## Embracing the Command Line

### CLI vs. GUI

Throughout this course, we'll be interacting with Git entirely from the command line. If you feel more comfortable using a graphical user interface, you can certainly run one alongside the command line as you follow along. However, I highly recommend that you make the command line your primary tool when using Git. Why? Because graphical interfaces, regardless of how well they're designed, always impose some kind of limitation on what you can and can't do with it. Generally speaking, people tend to fall into two camps when it comes to the way they prefer to interact with computers. On one side, there is the people who, given the choice, will always pick a command‑line interface, and on the other, the people who would rather use a graphical user interface, whichever you feel more comfortable with is largely a matter of taste. However, it definitely is the case that for certain tasks one is more suited than the other. For example, it would be slow and impractical to browse the web using a text‑based browser like LINQ. By the same token, when it comes to Git, the command line is a lot more efficient than any graphical user interface. Does this statement sound controversial? Well, considering what I said earlier about taste, it certainly might be to some degree. But to prove my point, let's take a look at a fairly‑common Git GUI. Does anything in particular stand out to you? Like many other tools of the same kind, it presents the user with toolbars and menus packed with commands and options. These applications are supposed to make it easier to use Git without having to invest a whole lot of time in learning how it works. However, in reality, if you don't know what to choose, there is very little they can do to help. If you think of Git commands and options as a domain‑specific language, one that allows you to manipulate the history of your source code, it becomes obvious how any attempt to make it fit into some sort of graphical interface works against its text‑based nature. It's destined to become bloated to the point of becoming a limitation rather than an empowerment, that's why it makes more sense to use Git from the command line. Much like when using a text editor, everything is only a few characters away without forcing any particular workflow. The trick to become efficient is to understand how the different commands work, how they affect history, and learn when it's appropriate to use them.

### The Tools

Just because we'll be using Git from a command prompt doesn't mean that we'll have to sacrifice user friendliness. There are a few utilities designed specifically to make Git easier to use from the command line. For example, they enrich the command prompt with updated information about the repository. Most of them also provide auto completion so we don't have to type as much. We look at a couple of these utilities in the beginning of the next module. Of course, we're going to need to have Git installed on our machine. It doesn't have to be a specific version, anything from 1.8.0 and above will work just fine. Since staring at a command prompt alone can sometimes become quite boring, we'll also be looking at a real‑time graphical representation of our Git repositories. Every time we run a command, the graph will show us how it has affected the underlying history. This kind of visualization really helps to understand how the commands work at their fundamental level. With that knowledge, it then becomes easier to recognize in what situations they're useful and how to use them appropriately. At this point, we should be all set. But before we dive into the advanced stuff, let's take a moment to go through some Git fundamentals.

### Git Fundamentals

One of the common complaints people have about Git is that it's too complicated. You may have heard someone say that Git is hard or, in some cases even, Git to me is harder than programming. These complaints are understandable. Git can, in fact, be rather intimidating at first, with its scary sounding commands and a huge set of options and parameters. However, beneath that unfriendly surface, Git's foundation is actually very simple. So simple, in fact, that we can summarize it using three basic concepts, commits, snapshots, and references. Let's look at each of them individually and how they relate to each other. Every time we commit some changes, Git creates a so‑called commit object. The commit contains information about its author, the timestamp when it was created, and most importantly, a reference to the commit that came before it, its parent. This snapshot represents the state of the directories and files in the repository at the time the commit was made. Internally, Git represents each directory with an object called a tree. Under a tree, there can be other trees or the actual files, which are called blobs. You can directly point to any commit, tree, or blob by its unique ID, which consists of the computed SHA1 hash of the object's contents. While having these IDs is sometimes pretty handy, if we were to use them all the time it would quickly become bothersome, so Git offers a better way to refer to commit, by using so‑called references. A reference is just a name that points to a certain commit. You can think of it as a symbolic link or a bookmark in case there are three kinds of references. A tag is a fixed reference to a specific commit, it never changes. This is commonly used to mark the commit associated with a certain version of the software that's in the repository. A branch is a reference to the latest commit in a line of history. Every time we make a new commit, the branch reference is updated to point to it. The history of a branch can then be recreated by starting from the latest one and track backwards following the trail of parents. Lastly, head is a special reference managed by Git to keep track of the branch or, in some cases, the commit whose snapshot corresponds to what's in the working directory. And that is Git's object model in a nutshell. As you see, there is nothing really complicated about it. The complexity everyone talks about lies entirely in the multitude of commands and options built to manipulate this object model in various ways. This kind of complexity has nothing to do with the essence of the problem that Git is trying to solve. Instead, it's accidental, a by‑product of design choices and collaboration that could have been avoided in the first place. Therefore, keeping a mental model of Git fundamentals really helps surpass this complexity, making even the most complicated command look trivial, and that's exactly what we're going to do in this course.

## Elevating Your Command Line Experience

### Command Line Utilities

In this module, we'll look at how to work efficiently with Git from the command line. We'll look at a few utilities designed specifically to make it easier to use from the command line. Next, we'll see how we can do more and type less by using aliases. Finally, we'll explore interesting ways to visualize the contents and history of our repositories without ever leaving the command line. Let's get started. Just because we use the command line doesn't mean we have to give up on user friendliness. A few utilities do exist to help being more efficient when using Git from the command line. The choice of which utility to use depends first and foremost on the operating system you're on. I'm going to present you with two very good options, one for Window and one for a Unix‑based operating system, such as Linux or macOS 10. If you're using Windows, your best option is to use Git from PowerShell using a module called posh‑git. You can easily install posh‑git by cloning it from its GitHub repository and running the script install.ps1. If you have PS Git installed, it becomes even easier since you can simply say Install‑Module posh‑git. From that point on, every time you see the inter directory that contains a Git repository, posh‑git will automatically enrich the command prompt with useful information, such as the name of the current branch and a number of added, modified, and deleted files, both in the working directory and in the index. If you're using Bash on Linux or macOS 10 and you're only interested in auto completion, there is a simple script in Git source code that will do just that, all you have to do is downloaded it and add it to your bash profile. That way, it will automatically become available in all new Bash sessions. If you're like me and prefer something a little more visually appealing, then I highly recommend that you replace Bash with Z shell and install a set of utilities known as ohmyz.sh. Ohmyz.sh can then be installed by downloading and running a shell script from the project's GitHub repository. This adds the same functionality for Git as posh‑git does for PowerShell, however, with a bit more style. Another option, and this has been my personal favorite for a few years now, is Starship. Starship looks good, is easy to configure, and is also cross platform, so you'll have the exact same experience regardless of which operating system you're on. Installing Starship is as straightforward as with the other command line prompts we've seen so far. You simply download a shell script and it will take care of installing it on your machine. Of course, these are not the only options. There are plenty of command‑line propts out there, which make different tradeoffs in terms of performance, features, and ease of use. We'll be using Starship throughout this course.

### Aliases

Sometimes, even without auto‑completion on, you just can't seem to type fast enough. For those situations, having short aliases for the most common Git commands can be quite handy. Defining an alias is easy. You simply say git config alias., the name of the alias, followed by the command that you want to map the alias to. In this case, we create an alias for a status command called st, so now we can write git st. Aliases are stored in a dedicated section in Git settings file. Here is how it looks like. As with any other setting, you can define aliases that apply only for the current repository, all the repositories that belong to the lockdown user, or for the entire system by passing the correct switch to git config. Git config ‑‑global, for example, creates the alias in the git config file located in the user's home directory. Of course, you can create an alias not just for a simple command, but also for a combination of commands and options. Here is a slightly more useful version of our previous alias. This one is less verbose than the standard status, and therefore, it's easier to read. Another alias that I use all the time is commit ‑‑all ‑‑message, which means commit all modified files using the following message. (Working) Aliases can also contain an entire sequence of commands and even accept parameters, for example, they can be associated to an entire shell function and do all kinds of things, like manipulative strings, invoke other programs, and so on. You can really take them as far as you need them to go. Here's an alias that I use fairly often and that demonstrates this capability. Dm stands for delete merged. As the name implies, this alias allows us to quickly delete all local branches that have already been merged into the current branch. Let's build it together piece by piece. Exclamation mark tells Git that what follows should be interpreted as a shell command. Next, we declare a shell function called f with no parameters. Inside that function, we run git branch des ‑‑merged. That command returns a list of branch names, which includes the name of the current branch prefixed by a star. Of course, we don't want to delete the current branch, Git won't let us anyway, so we pipe the output of git branch into grep, where we use a simple regular expression to exclude the line that begins with a star. We then pipe the filter list of branch names into xargs, where for each of them, we invoke git branch ‑d, which deletes a local branch. Finally, we invoke the f function we just defined, and here it is in action. Pretty handy, don't you think?

### Pretty Logs

When we want to look at the history of the current branch, we say git log. This way of visualizing a sequence of commits is fine, but we can do better. Since history is something we'll be wanting to look at all the time, every tiny bit of improvement we can make is worth the effort. In fact, the log command supports an option that allows to format the list of commits in different ways. For example, if we would like to limit the output to only one line per commit, we could say git log ‑‑pretty=oneline. This is definitely easier to read, but it's missing some important information. All we get is the SHA‑1 hash of the commits along with their messages, and that's pretty much it. What if we could specify the exact pieces of information we want to include and the order in which we want to see them? As it turns out, the same option can do just that. Here is an example. (Working) Let's break down this format string. Percent h represents the abbreviated commit hash, %d will show any reference currently pointing to the commit, %s contains the first line of the commit message, also known as the subject. Percent cr represents the timestamp when the commit was made relative to now. And finally, %an contains the name of the commits author. You can find a complete list of all available placeholders at this URL. Now, this output is short and expressive, but it's also quite boring to be honest, what it needs is some color, let's go ahead and add it. (Working) That's much better, but we aren't done yet. As a final touch, let's include a graphical representation of the branches by adding the ‑‑graph option. Of course, we wouldn't want to type all this every time we want to look at history, so let's make an alias for it, shall we? (Working) Now this is what I call a pretty history.

### Pretty Diffs

Besides history, the next thing we'll be looking at just as often are the contents of our working tree, the index and specific commits. For that, we use the diff command. What we see here are the changes currently in our working tree displayed in the universal diff format, pretty common stuff. However, believe it or not, there is a lot we can do here to improve the command line experience. By default, Git redirects the output of commands that might produce more text than would fit in one screen to a program called less in order to provide paging. We can also scroll through the text using the K and J keys to scroll up and down. Once we're done, we go back to the command prompt by pressing Q. What's interesting, is that less is not the only option when it comes to paging. In fact, Git has a setting that allows you to configure which program it should send its text output to for paging, and that option is called, unsurprisingly, core.pager. If core.pager is not set to anything, Git will default to using less. However, there are a number of programs out there that are designed specifically to improve the way diffs are displayed in a terminal. My favorite one at the moment is Delta. Delta takes the output of git diff and presents it in a much cleaner way, with support for keyboard navigation and even syntax highlighting for a variety of programming languages. Delta is also cross platform, which is a big plus in my book. You simply have to tell git to use D0elta as its pager by setting the core.pager option to delta. (Working) Delta ofers a bunch of configuration options to further customize the experience of doing diffs, so I would strongly encourage you to check them out. We have also a few different options in regards to how Git generates the diff itself. According to the unified diff format, a line of text is always either added or removed. Even if only part of a line is modified, it will be displayed as if the old version was removed and a new one added. That's fine in most situations, but certain kinds of changes are better visualized in line. The diff command is versatile enough to allow us to do just that by passing the ‑‑word‑diff option. (Working) By default, git diff shows exactly three lines before and after each block of changes according to the unified diff specification. However, we can increase the threshold with the ‑‑unified option. This time, we see 10 lines on each side of the change instead of just three. These two options, ‑‑word‑diff and ‑‑unified, are especially useful when looking at rows, for example, documentation or a blog post. Sometimes git‑diff doesn't produce the output we expect. Consider this patch, for example, we can see that a few lines have been modified, but it's hard to find any coherence among them. Since the default diff algorithm used by Git favors speed over readability, it sometimes misinterprets which portions of the file belong together, making it nearly impossible to understand at a glance what the change means. This can become a problem when we're manually editing patches during a partial commit or an interactive rebase. In order to remedy these situations, git offers a couple of different diff algorithms that we can use to produce better results, one such algorithm is called patience. The patience algorithm is named after the fact that it trades speed for accuracy. In other words, it takes its time to correlate groups of matching lines to find out which belong to the same chains, this produces a much more readable diff. You might not think that this was any slower than the previous example, but processing large files using the patient's algorithm might take significantly longer, for that reason, there is a faster version of it called histogram. (Working) Now, it would be nice if Git could use this every time it has to generate a diff and not only when we explicitly ask for it. Fortunately, we can set it as a configuration setting. (Working) Next, we'll look at how we can take advantage of the diff command to compare entire commits or even individual files within commits in our Git repo.

### Inspecting Commits

There are a couple of different ways to look at the changes introduced by a specific committee. First, there is the show command. Of course, we can specify any valid reference here. Like for example, where HEAD~2 refers to the second parent of head. Now, let's look at the output. The first section contains information about the commit object itself, it's SHA‑1 hash, its author, the timestamp when it was created and the message associated with it. The second section shows the difference between the snapshot referenced by the commit and the one referenced by its parent. The patch is formatted according to the unified d format, similarly to what you will get by running git diff. The show command accepts the same ‑‑pretty option as git log does, that way we can provide a custom visualization for single commits as well. Of course, we wouldn't want to type this all the time so we make an alias for it. Let's call it show object. (Working) If we're just interested in reading the commit metadata, we can do that by suppressing the diff output. (Working) Or, if we'd like to include a summary of the changes, we will specify the ‑‑stat option. Now, with markdown having become the de‑facto markup language for writing on the web, it come as no surprise that it found its way on the commit messages of many open‑source projects, as they're being viewed on websites like GitHub. Wouldn't it be nice if we could look at nicely‑formatted commit messages in our terminals, too? Well, it turns out there is a tool that can render marked down content inside a command line interface called glow. Here is an example of how it works. Really neat. Once we installed the version for our operating system, all we have to do is write an alias that uses git show with the ‑‑pretty option to extract the bits of the commit metadata that we want to display and then pipe that output to glow. Let's go ahead and do that. We'll call our alias.doc, as in document. (Working) There are a few important things to notice here, %s represents the first line of the commit message, the subject. We prefix it with the hashtag character in order to turn it into an h1 header with markdown. Percent n is a line break, we add two of them here to create an empty line after a title, as is the convention in markdown. Percent b represents the rest of the commit message, the body. Any markdown found in there will be rendered just as it is. Percent an is the name of the commit author, while percent as is the short date when the commit was authored. We surround that with underscores to format it in italics. The bigger‑than sign tells glow that the whole line should be formatted as a quote. Next, the reference that points to the commit we want to look at can be passed as an optional parameter to the alias. The syntax over here expands into the first argument provided to the function. If no argument is specified, the default value will be head. And finally, we pipe the output of git show to glow for rendering. Here is what our commit message formatted using markdown would look like in the terminal. Pretty nice, don't you think? Another way of looking at the difference between two commits is by using the diff command and specify a range of commits using the double‑dot notation. We can see that the snapshot in head adds two new files compared to the one two commits ago. Finally, git diff allows us to compare even individual files within different commits. For example, we could compare the README file in the working directory against the same file referenced three commits ago. As you can see, using the git commands directly is far more versatile and flexible than any GUI we could ever hope to be.

### Summary

In this module, we looked at how we can work efficiently with Git from the command line. Then, we saw how we can do more with less typing by using aliases. We saw how we can create analysis for basically anything from simple commands to entire combinations of commands, parameters, and options. Finally, we learned how we can use git log, diff, and show to enrich the way we look at the history of our repositories, the contents of the working directory, and individual commits. in the next module, we'll start creating commits, but not just any commits, we'll be creating commits that are crafted in such a way to turn the history of our repository into a journal that's self‑explanatory and easy to follow.

## Keeping a Clean History

### The Importance of a Clean History

In this module, we'll look at how to take advantage of Git's unique features to craft beautiful commits that respect the history of our source code. We'll start out by defining what it means to have a good looking history and why that's important. Next, we'll see how we can use features like the index, the stash, and commit hooks to carefully organize, verify, and properly document the contents of our commits. Finally, we'll see how to line up our local history to create a trail of commits that document our work in a way that's self‑explanatory and easy to follow. Let's get started. Study the past if you would define the future, said the ancient Chinese philosopher, Confucius. As with many other things in life, the way you move forward in a code base starts by understanding how things got to where they are in the first place. This precious information is captured by the version control system. But there is a catch, simply storing the history of our code base in a version control system doesn't necessarily mean that we'll be able to gather any value out of it. If the history is complicated, ambiguous, and poorly documented, it turns into a black box, keeping that precious information locked inside unaccessible. If instead, it is clear and easy to follow, it becomes the key to understand the decisions made by the programmers that came before us. The designers of Git understood this, that's why they built special features into Git to give us a fair shot at maintaining a good looking history. So, what makes a history good looking? Well, it all starts with the quality of our commits.

### Anatomy of a Good Commit

A good commit usually has four recognizable traits. It's atomic, consistent, incremental, and documented. Let's look at each of these qualities individually. A commit must be atomic or, in other words, self contained. This means that we shouldn't split semantically‑related changes across multiple commits. For example, if we were to rename a function, we would commit the renamed function, as well as all the references where the function was used into one single commit. Related to being atomic, there is the concept of coherence. Just as we should avoid the breaking apart thematically related changes, we should also make sure that each commit represents one logical change. Renaming a function, along with all its references, represents one commit, fixing a bug represents another one. Each committee should leave the code in a consistent state. At the very least, the code should compile with no errors and no broken tests. The reason why this is important is because it should be possible to apply individual commits to the working directory and be able to immediately build on top of them without first having to deal with compilation errors or failing tests. A code‑base evolves through a sequence of self contained and logically coherent modifications that build on top of each other incrementally, that's why the order in which the commits appear in the line of history matters. For example, if we were to build a feature, the order of our commits would reflect the evolution the code went through as the feature was implemented. So the order of the commits shouldn't be arbitrary, but rather it should be explanatory. In other words, it should clearly document the thought process the programmer went through as they worked on the code base. Speaking of documentation, a very important piece of information about the meaning and role of a change is the commit message, we should use it to communicate not only what a commit means for the system, but also the reasoning behind it. If it's not immediately obvious, that's why a good commit message is made up of two parts, a short summary in the form of a single sentence that describes what the change is about and a longer description containing more details about the change should it be necessary. Here is an example of a useful commit message from Git's it's own history, this is one of the very early commits in the Git source code. As you can see, the author has described what the patch does in one short sentence. But he didn't stop there, he used the next paragraph, the body, to describe how to use the new options of the update‑cache command. This is a great example of how useful it is to provide the background on the thoughts that went into modification, just imagine what a huge time saver this information will be in the case of a bug or in future design discussions. The way this commit message is formatted is also very deliberate. We'll talk more about how to properly format or commit messages and why it is a good idea to follow a common convention later in this module. At this point, you might have noticed that the properties of a good commit make up the acronym ACID, just like the one used when talking about database transactions. This is not a coincidence. A virtual control system is, in fact, a form of database designed to store information about the changes that happen over time in a collection of files. While these principles can be applied to just about any version control system, in practice, the vast majority of them don't offer the level of granularity and control that's necessary to maintain a good looking history. Git, on the other hand, was built from the ground up for exactly that goal. During the rest of this module, we'll look at how we can use Git's unique features to make sure our history is as clean as it can possibly be.

### Atomic Commits

A single coherent change, that's all a commit should be. It should represent one kind of modification, whether it be a piece of documentation, a refactoring, a bug fix, or a new feature. Following this rule with Git is easier than with any other visual control system thanks to the index. The index, also known as staging area, is one of Git's most distinctive features. According to the programmers who were involved with Git during its early days, it didn't take long before it became clear that having the possibility to pick and choose which changes to include in the next commit was a very useful feature. It was very obvious from the early days that it was very useful that you can trust git commit, after preparing the index with what is and isn't to be included in the commit, won't pick up debugging craft you keep around in the working tree, that's exactly what the index is for. Let's say we have two modified files in our dirty working directory, one of them changes a word in a documentation file, while the other modifies a piece of code. Now, we wouldn't want to have both of these changes in the same commit since they are totally unrelated to each other and the commit wouldn't be atomic. Thanks to the index, it's easy to create two separate commits, one for each change. First, we tell Git about our intention to create a commit with only the documentation file by adding it to the index. If we want to see what changes are going to be part of the next commit, in other words, what's staged, we can say git diff ‑‑staged. As you can see, only the README.md file is currently in the index. If instead we say git diff, we see the changes that are only in the working directory. Now, let's try something interesting. Let's say we realize that we forgot to make another change to the same file and we want to include that in the next commit as well, we open up our editor and make the change. (Working) Now let's go ahead and check the contents of the index, our latest change isn't there. Let's see what's in the working directory. The README.md file was indeed modified, but it wasn't automatically included in the index. This is by design. Once we add a file to the index, git creates a cached copy of it as it was at the time it was added. In fact, we can also check the contents of the index by passing the cached option, which yields the exact same result. If we want to include any modifications in the index, we have to explicitly say so by adding them. This might seem counterintuitive, but it's actually very useful. Here is how Linux Torvalds explained this design decision. It's simply how I've always worked. I tend to have dirty (working) trees with some random patch in my tree that I do not want to commit. We could also keep modifying the same files in the working directory without affecting our previous decision. Knowing that a staged file is cached, gives us the peace of mind and the ability to experiment without affecting our upcoming commit. Now, let's consider this, what if we made two different modifications to the same file, but wanted to commit them separately? In that case, we can take advantage of the index to include part of the file in one commit and the rest in another. The ‑p option means patch, and it allows us to interactively choose which sections of the patch to states, also known as hunks. The help text below lists what we can do. Y means yes, stage this hunk, n means no skip this one, s means split the hunk into smaller ones if we need it to be more granular. In this case, we just want to commit the add function separately from the comment. Since both changes are part of the same hunk, we first need to split them by pressing s. At this point, we can confirm that we want to stage the first hunk that is the add function by pressing y. Finally, we exclude the comment by pressing n. Since we're doing a partial staging, we can see that the calculator.c file is simultaneously stage and unstaged. It might be good to know that the same option also works to selectively undo individual changes from your working directory. If you run git checkout ‑p, git will ask you which hunks you want to discard from your working directory one at a time. The actions you are offered here are the same as with git add ‑p, but their meaning is reversed. In this case, y means that you want to remove a change, while n means that you wish to keep it. Pretty handy, indeed. If we have a very dirty working directory, we also have the possibility to do a so‑called interactive staging where we get an overview of all the modified files and can decide which files to add entirely and which to include partially. This is something I very rarely use since the interactive patch editor is usually good enough for most situations, however, it's useful to know it's there when you need it.

### Consistent Commits

Once we're done deciding which files are going to be part of our next commit, it's useful to verify that what's in the index is consistent. It shouldn't contain anything other than what we intended, nor should it introduce any errors in the code base. As programmers, one of the first things we check for is the presence of unintended whitespace, for that, we can simply use the diff command. Git will automatically highlight all the invalid whitespace directly in the path's output. We could also check for whitespace servers by passing the ‑‑check option to diff. In this case, git diff will output the lines that contain invalid whitespace and exit with a non‑zero status code if it finds any, which is particularly useful when used in scripts. What places are considered invalid for whitespace to be in is controlled by the core.whitespace configuration option. By default, Git will look for rogue whitespace in three places, indicated by these options. On top of that, we can also tell Git to look for spaces within indentation by using these options. Tab‑in‑indent checks for tabs used for indentation instead of spaces. For example, this will teach git to make sure that indentation is done exclusively with spaces. As you can see, Git is now telling us that line five is indented using tabs instead of spaces. Once we are satisfied with the contents of the index, it's time to verify that it actually works. The code we're about to commit shouldn't contain any compilation errors nor failing tests. But how can we verify the contents of the index alone separated from all the other changes that are in the working directory? Well, we can do that by taking advantage of the stash. The stash is a storage area where we can temporarily put unfinished work that we wish to take out of the working directory. For example, we've added two new functions, one for addition and one for subtraction. We want to commit this function separately, so we stage the add function first while we leave the subtract function out of the index. Now, a stash will normally include both the modified files in the working directory, as well as the contents of the index. In order to make sure that the contents of our next commit are consistent, we can stash away the changes in the working directory, while at the same time leaving the index untouched by saying git stash ‑‑keep‑index. If we had any new files that weren't previously part of the history, that is they are untracked, we would add them by including the ‑include‑untracked option. We also have to provide a message for the stash. Since this is just temporary, we can simply write work in progress, or WIP. The end result is that now we have a working directory that only contains the changes that are about to be committed, without losing the rest of our work. At this point, we can run a build script to verify that the patch is going to leave the code in a consistent state, and if so, proceed with the commit. In this case, we have a Makefile, so we can simply run make. (Working) Once we're done, we restore the files from the stash by saying git stash pop, and our changes are now back, safe and sound, in our working directory.

### Incremental Commits

It's important to leave a trail of commits that shows our thought process as we made our way through the codebase. For example, if we wanted to add a new feature, we might want to show how we went about it incrementally step by step, in this case, the first commit could be a refactoring that makes room for the new feature. The second committee will be writing a failing acceptance test and ignore it for now. The third would be actually implementing the feature. Note that each committee should leave the code base in a consistent state. The code should compile and all tests should pass, that's why we ignore the failing acceptance test. We want to communicate how the feature is supposed to work without breaking the test suite. However, keeping a tight sequence of commits while working on something is very hard. Some people are able to pull it off, but for most of us, the creative process is a little faster, it's littered with missteps, experiments, and changes of course. Fortunately, Git, being a distributed version control system, allows us to separate the history that exists only in our local machines from the one that we share publicly with the world. As long as we haven't shared our commits with anyone else, we're free to change their contents, messages, and order at the best of our liking. Once we've published them to a remote repository however, they become final, we are no longer allowed to change them. We look at the difference between private and public history and why we can rewrite one but not the other later in this model. One common way to clean up our local history before publishing it is by doing a so‑called interactive rebase. We started with the git rebase ‑i command. HEAD~4 refers to the commit where we want to start writing history. At this point, Git opens up our editor with a list of commits that fall within the specified range, offering us a series of actions we can take on each of them. For example, we might want to reorder them and merge together, or squash, multiple interim commits into a larger one. We can also change the commit messages or completely edit their contents. Once we're done planning our actions, we save the file and exit. At this point, Git will walk the sequence of commits starting from the first one in the range and stop whenever we said that we wanted to take an action. The first stop is the squashing of two commits. Here, we get a chance to edit the commit message to better describe the changes in the resulting commit. Second, we stop to reward a commit message. And last, we stop to completely add it the contents of a commit. (Working) Once we're done editing the file, we make it part of the same commit we stopped at by amending it. We'll talk more about how to change the contents of a previous commit with the ‑‑amend option in model seven, which is all about fixing mistakes. Finally, we target to continue with the interactive rebase. The resulting history is now much more self‑explanatory and easier for other people to follow since it reflects the incremental progress we made through the codebase. When rewriting history, it's important to make sure that the new commits are still consistent, one quick way to do that is to use git rebase with the ‑x or ‑‑exec option. This option lets us specify a shell command that Git should execute after each commit in a given sequence. In our use case, we want to run the build process to verify that none of our written commits introduce any problems. As you can see, the todo file now contains our make command after each and every commit, along with the exec instruction. After verifying that the commits we're interested in are included in the sequence, we can save the file and exit. At this point, Git will proceed to stop at each commit and run make. If a commit happened to break the build, git will give us a chance to fix the problem and amend the offending commit. This time, everything looks good. Rebase really is Git's Swift Army Knife when it comes to rewriting history.

### Documented Commits

Making sure that the changes in our commits are atomic and consistent isn't enough, we also need to properly document them. A good commit needs a well‑formed commit message that explains what the patch does to the code base. Git has a convention for how a well‑formed commit method should look like. It should consist of two parts, one, a short one‑sentence summary with maximum 50 characters in length, and two, an optional longer description that adds more detail about the change. For example, it could be the reasoning behind the refactoring, the problem a back fix is solving, or instruction on how to use a new feature. It's also good measure to wrap the lines at 72 characters to make it readable on a standard 80‑character console. Even with all good intentions, we all know that writing commit messages isn't something that comes natural to many programmers, someone even created a website dedicated to making the job easier by offering generic, pre‑generated commit messages. If this is where things are at, we need to find a better way to encourage descriptive and well‑formed commit messages. One way to address the problem is by creating a little reminder every time someone is about to make a commit. The reminder could take the form of a shell script that runs every time a commit is created in the local repository. The script will check the commit message to make sure that it's well formed, if it isn't, it will inform the user and ask them whether they want to correct it. In order to have Git run the script automatically, we could attach it to the commit client‑side Hook. Now, writing a script like that probably wouldn't be too hard. However, I happen to have already written a Bash script that fits the description exactly so we can use that to save time. Let's go ahead and download it into the .git/hooks/ directory in our repository and call it commit‑msg. Since it's generally not a good idea to blindly run scripts you found on the internet, let's take a moment to look at it. As you can see, the script is pretty straightforward. First, we check the length of the first line of the commit message. If it's longer than 50 characters, we print out a message to the user asking them whether they want to correct it. If they press Y, we open up the entire commit method in the default editor. Finally, we exit with a zero status code which tells Git that it's okay to continue with the commit. Now, let's grant all users the right to execute a script by using chmod. Finally, we can try it out by attempting to create a commit with a message that's too long. (Working) As you can see, the script is informing us that the summary of the commit method should be at the most 50 characters long. If we press Y, our default editor will open up allowing us to write a well‑formed commit method. Notice that Git goes as far as configuring the editor to indicate when the first line of a commit is longer than 50 characters. Now, this is called dedication. You can download the script using this demo at this URL and modify it to your liking. The one thing we can't do is to automatically check for descriptive commit messages, for that, we count on code reviews to provide appropriate feedback.

### Private vs. Public History

In Git, there is this convention that says that we are never allowed to rewrite public history only our own private one. The notion of public versus private history applies to any distributed version control system. In this model, everyone works on their own local copy of the repository building their private history. Once they're done, they agree to share their work with each other through a common instance of the repository, which everyone has access to. Once someone's private history is part of a shared repository, it's no longer only theirs, it becomes public. So, whenever we decide to tidy up our history, we only do it on our own private history, never on a public one. I think Git's original creator, Linus Torvalds, said it best. People can (and probably should) rebase their private tree their own work. That's a clean up. But never other people's code. That's a destroy history. But what does it mean by never replace other people's commits? It means two things, first, if you didn't create a commit, it's not for you to change. And two, if you've pushed a commit to a share repository, other people might pull it and built on top of it so it's no longer yours and you can change it. But why do we need this rule at all? Well, consider this, every time we change any aspect of a commit, we indirectly modify its unique ID. This ID is generated by calculating SHA1 of the SHA1 hashes for its metadata fields combined. These fields are the three objective references, the parent commit, the author, the committer, and the commit method, changing any of these fields is going to affect the commit's own ID. Now, since the idea of the parent commit is part of it as well, this means that once a commit changes ID, all commits that come after it also changed, like a domino effect. Now, if the old commit had been fetched by someone else, once they pull the modified ones, Git is going to treat them as completely different commits simply by virtue of having different IDs. Now, imagine if someone had added new commits on top of the old ones in their own local repository, now things are going to get even more complicated. Git is going to merge the old secrets of commits with the new ones. Now, nobody can tell what has actually changed. In other words, chaos is going to ensue so stick to the golden rule. If a commit only exists on your local machine, change it as you like. If you've shared it, it's final.

### Summary

In this module, we've learned the importance of paying attention to the way we shape the history of our repositories. We have determined that good commits are ACID, Atomic, Consistent, Incremental, and Documented. For the rest of the module, we have looked at how to take advantage of some of Git's unique features, namely the index, the stash, and the ability to rewrite history to prepare, verify, document, and line up our commits into a trail that's self‑explanatory and that can act as a journal of our work for generations to come. In the next module, we'll look at how to follow this trail by answering questions such as which committee introduce a particular change or how a committee has moved across branches.

## Searching Through History

### The Concept of Reachability

In this module, we're going to look at how to query the history of our source code to answer any questions we might have about its past or present. For example, what commits are in this branch, but not in that other one, which commit to introduce this line of code, and who modified that file during the past few weeks? These are only few of the questions we'll be able to answer by using Git's built‑in search commands. Just a little heads up, this model is a bit more information dense than the ones you've seen so far. If at any point you feel overwhelmed by the amount of technical details, don't worry, you don't have to absorb all of the information all at once. It's okay to take your time to pause and reflect on what you've just seen before moving on to the next concept. Also, you don't have to commit, no pun intended, everything to memory, just being aware of the fact that these features exist is enough to give you an advantage when you find yourself in a situation where they might come in handy. Now, with that said, are you ready? Let's dive in. Before we can start talking about searching the history of a repository, we need to take a step back and remind ourselves of Git's object model. In Git, history is structured as a directed acyclic graph. Visually, you can think of it as a multiple series of nodes, where each node always points to the one that comes before it. In the context of Git, each node represents a commit. The last commit in its sequence is called a head and it's associated to a name, the branch reference. Now we said that each commit has a reference to its parent. Most commits only have one child, but there is a special kind of commit that can have two or more. These commits are called branching points or fork points. Likewise, there are commits that can have more than one parent, these commits are called Merge commits. Now, the only way we can traverse this structure is by starting from the latest commit in a branch, the head, and work backwards following the trail of parents. Whenever history diverges into two branches, like in the case of M, we can choose to either follow its first parent, that is, stay on the branch where the other branch was merged into, or follow the second parent, that is, move to the branch that was merged. Based on this model, we can state this principle. A commit, A, is said to be reachable from another commit, M, if there exists a contiguous part of commits that lead from M to A. Or more simply, if we can start from M and follow the trail of parents until we arrive at A. In other words, commit A is reachable from commit M if A is an ancestor of M. Keeping this principle in mind, common version control questions like, which commits are in this branch, but not in that other one become which commits are reachable from this branch said, but not from the other ones. The way we can answer that question is by using the so‑called dot notation. For example, given two branches, we can say, and that will give us the list of commits that are reachable from the commit referenced by main, but not from feature. If we switch places between the two references, we get the commits that are reachable from feature, but not from main. This special syntax is called the two‑dot notation or sometimes called dot‑dot notation. But if there is a two‑dot notation, would you expect there to be a three‑dot notation? Well, if you thought yes, you'd be right. The three‑dot rotation, or dot‑dot‑dot, is unique to git log and it results in the commits that are reachable from either of the branches, but not from both. Of course, this command doesn't only work with branch references, you can use it with any commit reference, whether it be a relative one, a commit ID, or a tag. But what if we want to find the range of commits that exist between two references, like in the case of git diff? In order to do that, we need to pass the ‑‑ancestry path option to git log. This gives us the commits that are not only the ancestors of main, but also the descendants of feature. In this case, it corresponds to the Merge commit, M. Another very convenient way of visualizing the way our branches are structured, also known as the branch topology, is by using the log command with the ‑‑graph option to put them in a graphical representation. This visualization is fine, but, as soon as we start having more than two branches, it can quickly become harder to interpret. Fortunately, we have an alternative, meet the show‑branch command. When using the show‑branch command, we specify the branch references we want to include visualization, what we get is a hierarchical view of the ancestry lines that link our branches. Let's take a closer look at this output. The first section shows the commits that are the branch's heads. The branch currently in the working directory is marked with an asterisk, while the others are marked by an exclamation point. The second section shows the ancestry of commits for each of the branches. As you can see, they are intended to match the position of the branch name in the first section. Regular commits are marked with a plus sign, while Merge commits are marked with a minus. The first common ancestor of all three branches is shown at the end of the list, marked with both an asterisk and a plus sign for each branch. I use a show branch extensively in my daily work since I think it offers a more readable overview of the branch topology than the one offered by the plain log command.

### When Was This Merged?

When working with branches, one of the things we want to check on a regular basis is which branches have emerged and which haven't. The quickest way to answer that question is by using the branch command together with the ‑‑merged option. For example, git branch ‑‑merged shows the list of branches that have been merged into the one referenced by head. In order to answer the opposite question, that is, which branches have not been merged, we run the same command with the ‑‑no‑merged option. This approach works fine, but it has a problem, it can only tell us which branches contain commit that aren't reachable from head, it doesn't tell us what those commits are. In order to find the missing commits, we could use the git log command together with the dot‑dot notation, like we've seen earlier in this model. Another way of finding out which commits have been merged into a particular branch is by running the show‑branch command, the same one we've seen earlier in this model, only this time, passing the ‑‑topic option. The ‑‑topic option filters out the commits that have been merged into the first branch reference, thus only showing the ones that haven't. In this case, since main is the first branch in the list of arguments, we're seeing the commit from bugfix that hasn't been merged. Now, suppose we want to know when the feature branch was merged into main, that is, we want to get a list of Merge commits that have happened between feature and main. To answer a question, we could say git log ‑‑merges feature..main. In this case, since there has only been one merge between feature and main, we see only a single commit. If we're only interested in seeing the latest merge, there is the merge commit that is a descendant of feature, in addition to being an ancestor of main, we can add the ‑‑ancestry‑path option. This last option has one particular use case that's surprisingly useful. Say you want to know which merge introduced a specific commit into the main branch. Another way of phrasing that question in terms of reachability would be, which Merge commit is both a descendant of the commit we are interested in and an ancestor of main? Perhaps this is best explained with a concrete example. Let's see which committees are reachable from the latest committee in main, that is the merge committee M, but not from the first parent of main, that is commit C. Now let's assume that we want to know when commit D got merged into main, and all we have is its commit ID. What we're looking for is which Merge commit made it reachable for main. To answer a question, we can say we used the dot‑dot notation here to get a list of commits reachable from main, but not from our commit. The ‑‑ancestry‑path option makes sure to only include the commits between those two references, while ‑‑merges only shows the Merge commits. Now, there are likely going to be many merges between our commit and main, but we're only interested in the first one, so we use the ‑‑reverse option to get the list in reverse topological order. Lastly, we use head ‑1 to only return the first commit. Now let's put all this into an alias that we can call fm for find merge. All we did here was replacing the commit ID with the first argument passed to the alias, so now we can say, pretty handy if you ask me.

### Commits vs. Patches

Up until now, we have only tracked entire commits. The commands we've been using only look at the commit ID to determine whether a commit is reachable or not, they don't take into consideration the changes that a committee introduces. This can sometimes be a problem because we don't always merge entire branches, instead, we pick and choose single commits to apply on top of a branch, for example, by using the cherry‑pick command. In these situations, we want to look for commits that are patch equivalent, meaning they introduce the same set of changes as another commit, regardless of their commit IDs. In Git's notation, patch equivalent commits are marked as prime. So, for example, F prime is the patch equivalent commit of F, but has a different commit ID. To find out which commits are patch equivalent between branches, we can once again use the git lg command with the ‑‑cherry‑mark option. Note that we're using lg here, which is an alias for log ‑‑oneline to keep the command short. We talked more in depth about aliases in model three. As the name implies, the cherry‑mark option marks the commit reachable from one branch, whose patch is present in a commit that's reachable from another one and vice versa. The commits that are missing on the right side are marked by a less‑than sign, the ones missing on the left side are marked with a greater‑than sign, and finally, the ones that are equal are marked with an equal sign. If we only wanted to consider one side of the comparison, we would specify the ‑‑left‑only or ‑‑right‑only option respectively, depending on which brands we're interested in. Notice how we've used the three‑dot notation to specify the range of commits to include in the comparison, that's because we want all commits that are either reachable from one branch or the other, but not from both. Let's look at another example, this time, instead of marking the commits that are equivalent in both branches, we exclude them entirely using the cherry‑pick option. This is especially useful when we are only interested in knowing which commits are in one branch and not the other. Looking at only one side of the comparison is, in fact, so useful, that there is a shortened version for it. Instead of saying, (Working) we can simply say git lg ‑‑cherry. If we wanted to know which commits are missing in main, we would simply have to switch the order of the references.

### When Was This Changed?

Up until now, we've been tracking commits across different branches, the next step is to track changes across commits. Let's start out with an easy question. We want to know all the commits that contain modifications to a particular file. Once again, we can find this information by using the lg command, this time passing the ‑‑follow option. What we're seeing here are the commits whose snapshots contain a calculator.c file that's different from the one containing their parents. If we also wanted to know what changes were made in its commit, we would add the ‑‑patch option. (Working) Now, let's go a bit more granular and find all the commits that either add or remove the string calculator. In order to do that, we use the ‑S option of git log, followed by the string that we want to look for. We could also specify a regular expression instead of just a simple string by adding the ‑‑pickaxe‑regex option. One thing to remember is that the ‑S option only looks for lines that are either added or removed, but not both, this means that it doesn't show commits that contain modified lines. If we wanted to include those as well, we'd have to use the ‑G option instead. As you can see, this time, we got an extra commit in the results. If we take a look at its patch, we can confirm that commit C does, in fact, both add and remove a matching line. Another difference between ‑S and ‑G is that the argument of ‑G is always interpreted as a regular expression, so we don't have to pass any extra options. Now, let's do something a little more interesting. Say that we wanted to know in which commit the subtract function has been modified. Of course, we could look for a string like function subtract using the git log ‑G option, but that wouldn't be very precise. Git has long had the ability to parse the contents of text files in order to produce language‑aware diffs, providing, for example, language‑specific information in the hunk headers. Well, as of version 1.8.4, Git also learned to use that same functionality to do language‑aware searches. That functionality is exposed to the ‑L option, so, git log ‑L:subtract tells Git to look for commits that modify a c function called subtract. But how does Git know which parcel to use for a given file? Can it work with any language? As it turns out, Git is able to recognize C and C++ out of the box, but there are other built‑in language parcels available, all we have to do is tell Git which parser to use for certain files by adding an entry in the gitattributes file in our repository. For example, let's say we want to find all commits that modify the Usage section in the README.md file. If we just try to search for it using the ‑‑L option, we got no matches. However, watch what happens if we tell Git to use its built‑in markdown‑language parser for md files by adding the following entry to gitattributes. Now, let's try to search again. As you can see, we now get all the commits that modify that section in the documentation. How cool is that? You can find a list of all the available built‑in language parsers in the documentation page for gitattributes.

### Who Changed It?

Sometimes, knowing which commits modify a certain file isn't enough, we need to know who made those changes in order to gather more information about their context. Tracking authors across commit is done through the options of the git lg command. For example, say that we want to know the commits made by a certain author, in that case, we would say git lg ‑‑author equals and the name of the author we're looking for. Note that in Git, there is a difference between who authored a patch and who committed it to a repository. Let me show you. Let's look at the metadata of the latest commit in the current branch. As you can see, this particular commit was authored and committed at two different points in time, but there might as well have been two different names there. So, just as we can filter the commits by author, we can also filter them by committer. Now, in most projects, the author and the committer of any given commit are going to be one and the same. However, during the development of large projects, such as the Linux kernel, it's common to have contributors submit a patch, which is then reviewed by someone on the maintenance team, they then get to decide whether the patch should be committed or not. By recording who authored the change and who actually committed it to the repository into separate fields, makes it possible to give both credit as well a responsibility for the work done. If we also wanted to limit the list of commits to a certain timeframe, we could say git lg ‑‑author ‑‑since= "1 week". This shows the commit made by that author within the last seven days. Unfortunately, the list of supported date formats isn't documented, however, Linus Torvalds, was who originally wrote the date parser, called the implementation approxidate in the source code. If you want to know more, you can find it in the date.c file in Git's own repository. When tracking authors, we are not just limited to entire commits, we can also establish authorship for each individual line of code in a given file, thanks to the unfairly‑named blame command. The output shows the short version of the SHA1 hash that belongs to the latest commit that modified each line of code together with the author, as well as the timestamp. If we wanted to, we could also limit the range of commits by using the ‑‑notation. In this case, we're only seeing the commits made in the feature branch. Not only that, but we can also filter specific portions of a file, this can be done both by line ranges, as well as by function. (Working) Finally, something I like to do from time to time when I'm working in a long‑lived code base is to gather a bit of statistics about a number of commits made by different authors, the command that does that is called shortlog. By adding the ‑s option, we sum the total number of commits per author, while ‑n sorts the results by that number. As a fun example, here are the top 10 authors in the official Git repository on GitHub.

### Summary

In this module, we looked at how we can query the history of our repository in order to answer any questions we might have about its branches, commits, changes, and authors. We started off by defining the concept of reachability as it applies to commits and branches. From there, we learned how to track commits across branches using the dot notation and the show‑branch command. Finally, we saw how we can use the search‑related options of git log to track changes both across commits, as well as authors. In the next model, we're going to take a step back and see how fundamentally different Git's approach to branching is compared to traditional version control systems and which unique opportunities that approach offers to improve our daily workflow.

## Merging the Right Way

### The Power of Git's Branching Model

In this module, we'll look at how to take advantage of Git's unique branching model to improve our daily workflow, both for our own personal productivity, as well as for the entire team. First, we're going to see how Git's approach to branching sets it apart from traditional version control systems and what possibilities that approach offers. Then, we'll look at the different kinds of branches we can work with and how to choose the right merge strategy for each of them. Finally, we're going to go through a few unique, and frankly, rather impressive tools that Git puts at our disposal to resolve merge conflicts. Let's get started. Branching and merging in Git is radically different from how it worked in traditional version control systems. Tools like CDS and Subversion taught us that branching is slow and takes up a lot of disk space, the natural conclusion then is that we should avoid branching at all costs until it becomes absolutely necessary. Well, Git turns that rule entirely on its head by making branching a cheap and fast operation, while also being smart about intended merges. Let's take a deeper look at Git's branching model. In traditional version control systems, creating a branch means creating an exact copy of the entire working tree to a new directory, down to every single file. In some systems, the name of the directory becomes the branch name, while others record the directory's path and associate it with a name somewhere in their database. In Git, creating a branch literally means writing a value to a text file, that value is the SHA1 of the commit that represents the tip of the branch, also known as the branch head. The name of the file itself becomes the branch name and that's about it. No files are copied and no databases are updated. In Git, a branch is nothing more than a commit ID stored in a 41‑bytes text file. Let me prove that to you. Let's see what branches we have in our repository by saying git branch ‑‑all. In this particular repository, we have three branches. Since branches are really references to commits, another way of asking for a list of the local branches is by using the show‑ref command. What we're seeing here are the paths of the files that represent the branch references together with the SHA1 hashes of their latest commits. Let's look at the content of one of those files. And there you go, at this point, it should come as no surprise that deleting a branch is simply a matter of deleting its corresponding reference file. However, that's usually not a good idea since the branch references might have been mentioned in the repository configuration file. Instead, we should use Git's commands to make sure that no references are left broken, one such command is git branch with the ‑‑d option. Notice that Git warns us that we might lose those commits if we choose to delete that branch. This is a perfect example of why using the high‑level commands is better than manipulating Git's internal file system. In this particular case, we don't care about those unmerged commits, so we tell Git that it's okay to delete a branch by passing the ‑D option instead. ‑D forces Git to delete the branch, even if it contains unmerged commits. Note that the only thing we're deleting here is the branch reference. The commit that was once referenced is still in the repository, but is left dangling. That is, it's no longer directly reachable through a reference. We can get a list of dangling commits in our repository by using the filesystem check command, or git fsck, with the ‑‑dangling option. But since the tip of the branch is dangling, that means that all its ancestors also become unreachable. You can prove that by passing the ‑‑unreachable option to fsck. In this case, both the F and G commits are listed. The reason why F didn't show up before is because not all unreachable commits are dangling, some of them actually still retain a reference. They're referenced by their descendants, so technically, only the last commit in an unreachable sequence is dangling since it's not referenced by anything. Unreachable commits can be recovered under a certain period of time from the branch's own journal called the Reflog. We're going to talk about the different ways to recover commits later in this model. For now, suffice to say that the easiest way to recover our deleted branch is to simply create a new branch reference that points to the same commit as the old one. Now it's time to merge our feature branch into main. In order to merge a branch into the one currently referenced by head, we use the merge command. If any of the commits in feature happen to modify the same line in a file as one of the commits in the main branch, the entire merge operation will stop3. due to a merge conflict. We're going to look at how Git helps us resolve merge conflicts later in this model.

### Different Kinds of Branches

Making branching essentially free is one of Git's greatest achievements, that feature alone opens up a whole range of workflows that simply are impossible or very impractical to do in other version control systems. To be able to understand any of these workflows, we need to take a step back and go through some fundamental characteristics of branches. Regardless of which version control system you use, every time you create a branch, it's going to be one of two kinds, a long‑running branch or a topic branch. A long‑running branch is a broadly‑scoped branch that exists for a long period of time, anywhere between a few weeks and the entire lifespan of the project, and they're usually shared among a group of people or within the entire team. For example, it could be the branch where the next major version of the software is being worked on or the branch that contains bug fixes for an already‑released version. Now, not all projects need multiple long‑running branches. However, every project must have at least one, the main branch where all other branches are derived from and merge back into. A topic branch is a short‑lived disposable branch that focuses on a very specific task, hence the name topic, and is typically narrower in scope than a long‑running branch. They are created off long‑running branches to accomplish one goal, like, for example, implement a new feature or fix a bug. If they manage to produce some useful results, they might be merged back into the long‑running branch. In any case, once they are no longer needed, they are gone. While long‑running branches are typically shared, topic branches can be either shared or individual, in which case, they only exist in a repository where they were created. For example, we might have a long‑running branch, named vNext, that contains work towards the next version of our software. Then we might have a topic branch for a feature we're working on like say, login. Now, we mentioned that a topic branch can be either shared or individual. This property is particularly interesting because while the distinction between long‑running branches and topic branches applies to any version control system, when it comes to distributed version control systems, there is another distinction to make. Branches can be either public or private. Public branches exist in multiple copies of the repository, while private ones exist only in the repository where they were created. Git doesn't impose any rules nor limitations on what branches are allowed to be public or private. However, generally speaking, long‑running branches tend to be public, while topic branches can be either public or private. Consider the scenario where we have a public topic branch for the login feature, if we wanted to try out an alternative way of implementing the user authentication, we could create a private topic branch named idea‑for‑login in our local repository. If we are satisfied with what we have achieved, we might decide to merge that private branch into the public login branch and share it with the rest of the team. Private branches have a major advantage compared to public ones, they let us rewrite our history as much as we want before publishing them so no one will ever have to know or care about our private branches since they only exist in our local repository. So, how does all this apply to workflows? Well, cheap, lightweight branches means we can use them any way we like. It doesn't dictate how we should use our branches, it's completely up to us. However, when evaluating a workflow, it's useful to keep in mind this guiding principle. Regardless of its purpose, a branch can be either long‑running or topic depending on its lifetime and scope, and public or private, depending on who has access to it.

### Different Ways of Merging

A branch is created in the same way, regardless of how it's going to be used. It doesn't matter if it's going to be public or private, long‑running or topic, what does change however is the way it gets merged. Let's look at different ways we can merge a branch. In Git, there are two kinds of merge operations, a fast‑forward merge and a true merge. Let's talk about fast‑forward merges first. Consider this scenario, there are two commits, D and E, that are reachable from the feature branch but not from main. The important thing to notice here is that the main branch points to a commit that's an ancestor of the commit referenced by feature. If we were to merge feature into main by saying git merge feature, Git will notice that main is already reachable from feature and simply move the reference forward so that it points to the same commit as feature. This is what's called a fast‑forward merge and doesn't create a Merge commit. Now, let's look at a different scenario. In this case, the line of history of main has diverged from the one of feature with the creation of commit C. This time, if we were to merge feature into the main branch, git wouldn't be able to do a fast‑forward merge because C isn't reachable from feature and would be lost in the process. Instead, Git does a so‑called true merge. That is, it applies the changes contained in the snapshots of D and E on top of C, followed by the creation of a Merge commit M to tie the two lines of history. But what if we wanted to do a fast‑forward merge with a diverse history like in this example? Well, in that case, we could move the commits in the feature branch so that they become descendants of the latest commit in main. In other words, we would change the parent of D from B to C. The command to do that is git rebase. As the name implies, rebase allows us to change the base of our commits to another commit than they did originally. In this case, we want to rebase feature on top of the main branch so we say git rebase main feature. Git will now identify the first common ancestor of feature and main, also known as the merge base, to determine which commits are reachable from one but not the other. Then, it will reapply each of those commits one at a time on top of the commit referenced by main. At this point, we can merge feature with a fast‑forward merge. Note that rebasing is merging, the difference is that instead of merging two snapshots at once with their entire set of changes, each commit is merged one at a time. Any merged conflicts that will appear when doing a true merge would also be there during a rebase. However, instead of facing them all at once, we can deal with them as the conflicting commit gets applied. Now, let's think about the opposite scenario. What if we wanted to create a true merge, even if the two branches are on the same line of history? In that case, we can specify the ‑‑no‑ff option to get merge, which stands for no fast forward. So, how do we know if we should use one kind of merge or the other? Well, consider this, with fast‑forward merges, every commit appears to be in the same line of history, even if, in reality, it may have been done in any number of branches. Through mergers, on the other hand, reflect the way history is diverged and reconnected, even if the branch references that were involved no longer exist. So the advantage of using fast‑forward merges is that history is linear and therefore becomes very easy to read. The disadvantage is that if a merge introduce a problem, we can't easily revert it because we can't tell which line of history the merge to commit came from. With true merges, this becomes trivial since we can see the different branches of history and reverse the faulty merge commit. The downside is that a history littered with merge commit is harder to follow since there are multiple lines of history that intersect with each other. So, in which situation are fast‑forward merges more suitable than through mergers and vice versa? Well, the answer lies in one important side effect of rebasing, changing a commit parent causes the ID of the commit itself to change along with all its descendants, so rebasing is rewriting history. We should never rewrite commits that have been published since they may be fetched by someone else. So, here is a general rule of thumb. If you're merging a public branch into another public branch, like, for example, between two long‑running branches, then use true merges, this will make it obvious where the Merge commits came from. If instead, you are merging a private branch into a public one, like, for example, merging your own topic branch into a long‑running one, then preferred fast‑forward merges. Other people likely don't care about the fact that those commits were done in a topic branch that only existed in our local repository, so we should rebase our work on top of the public branch before merging it with a fast forward merge. Of course, the same thing goes for merging private branches. But what about pull requests or merge request branches? Are they public or private? The short answer is, as always, it depends. A pull/merge quest branch starts out as a private topic branch that only exists in the local repository. Then, when it's time to submit it for review, it gets pushed to a shared repository. At that point, does it turn into a public branch? Well, it does if other people start pushing commits to it, only then it becomes public. If the original author remains the only one allowed to push commit to the branch, it remains private while still being accessible for others to view. So, which merge strategy is appropriate for a pull/merge quest? Well, given that such a thing is, by definition, a public request and is often associated with documentation, call reviews, and discussion threads, I'd say a true merge is most appropriate here. You want to make it clear when, in the history of the repository, that particular contribution was merged into the main branch so you can easily associate it with the documentation records that surround it.

### Resolving Conflicts

Every time we merge a commit, like, for example, when doing a merge, a rebase, or a cherry‑pick, Git is going to combine the changes contained in the snapshot of the commit that's being merged with the one of the commit where it's been merged to. If a line in a given file has been added, deleted, or modified only on one side of the merge, then Git will include that line in the resulting file as is, no questions asked. However, if both files happen to modify the same line, Git can't decide which one should be included, so it calls a merge conflict and stops, waiting for us to settle the dispute. In this situation, some version control systems in a noble, but often vain attempt to be helpful, try to automatically resolve the conflict on our behalf based on some smart algorithms. Git does none of that. Instead, it helps us resolve the conflict by granting us a set of tools that make the job easier. Let's look at a few of them. First, let's get ourselves a merge conflict. Git tells us that it can't apply the patch because of a conflict in the calculator.c file and stops in the middle of the merge. Checking the status confirms that calculator.c was modified by two commits and is still only in the working directory. Now, if we wanted to, we could get out of this situation right now by simply passing the ‑‑abort option to git merge and HEAD would be back at the commit it was pointing to before we started the merge. Let's go ahead and open up the conflicting file in an editor. You may notice that Git uses the same notation as the merge program included in the Revision Control System, or RCS, suite of tools to highlight the lines involved in the conflict. The section above the equal signs contains the line as it appears in the commit that's been merged to, also called Ours and reference by HEAD. The section below it contains the line from the commit that's been merged, also called Theirs and referenced by a special ref called MERGE\_HEAD. Now, the key to successfully resolve a conflict lies in understanding the context in which the two changes were made. In this case, it's not obvious which line we should choose just by looking at them, we need more information about the context, and that's exactly where Git shines. Let's quit our editor for now. While searching for more clues, let's find out which commit contains the conflicting files by passing the ‑‑merge option to git log. Assuming the commits are well documented, we might, at this point, have enough information to make a decision based on the contents of the commit messages. If we still can't decide how to resolve the conflict, we might be able to gather some more insights by looking at the file itself as it was before the conflicting commits were made. In the Git parents, this commit is called the merge base, and it's the first common ancestor of two or more commits. We can ask Git to find a merge base between two commits by using the merge‑base command. In this case, we want to get the merge‑base between the commit that's been merged to and the commit that's been merged. If there are multiple ancestors, Git is going to choose the best one. There is the one that's closest to the specified commits in the line of history, like, for example, in this case, commits A and B are both common ancestors of C and E, but Git reports B as the merge base because it's the closest one. Once we have the merge‑base, we can do a so‑called three‑way merge. As opposed to the more traditional two‑way merge, a three‑way merge compares the conflicting files not only against each other, but also against their common ancestor. We can ask Git to include all three versions of the conflicting lines in the calculator.c file in our working directory by using the ‑‑conflict option of git checkout. At this point, if we open up our calculator.c file, we see that Git has added the version of the line from the merge‑base after the pipemarks. If we would like Git to always do that in the case of a conflict, we can tell it to do so by setting the conflictStyle option to diff3. At this point, we can probably tell that in order to resolve the conflict, we need to combine both changes, with the final line including both the words simple, as well as the new‑line character, so let's go ahead and do that before finally getting rid of the merge markers. (Working) If we now do a git diff after we resolve the conflict, we get a rather interesting output. Git shows a so‑called combined diff containing both original versions of the conflicting lines, as well as the merged one with different adaptations. Sometimes, the right way to resolve a conflict is to simply choose one version of the file entirely. If that's the case, we can do so by passing the ‑‑ours or ‑‑theirs options respectively to get checkout. Once we are done, we can add the immerse file to the index and create the merge commit. Regardless of how you decide to resolve a conflict, the important thing to remember is to never introduce changes in the Merge commit that aren't part of either side of the merge, this will create a so‑called Evil merge, which, as the name implies, can make it hard to track down the origin of the change since it only exists in the Merge commit.

### Ours vs. Theirs

In a merged conflict, each side is represented as either Ours or Theirs. While in most version control systems, it's pretty easy to tell which is which, there is a particular case in Git where things aren't the way you would expect. Let's talk about it. When a conflict occurs in a file during a regular merge, Ours refers to the version of the file in the branch that's being merged to. Theirs is the version of that same file as it appears in the branch that's being merged. This makes sense, since in Git, you merge someone else's branch, Theirs, to the one you currently have checked out, Ours. So far, so good. Now, let's look at what happens during a rebase. Remember, a rebase is a form of merge, where each new commit from the branch you're rebasing is applied one at a time on to the branch you're rebasing on top of. If one of the commits happen to contain a conflicting change, the operation stops to give you a chance to resolve the conflict. In this situation, Ours refers to the branch you're rebasing on top of, while Theirs is the branch you have checked out and are currently rebasing. Make sense when you look at it this way, but consider the typical scenario where you rebase your own topic branch in this example feature on top of a long‑running branch like main. This means that in the case of a conflict, Ours is the long‑running branch containing other people's changes, while Theirs is your own topic branch. Conceptually, this is exactly the opposite of a regular merge, where Ours is the branch you've checked out and Theirs is someone else's branch you're merging. Knowing how a rebase works in practice, makes it easier to resolve merge conflicts that may appear during a rebase operation since you now understand why the Ours and Their sides are reversed compared to a regular merge.

### Reusing Recorded Resolutions

So far, we've talked about how Git can help us resolve a single emerged conflict, but Git can do more than that. It can even help us resolve the same conflict multiple times, thanks to a fairly unknown feature called rerere, or reuse, recorded, resolution. Interesting for sure, but why would we ever need to do that you might ask? Well, consider this workflow. Let's say that we have a private topic branch named feature that we've been working on for a while. In order to make sure that our changes are still compatible with what's in the long‑running branch, we do a test merge from feature into main. But first, let's activate git rerere in our configuration file. We then proceed by emerging feature, which, unsurprisingly, leads us right to the same conflict we saw before, this time, however, git rerere is aware of the fact that we are in the middle of a conflict resolution and is ready to record which lines are involved, as well as the line that will end up in the merged file. At this point, we go ahead and resolve the conflict. (Working) As you can see, rerere has recorded the resolution in its own cache. We can now create the merge commit and continue working on our topic brands. At some point, we might feel ready to merge our work before sharing it with the rest of the team. However, we don't want to clutter the public history of the repository with our test merge commit. Instead, we want to remove it and replace the topic feature brands on top of the latest commit in main, this will allow us to do a fast‑forward merge and maintain a nice linear history, as we've discussed earlier in this model, so let's go ahead and do that. (Working) Remember that rebasing is the same as merging one commit at a time, this means that the conflict we resolved previously in the merge commit is going to appear again once we reach that same commit during the rebase, and sure enough, that's exactly what happens. However, this time, we don't have to do anything, git rerere has recognized the lines in each side of the conflict and reused our previous resolution. Notice that it didn't add the merge file to the index so we still have a chance to inspect it to make sure that everything looks okay. At this point, we can state the merged file and continue with the rebase. Git rerere is a real time saver when we have to merge two branches multiple times since we only have to resolve any given conflict exactly once.

### Summary

In this module, we looked at how to take advantage of Git's branches to work with multiple lines of history. We started out by looking at Git's branching model and how it differs from the one of traditional version control systems. Then, we identified the different kinds of branches we can create, public or private, long‑running or topic, and in what scenarios they're useful. Following up on that, we looked at the different ways we can merge branches and when it makes sense to prefer one way over the other. Finally, we saw some of the great tools that Git puts at our disposal to help us resolve merge conflicts. In the next module, we're going to discover Git's forgiving nature by learning how we can rewrite history to correct a mistake or to undo a previous decision.

## Fixing Mistakes

### Amending Commits

In this module, we'll discover Git's forgiving nature by learning how to rewrite the history of our repository in order to correct a mistake or to backtrack on a previous decision. We'll start out by looking at how we can amend commits, both recent and old, in order to modify their contents or metadata. In the process of rewriting history, we sometimes end up in a situation where we'd like to undo what we just did and start over, so our next topic is going to be how to revert the state of our history by implementing our own undo command. Next, we'll see how we can recover commits that we thought we'd lost by taking advantage of the ref log. Finally, we'll see how we can even debug our code base using Git. Let's get started. Look, we all make mistakes. Sometimes, we wouldn't even call it a mistake, we simply change our minds about a choice we made earlier. This happens all the time, especially when programming. Unfortunately, for us, though, version control systems have traditionally been rather unforgiving when it comes to changing history. Once something was committed to source control, it became final. Git is different, more human in a sense. In fact, Git is okay with us changing our history as much as we want, as long as we haven't shared it with anyone else. Let me give you a quick demonstration. Imagine that we just made a commit when we suddenly realize that we forgot to include one file in it. If we were using a tool like Subversion, we would have to make a new commit with only the missing file and some apologetic comment. In Git, we can simply modify our previous commit. How? Just like we are used to. First, we add it to the index and then we use git commit. However, this time, we pass the ‑‑amend option, this causes the contents of the index to become part of the same commit reference by HEAD. When we do that, our default editor opens up with a message from that commit, allowing us to further clarify it or just leave it as it is. If we know beforehand that we're going to want to reuse the message from the latest commit, we can speed things up a bit by specifying the ‑C option, which tells Git to reuse the message from the commit with the specified reference, so, ‑C HEAD reuses the message from the commit referenced by HEAD. Having the ability to rewrite history is incredibly liberating and arguably one of the greatest advantages of using Git, it allows us to make the tool fit our preferred way of working and not the other way around. For example, we can use commits as temporary snapshots of the state of our working directory while we work on a feature. If we make a misstep, we can quickly go back to a good state by removing the faulty commits. For example, let's say that the idea we developed in commits D and E turn out to be a dead end, that's not a problem since you're working in our private branch, we can simply get rid of them by using the reset command. Reset HEAD~2 moves HEAD to its second ancestor, that is commit C. Notice that we added the ‑‑hard option, this tells Git not only to move the HEAD reference, but also to update index, as well as a working copy, to match the snapshot of C. This means that any uncommitted changes we might have had in our working directory would be lost. Let me open a brief parenthesis here. You might have heard that in Git nothing is ever lost, and that certainly is true to an extent. The truth is that anything can be recovered, as long as it has been committed. This makes git reset ‑‑hard one of the few destructive commands that exist in git, since it will literally reset the working directory to match the snapshot of a certain commit without warning, so be careful when you use it or you might end up losing part of your work. Now, back to our discussion. At this point, we might feel satisfied with what we have achieved in our feature branch, and we are ready to share it with the rest of the team. However, before we do, we need to go through our temporary commits to ensure that they are consistent and well documented. For example, we might decide that the changes in commit B and C are strongly related to each other and should be squashed into the same commit. Once again, we can do that quickly by using the reset command, this time with the ‑‑soft option. The difference between ‑hard and ‑‑soft is that the latter moves HEAD to commit B, but leaves the index and the working directory untouched. This means that the contents of the index still match the ones that were part of commit C. At this point, we could include them in commit B by simply amending it like we did before, or we could change our mind again and decide that commit B should really be split into two different commit after all. In order to do that, we need to move ahead to its parent, that is, A, and have the changes contained in commit B exist only in the working directory. As it turns out, the reset command can do that, too, we just need to invoke it using its third mode of operation called mixed, which is also the default one. The ‑‑mixed option moves ahead to the specified commit and updates the index to match its snapshot, but doesn't update the working copy, which is left with the changes once contained in commit B. Now we can simply state and commit the files separately. (Working) So, to summarize, git reset has three modes, the control that's going to be changed, ‑‑hard moves HEAD and updates both the index and the working directory to mask the destination commit, ‑‑mixed, moves HEAD and updates just the index, and finally, ‑‑soft simply moves HEAD. Invoking reset without any argument implies mixed. Now, what if the commit we want to amend isn't the latest one in the branch, like, let's say when we realize that it's commit A that should have had an extra file? Well, we could use git reset to move HEAD all the way back to A and amend a commit, but then we would have to recreate all its descendants, in this case, B and C. So, instead of doing that, it's way easier to simply create a new commit that adds the missing file and then squash it into A through an interactive rebase. We can make this even quicker through a relatively unknown feature called auto‑squashing, let me show you how it works. First, let's change the message of our current commit by prefixing it with the string, fixup!, followed by the message of the commit we want it to be squashed into. Then, we start our interactive rebase with the addition of the ‑‑autosquash option. ‑‑Autosquash tells git to look for a commit whose message begins with whatever comes after a string fix up!, and automatically moves that commit under it, changing the action to fixup!. At this point, all we have to do is save the file and exit, letting rebase do its thing. You can imagine what a time saver this is, especially when you have a few of these fixup! commits spread throughout our branch an interactive rebase is all it takes to squash them into the right place. If we'd like it to always do that for every interactive rebase, we can enable autosquashing in the configuration file through the rebase.autoSquash option. At which point, we no longer have to explicitly add the ‑‑autosquash option when doing an interactive rebase.

### Git Undo

Sometimes, when we rewrite our commits, we end up with a history that looks nothing like the way we wanted. When that happens, we wish there was a way to undo our actions, just like we can do in any editing software. If you've ever found yourself in one of those situations, you'll be surprised to find out that there is, indeed, a way to undo our actions in Git. Let me show you how. Every time a branch reference moves, that is its points to a different commit than it did before, Git records its previous position in a sort‑of journal called the Reflog. In every repository, there is a Reflog for each branch, as well as one for the HEAD reference. We can get a list of entries in the Reflog of a given branch by using the reflog command, followed by the name of the branch we're interested in. If we, instead, wanted to look at HEAD's Reflog, we would simply omit the argument and say get reflog. The entries stored in the reflog are in reverse chronological order, with the most recent ones on top. Notice also that each entry has an index, this is very handy because you can use that index to point to the commit referenced by a certain Reflog entry using this special syntax, reference@index, where reference can either be the name of a branch or HEAD. Index is the entries position in the journal. So, for example, if we wanted to look at, say, the commit HEAD was pointing to two positions ago, we would say git show HEAD@{2}. If we instead wanted to look at the commit main was referencing just before the latest one, we would say git show main@{1}. Now, think about this, the Reflog keeps track of the history of commits referenced by a given branch, just like the history of our web browsers keeps track of the URLs we visit. This means that the commit referenced by @{1} is always the commit that was referenced before the current one. If we were to combine the Reflog with the git reset command that we saw earlier in this module, like this, for example, we would suddenly have a way to move HEAD to index and the working directory to a previous commit referenced by a branch. This is essentially the same as pressing the Back button in our web browsers. At this point, we have all the pieces we need to implement our own git undo command, which we do in the form of an alias. Here it is. Now, there are three interesting bits to note here. One, we're defining the alias as a shell function named f, which is then invoked immediately. Two, we're using the rev‑parse command, followed by the ‑‑abbrev‑ref option to get the name of the current branch, which we then concatenate with the @ syntax to form the reference to a previous position. Three, we have specified the position in the Reflog as a parameter, with the default value of one. This is the whole reason why we defined the alias as a shell function, to provide a default value for the parameter using the standard shell syntax. The beauty of using an optional parameter like this is that it allows us to undo any number of operations. However, if we don't specify anything, it's going to undo just the latest one. Let's try it out. Say that we first removed the last two commits in main, D and E, and then we merged the feature brands. Now, let's say that we want to undo our last two operations, we can do that quickly by using our undo alias. And there you go, our history is now back to the way it looked like before we started rewriting it. But, what if we wanted to undo the undo? Well, since git undo also counts as an operation, all we need to do is to once again undo our latest operation with git undo, which is the equivalent of saying git undo one.

### Recovering Lost Commits

Git, at its core, is designed to work like a file system, and as such, it cares a great deal about the integrity of our data. So when we move, edit, or delete commits in the process of rewriting history, nothing is actually lost. Even if there seems to be no way to get a hold of a commit, there is always one last reference, the Reflog, HEAD reflog to be exact, since it records all commits that have been referenced by HEAD at one point or another. Keep in mind that the entries in the Reflog won't stay around forever. Git will, in fact, remove entries older than a certain number of days as part of a garbage collection cycle, how many those days are is different based on whether the reference commit is still reachable or not. The default expiration dates are 90 days for reachable commits and 30 days for unreachable commits. We can change those values by setting the gc.reflogExpire and gc.reflogExpiredUnreachable configuration variables respectively. So, what does this mean to our day‑to‑day work? Well, it means that we can rewrite our history as much as we like without having to worry about losing our commits since they'll still be around in a repository between one and three months before they're finally deleted. But how do we recover a commit from the Reflog? Well, there are a couple of ways to do it depending on what it is we want to recover. For example, let's say that our history consisted initially of these commits, but then we accidentally removed commit C while doing an interactive rebase so that history now looks like this. Now, we know that commit C is still in HEAD's reflog since HEAD did reference it at some point so we could just go ahead and look through the Reflog entries until we find the one whose commit message is C. But there is a quicker way. Instead, we can simply search for that particular commit message in the entire Reflog by using the git lg command followed by the ‑‑grep and ‑‑walk‑relogs options, respectively. This will give us the list of reflog entries that point to a committee whose message contains that entry. At that point, we can simply cherry pick the one with the lowest position since that's the most recent one. There is a problem though, the cherry‑pick command is going to apply the commit at the tip of the branch, which isn't exactly what we want. We would like to put C back to where it was originally, between B and D. So how do we do that? Well, as always, there are a few different ways to do it, but the one that involves the least number of steps is this, first, we move HEAD to C's old parent commit that is B. We're using checkout here because we want to move HEAD by itself without modifying main, which is still pointing to commit D. Then, we cherry pick commit C on top of B by fetching it directly from the reflog. One thing to notice here is that since we moved the HEAD reference a few times, the entry for commit C has probably moved down a few positions in the Reflog since the last time we checked, so we should look for it again to make sure that we cherry pick the latest one. We then follow up by cherry picking commit D, which is still referenced by main. Finally, we move the main branch two points to the new D commit, the F option forces git to set an existing branch to a specific commit. As a final step, we move HEAD to main using checkout, and we are done. As you can see, commit C is now back to its place in our history.

### Debugging with Bisect

As incredible as it may sound, Git can even help us track down a problem in our code base. How? Let me give you an example. Let's say that we just pulled the latest commit from a share repository and decided to run the build script to verify that everything is okay. Oh‑oh, we got a problem, the code doesn't compile. In situations like this, the first thing we want to know is when did this happen, or to be more precise, which commit broke our code? If we knew exactly where the problem is, we could answer that question pretty much immediately using git blame on the file that contains the error, and that will give us the ID of the commit that modified the faulty line along with its author. However, if we can't determine where the problem is, our last resort is to manually go through a fair amount of commits manually inspecting each and every patch, tedious to say the least. Fortunately, once again, git has us covered, thanks to a command called git bisect. Bisect, as the name sort of implies, helps us dissect our code by doing a binary search through its history of commits. Here is how it works. First, we need to tell git that we want to start a bisecting session by saying git bisect start. At this point, we need to give git a range of commits to search, we do that by giving it two commits, first, the commit where things are bad, and then the commits where things were last known to be good. In this case, we know that things are pretty bad right now, so we say get bisect bad, which marks the current commit E as the bad end of the range. Now, we don't really know the last time things were good in this repository, so we're going to say that the latest known good commit is the first one, A. Git has now enough information to start bisecting. It starts by moving HEAD to the commit right in the middle of the range, in this case, C. The next step is for us to check the state of the working directory, we do that by simply running our build script. It completes successfully, so commit C is good. Let's tell Git about that. This means that a faulty commit must have happened after C. In other words, it must be a descendant of C, so Git moves head to the middle commit, this time in the upper half of the range, which brings us to commit D. Now, let's go ahead and run our build script once again, and here we have the compilation error, so commit D is bad. At this point, we can safely say that D is our faulty commit. When we're done with our bisecting session, we move head to its original position by saying, get bisect reset. Thanks to git bisect, we only had to check two commits instead of five to determine which one of them introduced the error. But I know what you're thinking, sure, this is certainly much faster than going through each and every commit manually, especially when you have a range containing dozens of commits, still, having to run our build script manually at each step is time consuming, as it turns out someone already thought of that. In fact, if you have an automated and repeatable way to verify the state of your working tree, like, for example, a build script, you can tag it to automatically run it at its step during the bisect operation. If the script exits with unknown zero code, it is going to assume that a current committee is bad, while a zero exit code means the committee is good. So, since we do have a build script, let's rerun our bisect session, this time, letting Git do all the work. Git bisect start HEAD HEAD~4 is a shortened way of start bisecting and give the search range, the bad commit first, followed by the good one. Then, we simply say get bisect run make, at which point git runs off doing its thing and gets back to us when it has found the first bad commit, which, as we know, is commit D.