## Manipulating Commits

### Introduction

Changing the history of a source code repository may seem like something you wouldn't want to do. After all, the whole point of a repo is to record how a codebase has evolved over time, right? While that is generally true, there are times when it can be helpful to manipulate a project's history to increase the clarity of how the project was developed. So in this course, we're going to learn how to rewrite Git history. Again, that's not something we're going to be doing every day, but when we do need it, we're fortunate that Git provides a set of very powerful tools to allow us to change and manipulate the history that we need to. So we're going to break this course down into two major sections. Now, we're going to start with a little bit of infrastructure work here. We're going to describe the versions of Git that this course is designed to be compatible with, and we'll do that in the next clip. Then through the rest of this module, we're going to talk about how to modify the commits in the repository itself. Specifically, we're going to break that down into how to reorder the commits in our repository, how to squash them, so taking multiple commits and condensing them into a single commit, how to split them, which is the reverse of squashing, and finally, if we ever do need to delete a commit in our repository, we're going to learn how to do that. Now, this course is assuming that you're fairly comfortable with using Git in your daily life, so we're not going to be talking about a lot of theory on its own here. We're going to be focused on using Git commands, and we'll talk through the theory as we explore the different features and different ways that we can modify those commits. Then in the next module of this course, we're going to change focus a little bit and talk about how to make changes that affect the entire source code repository. Now, Git doesn't really come with tools that allow us to do that easily, so we're going to explore a third‑party tool that really simplifies the process when you want to take a repository and make changes across the entire history of that project. Okay, so let's go ahead and jump in and review the version information that this course applies to.

### Version Information

Before we dive into this course's content, let's just make sure that we have in mind the versions of Git that this course is designed to work with. So this course was created using this version of Git, so if you want to exactly replicate the experience that you're seeing in this course, this is the version that you're looking for. However, this course is not exclusively compatible with just that version. This course is 100% compatible with all versions of Git that fit within this criteria. However, if you have a version that fits within these criteria, then compatibility is not guaranteed because of some changes that have happened in the API over time. So hopefully that lays out the groundwork and helps you understand if the version of Git that you're working with is going to work with the material in this course. Okay, so with that out of the way, let's go ahead and jump in and build a small repository and then explore how we can manipulate its history.

### Demo: Updating Commit Messages

I want to do the demonstrations in this course using Visual Studio Code. Not that we need Visual Studio Code for any of its special capabilities, but it is convenient to have the file listing on the left‑hand side while we have the terminal in the main body here. So we're going to use Visual Studio Code. You can, of course, follow along with the demonstrations in this course by strictly using the command line because that's exactly what we're going to be working with as well. So the first thing that we need to do is we actually need to build a repository. So we're going to keep things extremely simple in this course because our focus is on manipulating the history, not building any specific project. So I'm going to start by using the touch command to build several empty files, and we're just going to non‑creatively call them one, two, three, and four.txt. Then we're going to go ahead and initialize our repository using the git init command. Now that our repository is created, let's go ahead and check out the main branch, so we'll go ahead and do that, and then let's go ahead and start adding these files. Now, normally if you've got multiple changes at one time, we would do a single commit to commit that together. But we want to build a bit of history in this course so that we can manipulate it. So what I'm going to do is I'm going to take one of these files at a time, so I'm going to add one.txt to the repository. We can see that that's staged by running git status. We can see that one.txt is staged and the other three files are currently ignored. Then we can go ahead and commit that using the git commit command, and we'll just call this the initial commit and go ahead and run that. Now, if we run git status again, we see that we have our three unstaged files, but one.txt doesn't show up anymore because that's already been committed. Now, we're going to just rinse and repeat that three more times. So we're going to go ahead and add two. Then at the same time, I'm just going to combine this into a single line on the command line. We'll call this our second commit. Then we'll add three, commit that, and then finally, we'll add four, and commit that as well, just to build a little bit of history. Now, the first thing that we want to talk about is how do we view that history? How do we view the order of commits in our current branch? Well, to do that, we're going to use the git log command. Now, git status will show us the status of our current working directory, and if we run that, we see that there's nothing to commit because that's what status is reporting on. It's going to tell us the difference between the head of our repository and the current working directory that we're in. Since the head matches our current working directory, there's nothing for a status to show us. Git log, however, is going to show us the log of the most recent commits in a repository in reverse chronological order. So when we run git log, we can see that we've got the fourth commit, the third commit, the second commit, and the initial commit, all in reverse chronological order. So the first thing that we want to talk about is what happens if we've got a misspelling? What happens if we wanna change a commit message? Well, if you notice in my last commit, I wrote the message as four commit instead of fourth commit. So what happens if I want to fix that? Well, the way we're going to do that is the first thing that we're going to talk about with rewriting history because this is the history of the repository. It's just the most recent bit of history. So we're going to do that by using the git commit command again, but we're going to add the ‑amend flag. Now, the ‑amend flag allows us to alter multiple things in the most recent commitment. But what we want to focus on right now is how do we update that message? So all we need to do is provide the ‑m flag again, and then I'm going to provide the updated message. In this case, we're going to say fourth commit. So when I run this command and then I check my log again, we can see that now my most recent commit has that updated message, fourth commit. Okay, so we've got a little bit of work under our belts already. We see how to check the log of our most recent commits and how to change the message in our most recent commit. The next thing that I want to talk about is how we can reorder those commits. For example, what if the second and third commit are in the wrong order? We'll see how to reorder those in the next clip.

### Demo: Reordering Commits

The next thing that we want to talk about is how to reorder our commits. So let's imagine that our second and third commit for whatever reason, they're in the wrong order, or they're in an order that could be somewhat confusing when someone is looking at the project history. Now, I'm going to show you how to do this, but there is one thing that we want to keep in mind before we start talking about reordering commits, and that is if you're reordering commitments that affect the same file, you will need to resolve the conflicts in those files that have been modified by the multiple commits because Git doesn't know what order to apply those changes in since potentially the commits that used to be earlier in the history are going to influence the changes that came later. So that's why we're working at the file level in this course because we're not going to run into that same conflict. Now, the secret to doing this, the secret to reordering commits, and really the secret to doing everything else that we're going to be talking about in this module, is by using the git rebase command. What we're going to be doing with rebase is we're going to be reapplying commits to a base chain because that's where rebase comes from, we're reapplying commitments onto a base that's previously existing. Now, the way that we're going to do this is we're going to put Git's rebase mode into its interactive state so that we can actually see the changes that we're going to make before we commit those and before we update the repository. So we can do that by adding the ‑‑interactive flag, or we can shorten that by using ‑i. Now we also need to tell Git which range of commits we want to be working across. Now, the way that we're going to do that is we're going to tell Git where to start, we're going to be starting at the head of the current branch, and then by adding a tilde that's going to tell it how far back to go. So I'm going to add a 3. That's going to tell Git we want to work with the previous 3 commits starting at the current head. So we're going to have the HEAD, the HEAD‑1, and the HEAD‑2. So let's go ahead and run that and see what happens. We see that Git is telling us that it's going to wait until we close the editor. Now, this is another convenience of Visual Studio Code because I know some of you, like myself, aren't big fans of them, which is Git's default text editor. So I've set my version of Git to run in Visual Studio Code so that we can simply and easily make the changes, and again, you can follow along if you've configured Git in the same way. Now, we can see that Git has provided us a lot of information here. The first three lines are a summary of those three commits that we asked for. Remember, I asked it to start at the head and give me the three previous commits. So that is the fourth commit, the third commit, and the second, but notice that we're now in chronological order since the second commit, the earliest one chronologically, is listed first. Now, down below this, we can see quite a bit of information about the different commands that we can ask Git to perform during this rebasing operation. Now, we're going to explore quite a few of these in this course, but we're not going to explore all of them. As a matter of fact, for this demonstration, we're not going to use any of these. Instead, we're going to take advantage of the fact that Git is going to apply the commits in the order that they're listed. So right now, it will apply the second commit, the third commit, and then the fourth commit. If we scroll down, you see on line 29 that we have this helpful hint. These lines can be reordered. They're executed from top to bottom. So the way that we're going to reorder the commits is we're simply going to move a line here. So I'm going to take the second commit and move it below the third commit. Now, when I save this file and close the editor, remember that's what Git is waiting for. it's waiting for me to close the file, then Git is actually going to replay the second, third, and fourth commit, but it's going to do it in a different order. So let's see how that works. So we see that the rebase has been done successfully. If I check my log, you'll notice that I've got fourth commit, second commit, and third commit, and then my initial commit. So I've reordered the second and third commits, and I can actually prove that if I check out my third commit. Then notice that I only have files one.txt and three.txt. Two.txt doesn't exist anymore, and that's because we haven't committed that yet because that's committed later in the history of the project. So again, this is something that you can easily do when the commits are affecting different files. If you are having multiple commits that you're reordering affect the same file, you will have to resolve those conflicts. So let's go ahead and check out my main branch again so that we have that history back. Try that again with the proper command. Check out my main branch, and we're back to where we were. So just to review and to keep my project history clean, let's go back and rebase that again. So we'll go with ‑i for the interactive mode, we'll go ahead and grab those last three commits, and then we reordered these lines again, switching the second and third commit around, save that file, close it, and once again, if we check our log, we see that our history is back to where it was. So we've got our fourth, third, second, and initial commits in reverse chronological order. So the next thing that I want to show you is how we can combine multiple commits into a single one. In other words, how can we do what's called squashing commits. We'll see how to do that in the next demonstration.

### Demo: Squashing Commits

The next thing that we want to talk about is how to squash commits. Now, this is commonly done in preparation for combining our commits or creating a pull request to merge our local branch back into the main repository. Because as we're doing our local development, we might have several commits, and those commits might reflect a development of thought as we're developing a feature out that might be confusing or overly verbose as we look through the history of the main project. So squashing allows us to have an extensive commit history locally, but then trim some of that out before it actually gets combined into the main repo. So let's see how to do that. We're going to go ahead and start once again by using that rebase command. We're going to go into interactive mode, and then once again, going to ask for the HEAD and the three commits prior to that. So that's going to trigger our editor to open up again. It's exactly the same command to start with. It's just what we're going to do within this file that's going to change now. Now, what's going to happen when we squash commits is we're going to go down, and if you look at line 11, you see that we've got this command s, or squash, and that's followed by the commit number. Now, what that's telling us to do is it will combine or squash the commit that we specify with the previous commit. So to show you how this works, the command that it's actually talking about is right here, the first words on lines 1 through 3. So I'm going to take this first word pick, and I'm going to change that to squash. I'm also going to take this next one, and I'm going to change it to s. So we're going to see that both s and squash do exactly the same thing. Now, I'm going to save that file. Now, what do I expect to happen? What should happen is that all three of these commits are going to be combined into that second commit. So we're going to remove the third and fourth commit, and we're just going to have that second commit reflected. Let's go ahead and close this file and, and see what happens. And you see that now Git is asking us for another change. It's asking us for clarification on what that commit message is because it's recognized that it's combining three commits with three independent commit messages into one. So what's our ultimate commit message? Now, we can do whatever we want here. I'm going to keep the recommendations that it's giving us, but I do want to just scroll down a little bit just to give you a little bit of information about how this file works. If we come down here to line 14, notice that we have some hints on how to create that updated commit message. So anything that we enter into this file is going to be the new commit message, but there is some commentary provided that will automatically be ignored. So any line that starts with that hash is not going to be part of the commit message. So you don't have to worry about dirtying up your commit messages with all the text that Git is putting in here for us. As a matter of fact, the only text that's going to be maintained from this file is lines 4, lines 8, and line 12, which is our original commit messages. So let's go ahead and close this out and see what happens. So to see the effect, we do see that the rebase was successful, it created all three of those changes inside of a single commit. If we look at the log, we see that we now only have two commits. We can get that initial commit, which is where we checked in file one.txt, and now we've got the second commit that is adding in the other three files. And once again, we can prove that by checking out our initial commit, and we see that we only have the file one.txt recorded in our history. If we go back and check out that main branch again, however, all three of those files are going to show up. So this is how we can squash multiple commits into a single commit and clean up our project history. Again, that's a very common activity that's done when preparing a branch that we've worked on locally to commit it to the main repository. Okay, the next thing that we're going to talk about is how to reverse that. What happens if you combine multiple commits or recognize that really one commit should have been spaced out over several commits in order to properly reflect the change in evolution of your project? So we'll see how to address those situations by splitting a single commit into multiple commits in the next clip.

### Demo: Splitting Commits

So the next challenge we have is how do we split a commit that exists in the history of our project? Well, there's actually two different routes that we can go down here. So if we look at our log, we see that our projects is where we left it at the end of the last demonstration, where we've got two commits, that initial commit, and then that second commit that combines those three commits that we initially had. So the first way that we can update a commit is if it's the latest commit on the branch. In that case, what we can do is use the git reset command if I ask it to reset the HEAD, and I just trail that with the tilde, what Git is going to do is it's going to unstaged that change. So it's going to take that commit and it's going to kind of rewind it to just before we actually ran that commit command. So you see that if I run that and then run git status, you'll see that I've got three unstaged commits. So at this point, it's almost like that previous commit never happened. So we can then go ahead and add our three commits in. Now, I'm not actually going to do that because I want to show you another way to do this if the commit that you want to change is not at the head of the current brand. So let's go ahead and reset this. Now, the way that we're going to undo this reset, we actually need to find the commit that we want to reset to be the HEAD. So the easiest way to do that is by looking in the reference log. So if I ask for git ref log, we're going to see the recent history of our project. And notice this second line here is actually where we checked out that previous HEAD. The first line in the ref log is actually that reset action. So what we're going to do is we're going to reset again, we can just grab this reference, let me try that one more time, and now our project history should be exactly like we expected it to be. So what I want to show you is how do we split the commit if it's earlier in the history? So let me go ahead and create another file here, add that, and commit it. So now when we look in our log, we see that the commit that we want to split is now earlier. It's the second to last commit; it's not at the head of our branch. So the way that we're going to get to this one is we're actually going to run that rebase command one more time, so let's go ahead and run rebase ‑i, and I'm going to ask for HEAD, and then I'm going to go back three. It looks like I don't have three anymore, so we're going to go back just to make sure that I get all of them, and we can see that the second commit is now showing up. So what I want to do is I want to edit that second commit. We're going to select the line, and instead of saying pick, we're going to change that to edit. That's going to tell Git that we want to rewind the history to that commitment, and then we're going to redo it. Let's go ahead and save that and close it and see what happens. Now from here on out, it actually doesn't matter if we're working on the most recent commit or earlier in the history. We're now on that commit that we want to work with. So, what does that look like? Well, if we check our log, we see that we're on that commit that we want to change. Now, how do we change that? Well, the HEAD of the branch, at least temporarily, is right at the commit that we want to change. So we're going to go ahead and do that git reset again, and we're just going to tell git that we want to reset the current commit. So when I'm done with that and I check the log, notice that I've only got the initial commit to my repository. I've unstaged that commit that I want to change. So now if I check the status of my project, notice files 2, 3, and 4 are no longer staged because I don't have that commit that originally staged and added them to the repository. So we're just going to go through and we're going to add them again. So we're going to add two.txt and commit, that's our second commit. Then we're going to add three, commit that, then we're going to add four, and commit that. Now, if we check our log, we can see that we've got the commit that we want and we've split them. We've now added all three of those changes in the middle of the branch, but in a way that we want them, so we've split that one commit back into three. Now, how do we get back to the actual HEAD? How do I replay that fifth commit that I just added? Well, to do that, we need to tell Git that we want to continue the rebasing operation. What that's going to do is it's going to look for the next command in the rebase, and it's going to execute that. Now, the next command was the pick the fifth commit. So when we rebase and we tell it we want to continue, it's going to continue that rebasing operation, it's going to pick that fifth commit, and it's going to play that on top of our revised history. So that's how we split commits. The basic operation, whether it's the most recent commit that we want to split or earlier in the history, is exactly the same. The only difference is if it's earlier in the history than the HEAD, then you do have to use that rebase, go in and change the command to edit, so that Git knows that we want to do an update and change that commit. Okay, the last thing we're going to do in this module is we're going to learn how to delete a commit that's no longer required. So we'll talk about that in the final clip of this module.

### Demo: Deleting Commits

So what happens if you find out that you've got to commit in your project and for whatever reason, you simply don't need it anymore? It's no longer relevant. All of the changes that were included in it are irrelevant to your project history, and so they're just going to confuse that history by having one more commit that you're going to have to consider. Well, that's what we're going to take a look at now. So let's take a look at our log. And where we left this repository is we've got these five commits. We've got our initial commit all the way through our fifth commit. Now, the way that we're going to delete a commit from this is, remember, our rebasing operation is going to apply a set of commits to a base. So our first step is to once again use that rebase command going into interactive mode. And then this time, let's grab all four of the commits that have been placed on top of that initial commit. Now, there are two ways that we can remove a commit. The first that I would actually recommend for you to use is if we scroll down a little bit, notice here on line 19 there is this drop command that will explicitly tell Git to remove that commit. However, there is another option, and that's down here. On line 32, if you remove a line here, that commit will be lost. So I wouldn't necessarily recommend that you do that, but I want to show you that both of these will actually drop entries from your brand. Let's go ahead and drop the third commit by just deleting the line, and then the fifth commit, we're going to use that drop command, or the d. Let's go ahead and save that. So now what we would expect is we're going to have our initial commit, the second, and the fourth, and the other two commits are simply going to be gone. So let's see what happens when we close this file. And we see that it does take a second for it to run. But now in our file browser, we see that we do, in fact, only have three files. We've got one, two, and four.txt. So a couple of different ways that you can remove a commit from your project history. Once again, please be very, very careful with this because, first of all, those commits are going to be gone, it's going to be very hard to retrieve those, and second, if future commits are modifying the same files as a commit that you drop, you're going to have to resolve those conflicts that are going to arise from the fact that you've got multiple changes that are affecting the same file, and one of those change sets is gone. So you can certainly drop commits from your project history to clean things up. Just be aware of the ramifications of those. Okay, so that wraps up the commands that are built in to Git for updating the commit history for a project. In the next module, we're going to explore a third‑party tool that will actually take an entire repository and allow us to conduct operations across that entire project's history, potentially affecting every commit in the repository. We'll see how to do that in the next module.

## Making Repository-wide Changes

### Introduction

In the last module, we explored some powerful capabilities that Git provides to us to update and revise the history of our source code repository. But what happens if we need even more power than that? What happens if we need to change potentially everything that's recorded within a repository? So far, the tools that we've explored are really good at changing one or two commits at a time, but if you're talking about changing across an entire repository, we're really lacking the capability of doing that. So in this module, we're going to explore a tool that's available that allows us to potentially completely rewrite the history of an entire repository. So let's go ahead and dive in and get introduced to this powerful tool.

### Overview of Git Filter-repo

I mentioned at the beginning of the course that revising a project history is not something you're necessarily going to do every day, while rewriting the history of an entire repository is something you're going to do even less often. However, there are times and use cases where the ability to rewrite the history of an entire repository can be extremely valuable and save a lot of time and energy. So the project that we're going to focus on in this module is this project by Newren called git‑filter‑repo. Now, there's a lot of capabilities within this project. We're only going to touch on how to install the project and go through some very basic functionality. But I want to introduce this to you because if you ever do run into the need to rewrite a project history, you're going to know where to go. So if I scroll down, we're going to get a very basic introduction to the project right here in the top of the README. And as you can see, it's pretty much what it says on the tin. Git‑filter‑repo is a versatile tool for rewriting history. And that's it. Its only job is to rewrite history, but it can rewrite a lot of types of history in your Git repo. So if we come down, we can see some examples of how you might want to use this project. Now, I'm not going to go through them all in detail, but there is one use case that I can see that could be very valuable if your project and if your team runs into it, and that is what happens if sometime during your project's lifecycle, you recognize that a production secret, either a password or a key to a cloud service or something like that, got checked into the project. So now potentially you've got an extensive project history, and that key might be there several times, even replicated in multiple places. So how do you address that? How do you find all of the places within the entire project history where that secret was committed into the repository and remove those? Well, that's a perfect use case for git‑filter‑repo. So I'm not going to go through all the details of this website. I'm going to leave that up to you to explore. But I do want to take a few minutes in this module to walk you through how to get this tool installed and how to get it running. So let's go ahead and jump into how to do that in the install section, and we can see that there is an INSTALL.md file, so let's go ahead and look in there. This file, get‑filter‑repo, is very simply a Python script. Now, because of Git's extensibility model, there's some environment setup that we can do that makes things even more easily accessed than running it as a Python script. But you can see right here, we can simply download the script and then run that with Python. However, if the script is on the project's execution path, then you can actually just invoke using the git command, followed by filter‑repo. So we can see that if we jump over to Visual Studio Code. I've actually got my project set up with filter‑repo on the execution path. So if I run git‑filter‑repo, and I'm just going to use ‑analyze, that's a very simple flag that's available to us, and you can see that I do get a response. Now, this response here is yelling at me because I've run the same command twice, but you can see that filter‑repo is working. Now, this is not a built‑in command. We're using Git's extensibility model to invoke this Python script. As a matter of fact, if you want to see where this is located, I can show you that by popping over to Explorer, and you can see that the script is right here. Now, one thing to note. Notice that there is no .py extension on this file. You do need to remove that extension for Git to recognize it as a Git command. But if we do look at the contents of the script, we can see that it is, in fact, a Python script. So now that we have a little bit of an introduction to the git‑filter‑repo project, we see where the project is located and where to get documentation and how to install it, in the next clip, we're going to have a very simple use case. We're going to change the commit messages within our repository from what they say right now into some revised format. And that's going to just lay down the groundwork for how git‑filter‑repo generally works. So we'll see how to do that in the next clip.

### Demo: Using Git Filter-repo

So now that you have get‑filter‑repo installed, how do you use it? Well, conveniently in our documentation, if we scroll down to the, How do I use it section, it's going to tell us. Now, it's actually not going to tell us right here. It's going to redirect us to where we can learn. We're going to click into the user manual here, and we're going to see that we've got documentation that's written in the format very similar to Git documentation. So we see that we can run the git‑filter‑repo command, and that it's got two main tags. It's got the ‑analyze flag that allows us to analyze an entire repository and potentially gain some information that might help us either determine how to apply a filter, or to determine that a filter has actually executed correctly. Now, what operations might we want to do for a filter? Well, there are quite a few. I mentioned the use case of removing secrets that might have gotten checked into a repository. There's also the capability of distilling down. If you've got a source repository that's got multiple modules in it, you might want to distill out the history of just one set of modules, or just one module, and filter repo can do that. Now, the use case that we're going to use is very simple here. We're going to use this section here, commit\_message\_filtering\_options, which if I come down, you'll notice that the options are grouped together. So we've got filtering based on paths, we've got content editing filters. So this would be where you'd go if you want to remove a secret that crept into your repository. And then down here we see the filter commit messages. Now, the format for how filter repo works is generally going to be very similar across all of these different commands. So I'm just going to show you one, and then you can apply it as you need to. So we're going to explore this replace‑message command. So as you can see, we need to provide an expressions\_file for filter repo to act upon. So that expressions\_file is a file with expressions that if found in a commit or tag message will be replaced. So we're not actually going to put all of our replacements on the command line. We're going to create a file that contains all of those replacements. So let's go ahead and see how this works. I'm going to come into my source repository, and I'm going to go ahead and create a new file right here. We're just going to need to make sure that we don't check that in. And then I'll go ahead and edit it, and the way that this file is set up is we're going to start with the text that we're looking for on a line and then we're going to redirect it to the text that we want to replace it with. So if we look at our log, we can see that we've got our initial commit, we've got our second commit, third commit, fourth commit, and fifth commit. So let's go ahead and change commit to a different word. So we're going to look for commit, and then the way that we're going to tell filter repo that we want to replace this is we're going to add two equal signs followed by the closing bracket or the greater than sign. Then we're going to add the new text here. So we're going to save file added. So we're going to say that we have the initial file added, the second file added, the third file added, and so on. Then we're going to go ahead and save this. Now that we have that file in hand, let's go ahead and run this. So the way that we're going to do that is we're going to run git filter‑repo. Remember that we can just run this from the git command because Git is going to recognize that extension is on the path. Then the specific filtering option that we're going to do is replace message. And then we need to tell it where that file is going to be. And then we're going to go ahead and run this and see what happens. So when I run this, you're actually going to see that it didn't do it. Now, why didn't it do it? Well, filter‑repo recognizes that it's running some extremely powerful commands because it is literally going to rewrite the history of our repository. This is a destructive operation. There is no undo. So you better know what you're doing when you do it. So by default, what filter‑repo wants to do is act upon a fresh checkout of our source code repository. The idea here is that the filters aren't necessarily intended to change your main repository. They're meant to create a distilled version or projection of that main repository for very specific cases. So because it's so powerful because it's destructively updating our source code history, it does not want to change a project that has any changes that have not been committed upstream. So since I haven't committed any changes in our demonstration project upstream, it is very hesitant to proceed. Now as is normal with Git, we can override that if we know what we're doing and we're committed to this. So we can go ahead and use that ‑force flag, so let me go ahead and add that, and now we're going to see that filter‑repo is going to do exactly what we ask it to do. It is destructively, remember, that it is destructively, updating our project history, and it's going to update all of our commit messages. Now, we can show that by running git log again, and now you see instead of fifth commit, we've got fifth file added, so in this single command, we changed every commit message in the entire repository. So this is just the tip of the iceberg I've touched on. I've barely touched on the capabilities of filter‑repo, but this is the basic idea that the project provides. You're going to have a base repository, you want to apply some filtering or updating operations, so you're going to find the correct flag and provide a configuration file if necessary, and then you're going to execute that, and it's going to create a brand‑new projection of that original repository for you to work with. Okay, so I hope that that helps you understand a little bit about how Git's history works, and how to update it, either by updating the individual commits in our project, or, as we've seen in this module, by updating the entire repository.