is to it is this small file. I could actually delete or rename the main branch just by deleting or renaming this file. If you're willing to live dangerously, you could even create a new branch just by writing a new file into this folder containing the hash of a commit. I wouldn't recommend that kind of hack on a real project, but on a test project, try it if you wish, it does work. But in this training, let's stick to the proper way to do it. Let's create a new branch with git branch and branchname. About this branch, we mentioned that we want to insert recipes in our cookbook, but we also get ideas for new recipes, and we want to keep them separate from the official recipes until we try them. We want them in a separate branch. Let's call it ideas. So, our plan is that we can put all our established recipes in main and the new ideas in the ideas branch. There we are, we have a new branch. We can see it listed among the branches, and we can see it alongside the main in the refs heads folder. And if we look at it, we see that it has exactly the same content as main, same commit. So, this is what we have now, two commits, two branches, and the branches are pointing at the same commit.

### The Mechanics of the Current Branch

Now that we have two branches, if we look at the list of branches again, we see that one branch is marked with an asterisk because it's the current branch. What does that mean, concretely? I mean, how does Git know that main is our current branch? There must be some kind of information, probably in the .git folder that says which branch is the current branch? Some kind of file, maybe that contains that information. And indeed, there is such a file, and you probably know the name of this file already. If you look at the .git folder again, you will see a file named HEAD in here. If you have experience with Git, this should ring a bell. And if you look inside HEAD, then you will see that it contains a reference to another file. This is Git's way to reference files, this syntax. It's saying HEAD is currently pointing at refs heads main, the file representing the main branch. There is only one head, so there is only one current branch. That's what HEAD is, a reference to a branch, a pointer to a pointer if you wish. Let's add it to the diagram and move on. So now let's change the files in the project. I will add the list of ingredients for the apple pie. Here let me edit this recipe. I will just copy‑paste the ingredients here. We don't have a full recipe yet, but at least we know what to buy at the grocery store now. Here, let me add this file to Git and commit it. Okay, let's see what just happened inside Git step‑by‑step. Git created a few new objects in the object database for this commit. In particular, it created the commit itself. And this commit has the previous commit as a parent. Then Git looked inside the HEAD file to find what the current branch is, and it moved that branch to point at the new commit. So the main branch moved, but notice that HEAD itself did not move. It was pointing at main before the commit. It's still pointing at main. Main is moving, HEAD is just coming along for the ride. So far, we didn't touch the new ideas branch. Ideas is still pointing at the previous commit where it was when we created it. Now, let's make ideas the current branch. We need to switch branch. That's an operation called git switch. We say git switch and the name of the branch. One note about this, I already mentioned it earlier, but this is a good time to repeat it. There are two common ways to move to another branch. One is git switch and the other is git checkout. These two commands have different options and they are useful in different cases. However, when you use them to just move to another branch without any special option, like we are doing here, they work the same. You can say git switch ideas or git checkout ideas. The same thing happens. The only important difference is that a switch is a relatively recent command. So if you're using an old version of Git, then maybe you don't have switch, and in that case, use checkout; it works the same. I generally use switch here to change branch because that's the whole reason why switch was introduced, to change branches, while checkout we'll see later that it can also do other things. So, switch is more specifically for changing branches compared to checkout, but they both work. So back to where we were, git switch ideas. When I do this, two things happen. The first thing that happens is that Git changes HEAD to point at ideas. There. Now head is pointing at refs/heads/ideas. The second thing that happens is more subtle. Git just replaced the files and folders in our working area, the working directory, with the files and folders in this commit. So after the switch, our working area changed to the content of the commit pointed at by ideas. If I look at the content of the apple\_pie file here, the ingredients are gone. This is the previous version of the file. I'm sure this doesn't surprise you. I mean, this is what you expect when you do a switch, right? By the way, we will look at this process in more detail in a few minutes. So that's what switch means. It means move HEAD and update the working area. Now, let's modify the apple pie recipe again. I will pass in a variation on the list of ingredients. Almost the same list as we have in the main branch, but the version in the ideas branch also has cinnamon in the pie and more apples, 10 apples instead of 8. Let's add these changes and commit them. There. Tweak the apple pie recipe. Okay, by now we know what happens when we do a commit. Git created a new commit in the object database and it moved the current branch, ideas, to point at the new commit. HEAD didn't change. Main didn't change, of course. But ideas changed. Now it's pointing at the new commit. Now this looks a bit more like our intuitive notion of branches. But remember that branches are still just references to commits. That's all there is to branches, actually. Now let's see what happens when we merge these branches.

### Let's Merge!

Let's switch back to the main branch. There. Now, the branches didn't move, remember, but HEAD did move. It's now pointing at main. And if we look into the apple pie recipe, we find our own version of the recipe here, not the alternate version. Now, let's say that I tried the two apple pie recipes. I actually baked both apple pies, and I like the alternate version a bit more, the one in the ideas branch. So, I want to merge changes from the ideas branch into the main branch. Let's do the magic, git merge. And there you are, we have a CONFLICT. We want to have both our changes and the changes from ideas in main, but Git is warning us that at least some of those changes are conflicting. We need to solve the conflicts manually. And chances are, you probably got in this situation already either while using Git or some other version control system. If we look inside the apple\_pie file, we will see that this line, this one, was changed in divergent ways in our recipe and in the alternate recipe in the ideas branch. So now we need to take a stand and decide how many apples to use in the pie. I can't decide which version is better, so let's go for one of those compromises that usually work fine in the kitchen and add just one more apple. Middle ground. Okay, there. Now, if we get the status, we see that this file is not staged for the next commit. We need to add it explicitly. This is our way to tell Git that the conflict has been fixed. There. And now we can complete the merge. If we hadn't had conflicts, then Git would have done this last step automatically, but because we did have conflicts so we have to say, okay, we're done fixing all the conflicts, and we do that with a commit. We don't even need to give it a commit message. Git knows that we are in the middle of a merge, so it will create a suitable message automatically. We could change the message, but I won't. I will just approve it by saving and quitting the editor. If you look at the log now, you will see a brand‑new commit. And if you look inside this commit with cat‑file, remember cat‑file, the low‑level command that we used to pick inside the objects in the database, there it is, it's just like any other commit that we've seen so far. A merge is just a commit with one exception‑ it has two parents. That's what makes it a merge. A commit in Git usually has one parent, but it can have as many parents as you like, actually. So let's update the diagram. Git created a new commit with two parents, to represent the merge, and moved the main to point at the new commit. That's how merging works.

### Time Travel for Developers

Okay, now give me just a few minutes for a short aside. From the first module of this training, you might remember diagrams such as this one. It's about tourism blobs. To make things easier, I avoided talking about tourism blobs in the second module. I mostly talk about commits. I'll just mention tourism blobs again quickly to show you in more detail how Git manages your working directory. You know that the objects in the database are commits, trees, and blobs, and also annotated tags, although there are none of them in this example. You also know that all these objects are arranged in a graph, they reference each other. There are references from a commit to its parents, references from a commit to its tree, and references from trees to blobs and other trees. These references all look alike, but they're used in two different ways. References between commits are used to track history. All the other references are used to track content. We've also seen that Git is good at reusing content, so you can have objects that are reachable from more than one commit, like these ones here. The point I want to make is that when you move to another commit, for example, you switch to another branch, Git doesn't care about history. It doesn't look at the ways that commits connect to each other. It just cares about trees and blobs. So if you look at the world from this commit here, then Git forgets about the link to the parent of the commit, and it looks at the tree in the commit and all the objects that can be reached from there. That is the entire state of the project at the time of the commit, a complete snapshot of every file, every folder. Git uses this information to replace the content of your working directory. That's how you travel back and forth in time with Git; it is the whole point of versioning. And if you look at this commit here, same thing. It comes with an entire representation of the entire project. You might think that merge commits must be more complicated than that, but actually they're not. Okay, they have multiple parents, that's the definition of a merge, but Git doesn't care about that when you move to a commit. It just goes into the commit and retrieves the tree in the commit, as usual. A merge commit will in general have its own trees and blobs because the objects in the merge might not be present in any of the parents. Think of a file that has lines from both parents, for example. On the other hand, from the merge commits tree, you can probably reach objects that are also reachable from other commits. And once again, Git doesn't care about which blob of tree was introduced by which commit. When it stores the commit, it just reuses objects that are already there, and it creates the objects that are not already there. A way it checks out the commit, it just looks at the tree and rebuilds the state of the project from there. I told you this story because I want to make a couple of points. First point, don't get confused with trees and blobs. Retrieving a past state in Git is a pretty straightforward thing. It's just a stupid content tracker. It goes into a commit and retrieves the trees and blobs from there. Nothing more complicated than that. When you use Git, focus on the history, how commits connect to each other, and trust Git to do the right thing with trees and blobs. The second point I want to make is that Git doesn't really care much about your working area. Remember, when you move to another commit, Git just replaces the working area with the stuff from the object database, and Git mostly cares about the objects in the database, not your working directory. The objects in the database are immutable and persistent, while the files in your working directory are as transient as they get. They can change as quickly as you can do a git switch or checkout. Git is not reckless with your working area. It will give you a warning before overriding your files. For example, if you try to switch to another branch, but you have uncommitted changes, Git will tell you that. But other than that, as far as Git is concerned, your working area is the least important part of your project. All the good stuff is in the .git directory. And now that I made this aside, you can forget about trees and blobs for the rest of this training. From now on, we will mostly be concerned with commits and history.

### Merging Without Merging

We have seen how branches and merges work, but there are a couple of interesting corner cases that we didn't consider. They are quite important in practice, so let's look at them. The first corner case is a special case of a merge. Let's switch to the ideas branch. There. HEAD moves to point at ideas. Now we are on this branch again. Now let's say that we made up our mind on this whole apple pie debate and we decided that the recipe that's currently main is the best. We baked it, and having one less apple really worked great. So now we want to make this apple pie from main the official apple pie, and we want the same recipe in ideas so that it can be the basis for future experiments. In other words, earlier on, we merged ideas in main. Now we want to merge main in ideas. Now how does Git handle this merge? It could do it in the usual way, like it did when we merged in the other direction. It could create a new commit that has two parents. These two commits here would be the parents, and then move ideas to point at the new commit. This new commit would be guaranteed not to have conflicts because we already solved the conflicts when we merged in the other direction, so it would be easy for Git to create this commit, but it would also be wasteful. Think about what we are trying to achieve here. We want a commit that contains the latest version of all the stuff in main, and the latest version of all the stuff in ideas, that's what we want. But we already have such a commit. It's the latest commit of main. It contains all the latest objects in main, of course, and also the latest objects in ideas because the latest commit in ideas is an ancestor commit of main, and all the conflicts have already been sold in main. We learned by now that it is frugal. It doesn't like waste, so it can spare a commit and just do this instead. It moves ideas to point to the same commit as main, so Git didn't have to create a new commit. This trick happens all the time in practice. It's called the fast‑forward. Whenever you see this message on the screen, this is Git bragging about being able to spare a few objects in the object database and making your project's history less complicated. Good, Git.

### Losing Your HEAD

The second and last corner case I want to tell you about is a feature that turns out to be quite useful in practice. I will simplify the diagram for this. I will switch to main and forget about the ideas branch for a while. Actually, let's forget about everything for the moment except for the very latest commit. So far, I always told you that HEAD is a reference to a branch, which in turn is a reference to a commit. When you switch to a branch or you check out a branch, you can use either switch or checkout, that means that you're changing HEAD. However, you can also do something different. You can directly check out a commit instead of a branch. In this case, you have to use checkout, you cannot do a switch. Switch is only for branches. So I will checkout this commit. I will just use the commit's hash. There. Now, if you look inside HEAD, it's not pointing to a branch, it's pointing directly to a commit. And indeed, there is no current branch at all. We're not on a branch. This is a situation that is called detached HEAD. How is this useful in practice? Well, let me work a little bit more so I can show it to you. Let's make some experiments in the apple pie recipe, something that I'm not sure I want to keep around. There. It's good with 9 apples, so it must be even better with 20, right? And I will commit this. What happens when I commit? Well, in this case, Git cannot move the current branch, as usual. There is no current branch. So it will track the latest commit by moving HEAD directly. HEAD is working like a branch here. Okay, let me hack in a few more changes. Let's make the pie sugar‑free. It's healthy. Another commit, another HEAD movement. Okay, now let's say that we had enough of this. I tried baking an apple pie with all these extra apples and no sugar. It tastes like cooked apples. I don't like that. So we'll abandon the experiment. I will switch to main again. Okay, now head is back where it belongs on the main branch. So are our files. Everything is business as usual. We are all back the latest to commit. But there is a nagging question here. What happened to these commits? Well, they're still in the object database somewhere together with all their trees and blobs, but unless I took note of their hashes, these commits and their connected objects are now unreachable. They can't be reached by starting from a branch or a tag and walking the objects in the database. They are effectively isolated. I can only reach them directly by their hashes, and I'll probably forget those hashes soon. If you have any experience with object‑oriented languages, then you know what happens to an object when it can't be reached by any reference. It gets garbage collected. At some point, the system decides that the object is wasting precious memory, and it deletes the object and recovers the memory. And this is exactly what happens in Git as well. Every now and then in the course of other operations, Git decides that it's worth running a garbage collection. The garbage collector will look for objects in the database that cannot be ultimately reached from a branch or HEAD or tag, and it will remove them to save disk space. Remember, each object is just a file in the object database, so removing them is as easy as deleting those files. So, these commits I created will likely stay in the database for some time and then disappear. If I want to save them, I must act now. How do I do that? One thing that I can do is move back to the last commits. I can still do it because I have their hashes here and the garbage collector didn't run yet, so these objects are still in the database. There. That was a last‑minute save. And now that I have the commit, I can put a branch on it. Here, let's create a branch called nogood. Now I can switch to main again, and this time around, the commits are safe. There is a branch now that accessed the entry point to this section of the object graph, so these objects will never be garbage collected, and I can easily get back to them by checking out nogood if I wish. This is a common way to use a detached HEAD. When you want to try out something, go down a detour, and in general experiment with your code, you don't have to leave behind the convenience of using Git. You can detach HEAD, do your experiment, stay committed to the experiment as much as you wish so that you won't lose data, and then you decide whether to keep the experiment or do away with it. Just remember to put a branch on the stuff that you care about before you leave it behind.

### Objects and References

Now we have a better picture of the nature of Git. Let's recap. A Git repository is a bunch of objects linked to each other in a graph. They can be commits, blobs, trees, or annotated text that we'll talk about a bit more later. Then there are branches that are references to a commit. And finally, there is HEAD. That's also a reference, but there is only one of it, and it marks our current position in the graph. It's usually pointing to a branch, but it could also be detached and pointing directly to a commit. Then there are a few rules. First, the current branch tracks new commits. So if you create a new commit by saying git commit or git merge, for example, then the current branch moves to the new commit. If you are in detach the HEAD state, then HEAD itself moves to the new commit. Second rule, your working directory is updated automatically. When you move to another commit, typically with git switch or git checkout, Git replaces the content of your working directory with the content that can be reached from that commit. Three, any commit, blob, or tree that cannot be reached from either branch, HEAD, or a tag is considered dead and can be garbage collected. And essentially this is the whole thing. Branches, merges, moving back and forth in time, it all boils down to these simple rules. Okay, I know what you're thinking now. What about rebases? Yeah, rebases, we'll talk about that in the next module.

## Rebasing Made Simple

### Introduction

Welcome back! Git is an onion, remember, and we're still looking at the version layer of the onion, the features that turn Git into a full‑fledged revision control system. In the previous module, we talked about branching and merging. Now, let's look at a couple more features that are also related to revision control, and in particular, one important feature, rebasing. Branching and merging are standard features for any revision control system, but rebasing is way less common. Only a handful of version control systems have it, and Git is by far the most popular of them. In a way, rebasing can be seen as Git's signature feature. Let's see how it works.

### What a Rebase Looks Like

Here is our cookbook project. Again, I worked on it for a bit since the last module, and this is the situation that we have now. There are two branches. The main branch got a couple of new commits since the last time we looked at it, and these commits changed the apple pie recipe a bit, just minor changes. The other branch, spaghetti, is brand new. It also has a couple of new commits, and these commits introduce a new recipe for spaghetti alla carbonara. I just had to sneak an Italian dish into this training, at least the one thing that I can pronounce properly. So here is that situation again. We have two branches that diverged. To make the diagram simpler, I also used the different colors for the commits in the two branches. The apple pie commits are pink and the spaghetti commits are light blue. Also, because the spaghetti is the current branch, I do it in green instead of throwing a separate HEAD pointer. Now we want to put the content of the two branches together. We already know one way to do this. We can merge the two branches. We're already on the spaghetti branch, so we could merge it with main. I will not do this, however, but if I did, here is what would happen. We would have a new commit, and the parents of this new commit would be the former tips of the two branches. Also, the current branch would move to this new merged commit. This is the usual merge thing that we already know about. In this case, it should also be an easy merge because we're not expecting any conflicts. However, I will not complete this merge. Instead, I will use another way to put two branches together. I will rebase the current branch over the other branch. If we rebase the spaghetti on main, then here is what happens. Git looks for the first commit, spaghetti, that is also a commit in main. It's this commit here. This is the base of the spaghetti branch. All the history before this commit is already shared between the two branches, so it's not relevant here. Now, Git detaches the entire spaghetti branch from this commit and moves it on top of main, so it changes the base of this branch. That's why it's called a rebase. Like in the merge, we might have to solve conflicts to complete the rebase, but in this case, there are no conflicts. The changes in the two branches impact different files, so we're done already. Now the spaghetti branch contains all the commits from the main branch, plus the spaghetti stuff, which is what we wanted. What happens if we want it to work the other way as well and we want the stuff from spaghetti in the main branch? Just like in a merge, we can just switch to main and rebase the other way. Let's switch to main. Here, main is now the current branch. It changed to green in the diagram. And now let's rebase. Actually, in this particular case, I could either rebase or merge and it would make no difference whatsoever. In both cases, Git can just fast‑forward the branch. A rebase can be fast‑forwarded just like a merge. So, this is what we have now. Like in a merge, we have all the commits that deal with the spaghetti and all the commits that deal with the pie in the same history. However, different than a merge, we got that result not by having multiple branches flow together, but by rearranging the branches so that they look like one single branch. The way I just described it, a rebase is pretty simple. However, to be honest, I'm making it a bit too simple maybe. t's actually slightly more complicated than that. Let's see why.

### An Illusion of Movement

I didn't tell you the whole story about rebases. Let's take a small step back. I told you that when you rebase, Git detaches the current branch from its base and moves it to the top of the target branch. But actually, this process cannot happen literally like that. That would be impossible in Git. You cannot detach a commit from another commit and move it elsewhere because commits are database objects, and database objects are immutable. Remember what we said in the beginning of this training: if you change anything in a commit, then you get a different SHA1 hash, which means a different commit. And if you want to move commits around, then you must change at least one piece of data inside the commit, its parent. So you cannot do that. Let's take a step back. Imagine what happens if you change the parent of this commit. The parent's hash is stored inside the commit, so the commit data must change, and the commit must get a new hash. Now that this commit has a new hash, this other commit also has to change because its own parent has changed, so it gets a new hash, and so on for all the commits in the branch. So, Git cannot just move the commits. The commits in the rebase branch must have different hashes, so they must be different objects in the database. In other words, new commits, and indeed, that's what they are. Here is how rebasing really works. When you rebase, Git makes copies of the commits. It creates new commits with mostly the same data, actually, exactly the same data, except for their parents. And also, except for any conflicts that you might have to fix during the rebase. Aside from those changes, these new commits look almost exactly like the original commits, but they are new objects with new hashes. So they are new files with new file names in the database directory. And finally, Git moves the rebase branch to the new commits, leaving the old commits behind. Keep this in mind because as we will see in the rest of this training, sometimes rebases can be tricky, and you can avoid some confusion if you remember that rebasing is an operation that creates new commits.

### Taking out the Garbage

One more thing about this rebasing process. I just told you that rebasing copies in the data in the old commits to create new commits, but what happens to the old commits then? It depends. In this case here, these commits are not very useful. There is no branch pointing at them. The only branch that was pointing at them has moved over to the new commits. So, these all commits are impossible to reach, almost impossible, because there are a few ways retrieve them. For example, if you had written down their hashes, then you could still check out them. But it's more likely that you will just lose track of them. So why would Git waste disk space to keep around commits that cannot even be reached? In fact, Git doesn't keep them around. This is another case like the one we've seen in the previous module when we talked about detached HEAD. We have unreachable objects in the database, and Git will eventually garbage collect them. If I keep working on this project and at some point in the future I look into the Git database, these commits might well have been deleted. As usual, Git doesn't like waste.

### The Trade-offs of Merges

Now we know what the rebase looks like and how it actually works under the hood. However, you might still wonder why rebases even exist. I mean, we already have merging, and rebasing and merging seem to do something very similar. They both take existing commits from one branch and roll those commits into the history of another branch. So if I'm working on the apple pie recipe and I want to also get the spaghetti recipe, I can have both in the same history either by merging or rebasing. So why do we have two ways of doing something similar? And the reason why we have both merge and rebase is that they have different tradeoffs. Let's focus on merge first. The whole point of merging is that it preserves history exactly as it happened. In this case, for example, you can clearly see that the pink commits and the blue commits were created independently, maybe by two developers working in parallel, and then they were merged into one single timeline. If there were any conflicts during the merge, then this merge commit would include the fixes to the conflicts. There is nothing else to understand, really. It's this simple. But merging becomes a bit less simple when you're looking at a large project where it's used a lot. For example, let me show you the Git project for a popular open source library. I'm using a tool called the source tree to visualize the Git history. As you can see, there is a lot of branching and merging going on. Look at this area here. The developers seem to have been particularly merge‑happy in this period, so it can be very hard to follow the way that all of these branches diverged and then converged again. It's hard to understand, for example, which of these commits are contributing to which branches. Compare this graphical tool to the git log command that we've been using so far. In a project such as this one, git log can be misleading. The log is showing history as if it were a single long timeline, one commit after another, but that's not what the project's history actually looks like. It looks like a graph, not a line. The log is squashing the real history somehow, interweaving related commits from different branches as if they were connected to each other while they aren't. So, merges preserve the project history, and in general, that's good, except that the project history can be complicated and confusing, so that's not always necessarily a good thing. But one thing is for sure, merges never lie.

### The Trade-offs of Rebases

Now let's look at rebasing. A rebased history looks really simple and neat. There is no reason for commands such as git log to squash commits into a single timeline because commits are arranged in a single timeline already. So a project that uses a lot of rebasing generally looks more streamlined and clean than a project that uses a lot of merging history‑wise. Essentially, rebasing helps you refactor your project history so that it always looks nice. This neatness, however, comes at a cost. This nicely designed history is not real. It was forged by rebasing, which is a destructive operation. Rebasing create new commits and leaves behind existing commits that might get garbage collected. So a rebased history looks cleaner, but it is a lie in a way. For example, in this case, it looks like the pink commits were created first and the light blue commits were created later on top of them. But this is not what really happened. The pink and blue commits were created in parallel in different branches. So in contrast to merges, rebases change the project history. This might not sound like a problem at all. You might say, who cares what the history looked like originally? Surely you only care about the final result? Well, actually, there are a few situations when you do care about history. There are some advanced Git commands, for example, that become less useful if you tamper with the project history. But most importantly, there is a common scenario when this rewriting of history can become painful. That scenario has to do with distribution, so I have to ask you to be patient. I will talk about it in the next module. For now, just remember this. Rebases make your history cleaner, but they can also cause unwanted side effects. If I had to condense the differences between merges and rebases in just a single recommendation, it would be this: when in doubt, just merge. Rebasing is a power tool. It is quite useful, but you should only use it if you know what you're doing and you understand the consequences. And that's it about merging and rebasing for now. I promise that in the next module I will show you a concrete example of how mindless rebasing can land you in trouble.

### Tags in Brief

This module is mostly about rebases, but I'm cheating a bit. I'm squeezing a few minutes about another topic in here, tags, because tags are one of the features that are getting into a revision control system, so we need tags to complete this layer of the onion. And you might remember that when we talked about the four types of objects in the object database, I said that annotated tags are one of them, and I promised more details. Here they come. A tag is like a label for a commit. For example, let's say that I just released the first version of this cookbook project and I want to put a tag on the commit where the release happens. I can do git tag to create a tag on the current commit and give it a name for the tag, like release\_1. I have an option here, I can create a simple tag with just the name, like a simple label, or an annotated tag, that is, a tag with also additional metadata, like a message, for example. Let's go for the second option. To create an annotated tag, you use this option ‑a for annotated, and we can add a message with ‑m, like we do for commits. First release, still unstable, as far as a cookbook can be unstable, I guess. And now if I type git tag without any option, I get a list of tags. We just have the one in this project. And in the future, if I want to move back to this commit, I can do git checkout release\_1. It's checkout, in this case, because switch doesn't work with tags, only branches. So that's how tags work. But what's a tag under the hood? Let's go to the .git directory. We feel right at home there now. I'll show you where this tag is. If we look into our refs, you remember this is the directory that contains branches, and tags are here in a tags subdirectory. And here is a file named release\_1, our tag. And if we peek inside this file, we find a hash. What hash is this? Let's copy it and use the cat‑file ‑t command. Remember, it means show me the type of this object. And I'll paste the hash in here. So, this file is pointing to a tag object. Let's check the content of this tag object with cat‑file ‑p and this hash. Here is the tag object. It contains information similar to a commit, the tagger, that's me, the date, and the message, and also a reference to another database object, that is a commit. And this is finally the commit that the tag is pointing at. If we do git log, see, this is the commit we tagged earlier. So this might have been more complicated than you expected. Let's recap. A tag is actually two things. It's made up of two parts. One is a branch‑like thing, a reference. Actually, this thing is so similar to a branch that Git places it in a folder just as side branches. But this thing is a reference, so it cannot contain data, just a name. So the reference points to a database object of type tag, and that's where the tag metadata is, like the message. And in turn, this database object points to a commit. I also told you that in Git there are actually two types of tags because in some cases, you might decide that you don't need the metadata. Maybe all you want is a simple label. You want to put a name on a commit. In that case, you can create a simpler kind of tag by using git tag without the "‑a" option. Let's say that I want to tag the current commit with the dinner tag. In this case, I cannot provide a message or any other data. So now we have another tag, there it is, and like before, this tag is another small file in the tags directory. And if we look into this tag, it still contains a hash, only this time, the hash isn't the hash of a tag object in the database, it's directly the hash of this commit. See? So that's the difference between annotated tags and lightweight tags. Annotated tags are a branch‑like object that points to a tag object that points to a commit, while lightweight tags just point to a commit, that's all, because a lightweight tag doesn't need a place to put those additional metadata, so it doesn't need the tag object, it just needs a name. So I told you that tags are similar to branches, but actually lightweight tags in particular are almost identical to branches. Tags and branches are in different directories, and Git knows that it needs to treat them differently. For example, if I'm on a branch and I create a new commit, the branch will move to track the new commit, but the tag will just stay there and stick to the same object forever. So, branches move and tags don't, but I could literally turn this tag into a branch by moving it to this other directory. Don't do it, but you could. And that is all I have to tell you about tags.

### A Version Control System

So, let's recap. Branches, merges, rebases, tags. These are the main features that turn Git from a stupid content tracker into a full revision control system. It took us a lot of talking, two entire modules, half of this training to go through the versioning features, but we finally completed this layer of the onion. Now we only have one more layer to go, and then we will know the entire onion. We'll have the full picture of Git and how it works. So, let's talk about that last layer.

## Distributed Version Control

### Introduction

Hello, and welcome to the last module of How Git Works. We're only missing one last layer in our description of the Git onion, but it's a really important one, distribution. So far, we imagine that there is only one computer in the world, the computer that you're running Git on. Now, let's see what happens if you use Git the way it's used in practice, to share a project across multiple computers.

### A World of Peers

Imagine that you have a Git repository in your computer somewhere. It's this orange repository here. And you also want the same repository is somewhere else, probably on a different machine, here. I made it purple. Now, the machine that hosts the purple repository must be able to connect to the machine that hosts the orange repository. So you might have some technical setup to do here. You have to run and get the daemon process on the orange repo so that the purple repo can connect to it, and so on and so forth. But in this training, we don't care about these technical details, so let's make it easier. I just moved the whole cookbook project to GitHub and removed it from my computer. So now the orange repo is in the cloud, right? A service hosted on the GitHub servers. And the purple repo will be on my own computer. So I want to get a copy of the project on this computer inside this empty directory. You probably know which command to use here. It's the git clone command. It takes the address of a Git repository, which I can copy‑paste from GitHub. There. And now I have the project. All the files are here. But I didn't just get the files; I got the entire .git directory as well and all the files it contains. Here is what git clone did. It created an empty directory for the cookbook, and it copied the .git directory from the GitHub project to this directory. I'm simplifying here. It didn't literally copy each and every file. For example, unless you're using a very old version of Git, git clone only copies one branch, the main branch. If I want to work with the other branches on the remote repo, I need to give specific commands to do so. But that's a detail, an optimization, if you wish. The important part is Git did copy over the objects in the object database. They are in here. After coping this stuff, Git checked out the main branch to rebuild these files in the working area. Remember, the working area in Git is not very important. You can always rebuild it on the fly from the content of the .git directory. And since the .git directory contains the entire repository, now we have a copy of the project and its history on this computer. This is an important point, so it's worth repeating it. Now that we have two clones of the repo, one on GitHub and one on this computer, both clones are equally good. Git is not like subversion or other traditional revision control systems that need a centralized server and everyone else is just talking to that one server. Instead, both computers now contain the whole project in this history. We could have as many of these clones as we want synchronizing with each other. Of course, you can still decide that one clone is the most important one. For example, if you have multiple developers working on the same software project, then you will probably decide that a repo on GitHub is the reference repo, the one that you build the releases from, and everybody must synchronize with that one. That's why I tell you to GitHub the repo on top. You can still synchronize the developers or repos directly with each other, but even then, you probably want to appoint a well‑known reference copy that is visible to everybody and everybody synchronizes with it. However, in Git that's not really a technical issue, it's more of a social issue. It's a convention. From a technical standpoint, all of these clones are peers.

### Local and Remote

Now we have the same project in two separate repos, orange and purple. We're working on purple, so it would be useful if purple could remember the address of orange because we decided that orange is an important copy and we want to stay synchronized with it. Indeed, when we issued the git clone command, Git added a few lines to the configuration of our repository. It's here in the config file. We never looked at this configuration file before, but now that we cloned the repo, we can find some useful information here. Each Git repository such as this one can remember information about other copies of the same repository. Each of the copies is called a remote. You can define as many remotes as you want. But when you clone the project, Git immediately defines a default remote and calls it with a conventional name, origin. Here is the configuration of origin, and it points to the URL that we cloned the project from. The rest of the configuration is more complicated. We don't need to look at the details here. Just know that the default configuration says that we have one main branch that maps over the main branch of the remote. You can tweak this configuration to change the policies that you use to synchronize with the remotes, but this configuration here is the default. So, now Git remembers which other repo or repos we want to synchronize with. But to synchronize, Git also needs to know the current state of origin, which branches are there on the remote, which commits those branches are currently pointing at, and so on. And, in fact, Git does store the information. If we ask Git for branches, Git shows the local branches. We only have main now. But if you list the branches with the ‑‑all switch, then you see all the references, including the ones on the remote, the remote branches and the current position of HEAD. Git tracks the remote branches exactly like it tracks the local branches, by writing those branches as references in the refs folder. If you look inside that folder, you will see an origin folder in here that contains the references to branches, tags, and the current HEAD point of origin. Git will automatically update this information when we connect to remote. There is one wrinkle here. If you look inside this folder, you might find that some of the branches are missing. In this case, I can only see the remote's head here, not the branches. That's because of a low‑level optimization in Git. To avoid maintaining one small file for each branch, Git sometimes compacts some of them into a single file called packed‑refs here. There is no simple command to unpack these files, so you will have to take my word for it that the branches that are not in the refs directory must be in this file. This can happen for both local and remote branches, but in both cases, whether the branches are still individual files or packaged together in packed‑refs, they are still conceptually the same thing. All branches, local or remote, are still references to a commit, and Git tracks all of them. Since we cannot peek inside the files for some of these branches because they've been packed, let's use this plumbing command git show‑ref to see which commit they're pointing at. Git show‑ref main lists all of the branches that have main in their names, which means the local main branch and the remote main branch. And as you can see, they're pointing to the same commit, while the ideas branch is still pointing to an older commit. So, bottom line, you know that the local branch in Git is just a reference to a commit. Well, a remote branch is exactly the same thing. Whenever you synchronize with remote, Git updates the remote branches. Let's see how that synchronization happens in practice.

### The Joy of Pushing

In the very first module of this training, we said that each Git object is just a sequence of bytes identified by hash. I also insisted a lot that SHA1 hashes are unique. I said unique in the universe. Finally, this is the point in our training where we can see how that uniqueness is truly useful. Look back at our two repositories. When we cloned, we copied the objects from the orange repo to the purple repo. Now imagine that we add a few new objects to the purple repo, for example, a new commit and the associated blobs and trees. Synchronization is mostly about getting the same objects on all the clones, but now it's easy for Git to synchronize those objects because each object is immutable and has a unique hash. So, Git will never get confused. It can just copy the missing objects from one repo to the other. Well, okay, it's not quite that simple because copying the objects around is not enough. Git also has to keep the branches synchronized on the various clones, and that's where things get a bit tricky. Let's see how this works. I will make a change to this repo. After some experimentation, I realized that having just a little bit of lemon juice in my apple pie makes it taste even better. I will add this to the recipe and commit it. There we are. So now we have a few new objects in the database, a new blob to represent the file I changed, a new tree that represents the updated project root folder that is appointed to that blob, and this new commit here. And the local main branch is pointing at the new commit, while the main branch on origin is still pointing at the previous commit, of course. Nobody changed that branch yet, and origin doesn't even have this commit, and neither does it have the other new database objects. So, let's send both the new objects and the updated branch to origin. You probably know the command that does that, git push. There we are. Now our new objects have been pushed to the remote, and let's do show‑ref again. The branches on origin moved to point at the latest commit. GitHub updated our remote branches to align with the current state of origin.

### The Chore of Pulling

Now what happens when there are other repos pushed into origin, so the state of origin might change at any time? Now, we cannot just write changes to the remote. We also must read changes from the remote. Things get a bit more complicated here, so I will use a diagram here instead of a demo. Imagine that we have a remote repo that looks like this. It's a single commit. I will use different colors for the commits, and I will not draw trees and blobs. I will skip them because they would make this diagram too busy. When we clone this repo, we get the same objects in our local repo, and here are the branches. I'm assuming that the remote has the default name origin, so the remote main branch is called origin/main. Now let's say that we add one commit, and we push. If the remote's main branch hasn't changed in the meantime, then things are easy for Git. It copies our new commit and the associated object to the remote and then updates the remote's main branch to point at the new commit. And it also reflects the change locally by updating the remote branch origin/main, and everything is aligned again. This is what happened when we pushed our changes a few minutes ago. Now let's do it again. This is the initial situation again. We add the commit, and we prepare to push. But this time, imagine that something has changed on the remote as well. Someone pushed another commit to the remote, this green commit. Now, we can't just push. We have a conflict here. We have two different histories that need to be reconciled. In this case, we have two options basically. One option, which I would not recommend, except in very special cases, is to force a push. We can do that with git push ‑f. That stands for force. That's like saying, I don't care what changes are on the remote, forget about them, and just match my local history. That's the real one. So when we force a push, this happens, and we're probably losing that on origin. Here we're losing the green commit. No branch is pointing at that commit anymore, so it will be garbage collected eventually. We're also creating a very confusing situation for all other people synchronizing to the same remote because now their local history will be conflicting with the history and origin. This is our teammate, Annie, working on the same cookbook repository, and she still has the old green commit in our repository. But after our force push, if Annie tries to interact with the remote, she'll have a conflict. Her history doesn't match the history of the remote, even though she didn't cause the conflict in the first place, so she'd probably be left wondering what happened and how to solve it. So, forcing a push is not a good idea in general. It doesn't really solve conflicts, it just puts the burden on other users to solve the conflict, so it's not going to win us many friends in the team. But we have a second better option. Let's say that we didn't do that first push. Let's go back to the situation we had in the beginning. We added the light blue commit, and we had a conflict because somebody else added the green commit on the remote. What we want to do in general is we want to fix the conflict on our own machine before we push. To do that, we need first to fetch the data from the remote. There is a command to do that called git fetch. We get the new objects from the remote and we also update the current position of the remote branches, origin/main, in this case. Now that we have the new commit and the related objects, we can merge our local changes with the remote changes. So we did the fetch. Now we do a merge. Of course, during the merge, we might have to fix merging conflicts and the like, but the important point here is that we're not re‑writing history. Merges never do that. They just add the new objects. So once we do the merge, our history is the same history as the remote, plus our local stuff, the light blue commit in this case, and we can push that new stuff to the remote without rewriting the remote's history. This is what you do most of the time. You fetch the changes from the remote, you merge them into your own repo, and then you push the result. This sequence of a git fetch followed by a git merge is so common that there is one single command that does both. It's called, you guessed it, git pull, a fetch followed by a merge.

### Rebase Revisited

There is one more important thing to say about this process of pushing and pulling, and it has to do with the rebases. In the previous module of this training, we talked about rebasing. And back then I told you that there are cases where rebase doesn't work well. Now we can finally see why. Say that we have this repo freshly cloned with two branches that are both tracking branches on the remote. I didn't throw the remote branches in this diagram. They're not necessary for this explanation. Just know that we are working on the ideas branch, and we decide that we want the changes from main into ideas. You know that we can do that with either a merge or a rebase. Let's start the rebase this time. So we are on ideas, and we say git rebase main. And Git does the rebase. It copies the green commit from ideas on top of main, and there we are. However, remember what we said when we talked about rebases? This new green commit is not the same commit as the previous green commit. It looks the same, but it's a copy, a different database object. I'll mark it with an exclamation point to tell it apart from the original commit. The original commit will actually be garbage collected at some point. Now, after this rebase, we have a similar situation to the one we've seen a few minutes ago, we have a conflict. We can't just push to the remote because we have different histories on our local repo and on the remote, and we've seen that we have a couple of ways to deal with the remote conflicts. One is to force a push, to do a git push ‑f, but we've also seen that it's almost always a bad idea because it creates conflicts for other users. So now you might think we know what to do, the same thing we did before. We can fetch and merge, or, in a single operation, we can pull, solve the conflict locally, and then push. Only this time if we do this, the result is probably not what you want. Let's see. First, we fetch. There. Second, we merge. There. And look at the history we came up with. Now we have both the original green commit and its copy, the rebase green commit with the exclamation mark in the same history, which can be really confusing. Now if we do a git log, we'll see two commits that look exactly the same with the same message and date and content. And if we rebase the multiple commits, which is a pretty common thing to do, then we'd see an entire line of duplicated commits, which would be very perplexing. So we just defeated our original purpose. We rebase to clean up our local history, but in the end, we got a history that's even more messy and harder to understand. So, forcing a push didn't work, and pulling to solve the conflict, that didn't work, either. In both cases we got in trouble. The bottom line when it comes to rebasing is as a general rule, don't rebase stuff that has been shared with some other repository. It's okay to rebase commits that you haven't shared yet, in general. I do it all the time. But just remember that it's easy to do it by mistake, to rebase commits that you already shared, even if you think you didn't, and then you get in trouble. And that's the major reason why I warned you about rebases in the previous module. Rebases are great, but you should be aware of the way they really work; otherwise, they might land you in a tight spot.

### Getting Social

We're almost done with our discussion of Git distribution, but it's worth taking a few minutes to discuss two distribution‑related features that are not features of Git. Instead, they are features of GitHub, but they are so essential for modern development, especially modern open‑source development, that it's a good idea to talk about them here very quickly, just to avoid confusion. Imagine that there is this project on GitHub that we want to contribute to. It belongs to a user named Pluralsight. We could simply clone this project, and then we could modify it, but we couldn't send our contributions back to the project because we don't have the right access to Pluralsight's repository so we can't push to it. What we can do from the GitHub web interface is to create our own copy of the project on GitHub. This is the first feature I wanted to tell you about. It's called the fork. It's kind of like a clone, but it's a remote clone. We're cloning the project from someone else's GitHub account to our own GitHub account. So now we have a new project in the cloud, and we can clone that one on our local machine. When we do a clone, Git automatically creates a remote in our local repo, pointing at origin. So, just to be clear, origin is our own GitHub project. Our fork, not the original project. Actually, from Git's point of view, there is no connection at all between our project and the original project that we forked from. GitHub does know that the two projects are connected, but Git doesn't. So if we want to track changes to the original project, we need to add another remote pointing at it. This is not something that Git does automatically; we have to do it ourselves. A common convention is to call this remote upstream. Now we have our local project with two remotes. We can work on it, and we can synchronize all our local changes with origin. If we commit to local changes, we can just push those changes to origin. If there are changes on upstream, we can pull them into our local project, solve many conflicts, and then push them towards you. One thing that we still cannot do, however, is to push changes to upstream. For example, we might like to contribute our green commit to the original project, but we still do not have the right access to upstream. So, GitHub gives us an alternative. We can send a message to the maintainers of upstream and ask them to pull our changes. This is called a pull request. Once again, pull requests are not a Git feature. They're not even a version control feature, strictly speaking. In a way, they are a social network feature. You're just sending a message to people. If those people like your changes, then GitHub makes it easy for them to do a pull and get your changes from origin. And other services similar to GitHub, like GitLab, for example, have their own version of the same feature. And that's all about forks and pull requests, two of the most important tools in modern open‑source development.

### The Whole Onion

At the beginning of this training, I promised that by the end of it, you would understand how Git really works. Congratulations, now you do! To get here, we started right from the core of Git, a simple map of hashes to objects. Then we looked into those objects and we got to the point where we could see Git as a stupid content tracker that tracks changes to your files and directories. From there, we moved down to the revision control features of Git. We talked about branches and merges and rebases. And finally, we looked at the distribution‑related features of Git that are probably the main reason why you use Git in the first place. And there it is, the whole onion. We wrapped our heads around the entire thing. This is all you need to go on exploring Git on your own. Sure, if you're planning to use Git a lot, then there are still plenty of things that you might want to learn about more advanced features, dozens of command‑line options, okay, actually, hundreds of command‑line options. But now that you understand the fundamentals, you can confidently keep learning without fear of getting confused. And if you liked this training, you might want to do two things. First, rate this training on Pluralsight. And if you have a minute, even leave me a comment here with feedback. I love ratings, and I love feedback. And also you can move on to this other training that I published here on Pluralsight. It's called the Mastering Git, and it goes a few steps farther than describing the Git onion. In a sense, it tells you how to cook the onion. It explains more sophisticated commands and techniques that you will want to know when you're using Git in your daily job.