Switching to Git in Visual Studio

Micheal Padden

Version 2.1.227-5-g51a3aa9, 2020-06-07

Table of Contents

Licence

Preface

This introduction combines sections of the first three chapters of "Pro Git" with Visual Studio screenshots, to help Visual Studio developers who are familiar with version control systems such as Source Safe and Subversion switch to Git.

Pro Git, by Scott Chacon and Ben Straub, illustrates these and many more topics in an accessible way using the git command line. The full on-line version is available at https://git-scm.com/book/en/v2.

Introduction

Here is a quick summary of what the three chapters of this book will cover.

In **Chapter 1**, we're going to cover the Git basics: what Git is, the components and basic workflow, and how to set it up in Visual Studio.

In **Chapter 2**, we will go over basic Git usage — how to use Git in the 80% of cases you'll encounter most often. After reading this chapter, you should be able to clone a repository, see what has happened in the history of the project, modify files, and share changes.

Chapter 3 is about the branching model in Git. This is the feature than enables Git to manage multiple parallel changes to a project with minimum fuss, and will revolutionise your development workflow and ability to develop collaboratively with colleagues.

Further Topics gives a brief introduction to further topics and where you can find and find out about more powerful Git tools.

Let's get started.

Getting Started

This chapter will be about getting started with Git. We will begin by explaining some background on Git, it's main components and workflow, then move on to how to set up Team Explorer to start working with Git. If you have already read "Getting Started" chapter of "Pro Git", you may want to skip ahead to First-Time Git Setup. At the end of this chapter you should understand the basic Git toolset and you should be all set up to use it.

What is Git?

So, what is Git in a nutshell?

A git repository is a project version history. Every time you commit, or save the state of your project, Git basically takes a picture of what all your files look like at that moment and stores a reference to that snapshot. For efficiency, unchanged files are stored as linked to the previous identical file. Git thinks about its data like a *stream of snapshots*.

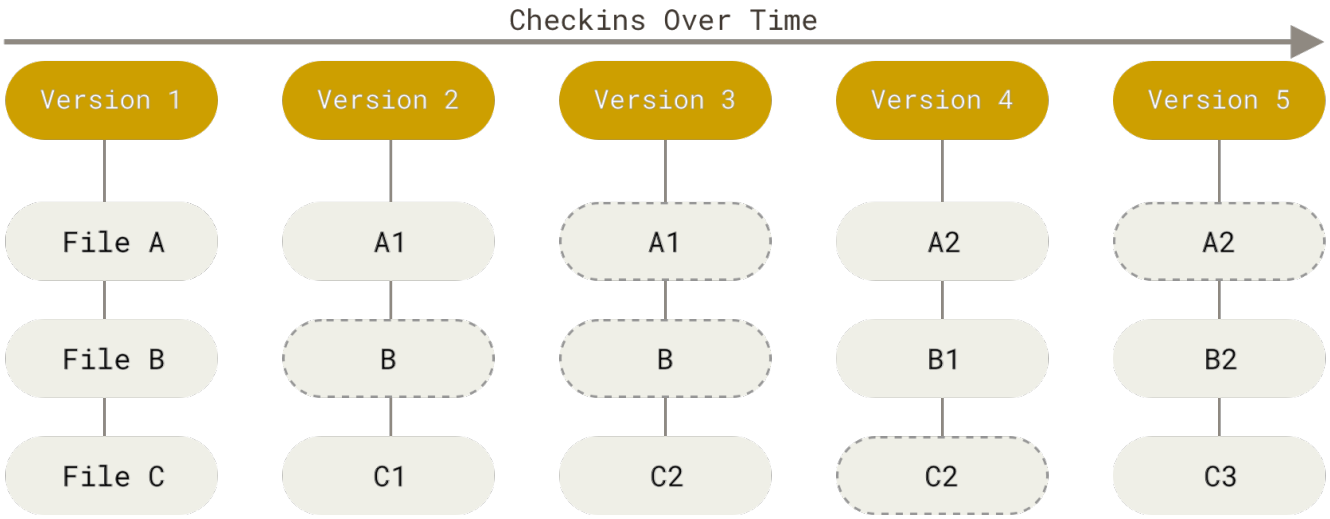


Figure 1. Storing data as snapshots of the project over time.

This is an important distinction between Git and nearly all other VCSs. It makes Git reconsider almost every aspect of version control that most other systems copied from the previous generation. This makes Git more like a mini filesystem with some incredibly powerful tools built on top of it, rather than simply a VCS. We'll explore some of the benefits you gain by thinking of your data this way when we cover Git branching in Git Branching.

Nearly Every Operation Is Local

Most operations in Git need only local files and resources to operate — generally no information is needed from another computer on your network. Because you have the entire history of the project right there on your local disk, most operations seem almost instantaneous. For example, to browse the history of the project, Git doesn't need to go out to the server to get the history and display it for you — it simply reads it directly from your local database. This means you see the project history almost instantly. This also means that there is very little you can't do if you're offline or off VPN. This may not seem like a huge deal, but you may be surprised what a big difference it can make.

Git Has Integrity

Everything in Git is checksummed before it is stored and is then referred to by that checksum. This means it's impossible to change the contents of any file or directory without Git knowing about it. This functionality is built into Git at the lowest levels and is integral to its philosophy. You can't lose information in transit or get file corruption without Git being able to detect it.

The mechanism that Git uses for this checksumming is called a SHA hash. This is a string composed of hexadecimal characters (0–9 and a–f) and calculated based on the contents of a file or directory structure in Git. A SHA-1 hash looks something like this:

```
24b9da6552252987aa493b52f8696cd6d3b00373
```

You will see these hash values all over the place in Git because it uses them so much. In fact, Git stores everything in its database not by file name but by the hash value of its contents.

Git Generally Only Adds Data

When you do actions in Git, nearly all of them only *add* data to the Git database. It is hard to get the system to do anything that is not undoable or to make it erase data in any way. As with any VCS, you can lose or mess up changes you haven't committed yet, but after you commit a snapshot into Git, it is very difficult to lose, especially if you regularly push your database to another repository.

This makes using Git a joy because we know we can experiment without the danger of severely screwing things up. For a more in-depth look at how Git stores its data and how you can recover data that seems lost, see Undoing Things.

The Three States

Pay attention now — here is the main thing to remember about Git if you want the rest of your learning process to go smoothly. Git has three main states that your files can reside in: *modified*, *staged*, and *committed*:

- Modified means that you have changed the file but have not committed it to your database yet.
- Staged means that you have marked a modified file in its current version to go into your next commit snapshot.
- Committed means that the data is safely stored in your local database.

This leads us to the three main sections of a Git project: the working tree, the staging area, and the Git directory.

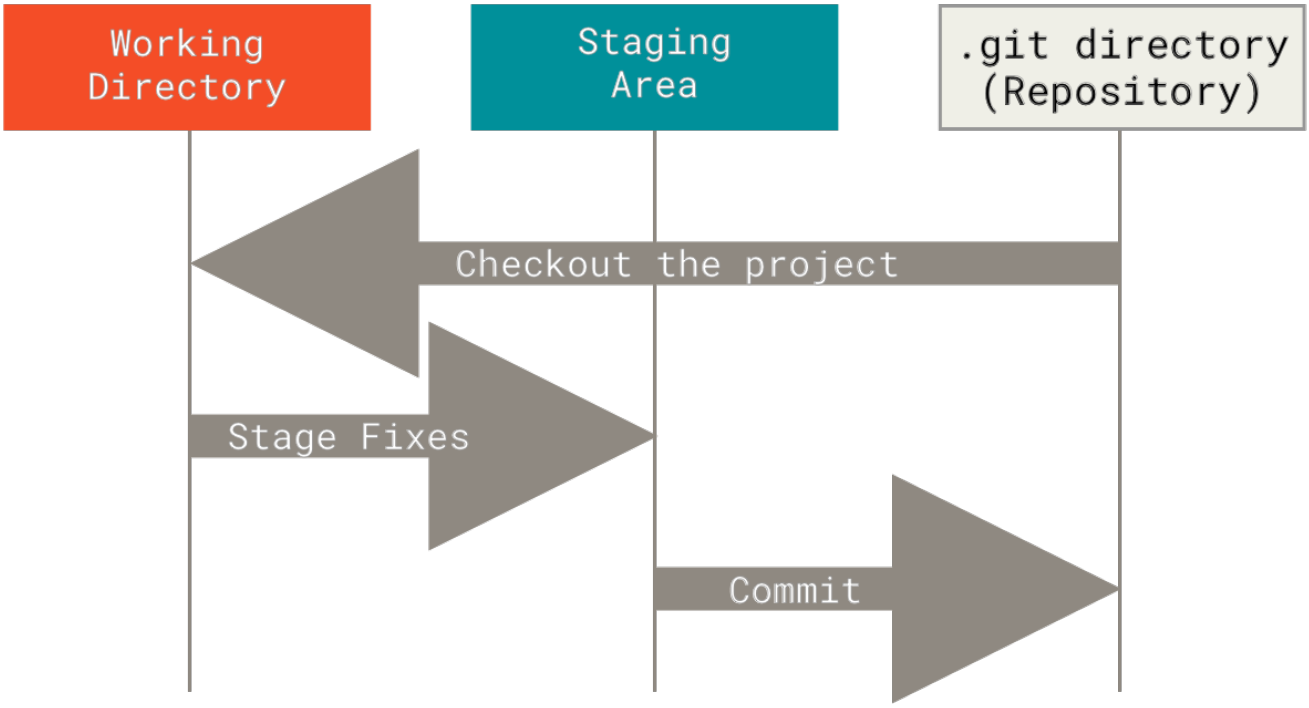


Figure 2. Working tree, staging area, and Git directory.

The working tree is a single checkout of one version of the project. These files are pulled out of the compressed database in the Git directory and placed on disk for you to use or modify.

The staging area is a file, generally contained in your Git directory, that stores information about what will go into your next commit. Its technical name in Git parlance is the “index”, but the phrase “staging area” works just as well.

The Git directory is where Git stores the metadata and object database for your project. This is the most important part of Git, and it is what is copied when you *clone* a repository from another computer.

If a particular version of a file is in the Git directory, it’s considered *committed*. If it has been modified and was added to the staging area, it is *staged*. And if it was changed since it was checked out but has not been staged, it is *modified*.

The basic Git workflow goes something like this:

1. You modify files in your working tree.
2. You stage your changes for your next commit.
3. You do a commit, which adds a new snapshot of the project to the Git directory by combining the previous snapshot with the staged changes.

In Git Basics, you’ll learn more about these states and how you can either take advantage of them or skip the staged part entirely.

Git Interface

There are a lot of different ways to use Git. Visual Studio Team Explorer provides basic Git functionality while Git Extensions provides acceess to the full power of Git.

Most tutorials cover usage of Git on the command line, because most GUIs only implement a partial subset of Git functionality for simplicity, choice of graphical client is a matter of personal taste.

A good way to learn Git would be to open Terminal in macOS or Command Prompt or PowerShell in Windows and follow the examples in the ProGit book. This tutorial illustrates the ProGit Book examples with clips of the Visual Studio user interface.

Installing Git on Windows

An easy way to get Git installed is by installing GitHub Desktop. The installer includes a command line version of Git as well as the GUI. It also works well with PowerShell, and sets up solid credential caching and sane CRLF settings. We'll learn more about those things a little later, but suffice it to say they're things you want. You can download this from the GitHub Desktop website.

The most official build is available for download on the Git website. Just go to https://git-scm.com/download/win and the download will start automatically. Note that this is a project called Git for Windows, which is separate from Git itself; for more information on it, go to https://gitforwindows.org.

To get an automated installation you can use the Git Chocolatey package. Note that the Chocolatey package is community maintained.

First-Time Git Setup

https://devblogs.microsoft.com/devops/customize-git-settings-in-visual-studio/ Before using Git, you may want to do a few things to customise your settings. You should only need to customise global settings once on any given computer.

Open the Git Settings dialog from the Team Explorer Home,

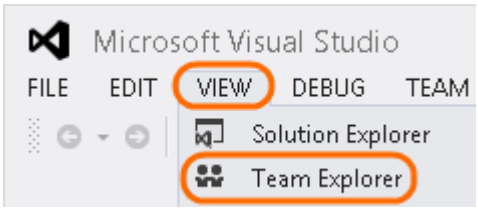


Figure 3. Team Explorer.

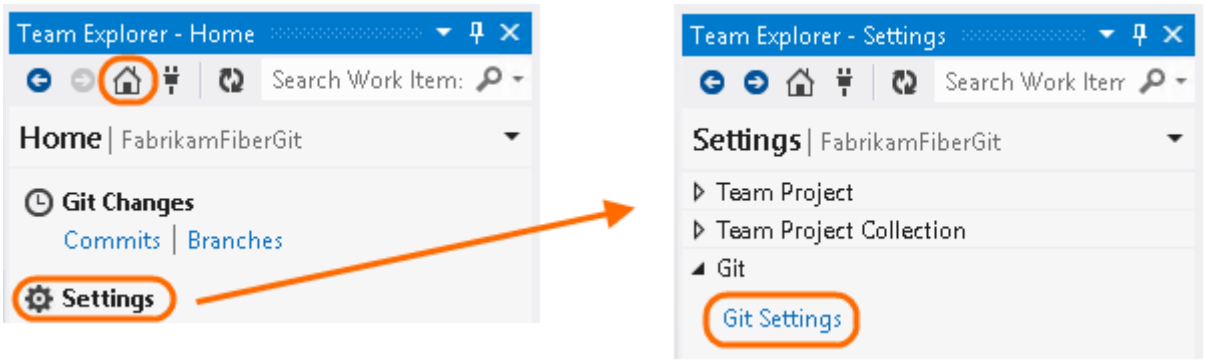


Figure 4. Open Git Settings.

Enter the username and address to be associated with your commits, and the default base directory

where local new or cloned repositories will be created.

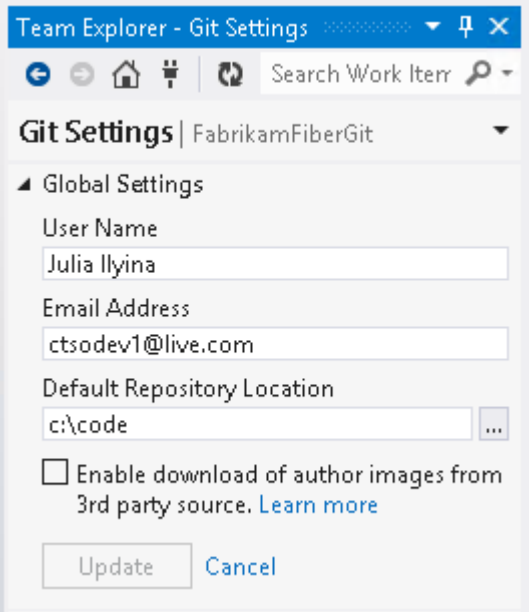


Figure 5. Git Global Settings.

Summary

You should have a basic understanding of what Git is and how differs from any centralized version control systems you may have been using previously. You should also now have Visual Studio set up with your personal identity. It's now time to learn some Git basics.

Git Basics

If you can read only one chapter to get going with Git, this is it. This chapter covers the basic commands you need to do the vast majority of the things you'll eventually spend your time doing with Git. By the end of the chapter, you should be able to configure and initialize a repository, begin and stop tracking files, and stage and commit changes. You should also know how to set up Git to ignore certain files and file patterns, how to undo mistakes quickly and easily, how to browse the history of your project and view changes between commits, and how to push and pull from remote repositories.

Getting a Git Repository

You typically obtain a Git repository in one of four ways:

1. You can create a new project with a Git repository.
2. You can add a Git Repository to an existing project.
3. You can *clone* an existing Git repository from elsewhere.
4. You create an new repository in a new working folder.

In each case, you end up with a Git repository on your local machine, ready for work.

Creating a New Project with a Git Repository

When you are creating a new project or solution, you can ask Visual Studio to place it under version control.

Figure 6. Create a new Project and Repository.

Adding a Git Repository to an Existing Project

If you have a project that is currently not under version control and you want to start controlling it with Git, you first need to open that that project, and then select Add to Source Control in the bottom-right of the lower status bar, and select Git.

[Add To Git Source Control.] | *images/0TWRV-addtosourcecontrol.png*

Figure 7. Add To Git Source Control.

This creates a new subdirectory named `.git` that contains all of your necessary repository files — a Git repository skeleton, Ignore File, Attributes File, and adds commits files and the solution source into the repository.

Cloning an Existing Repository

If you want to get a copy of an existing Git repository, you need to *clone* the repository. In Team Explorer, open the Connect page by selecting the Connect button. Choose Manage Connections then Connect to Project.

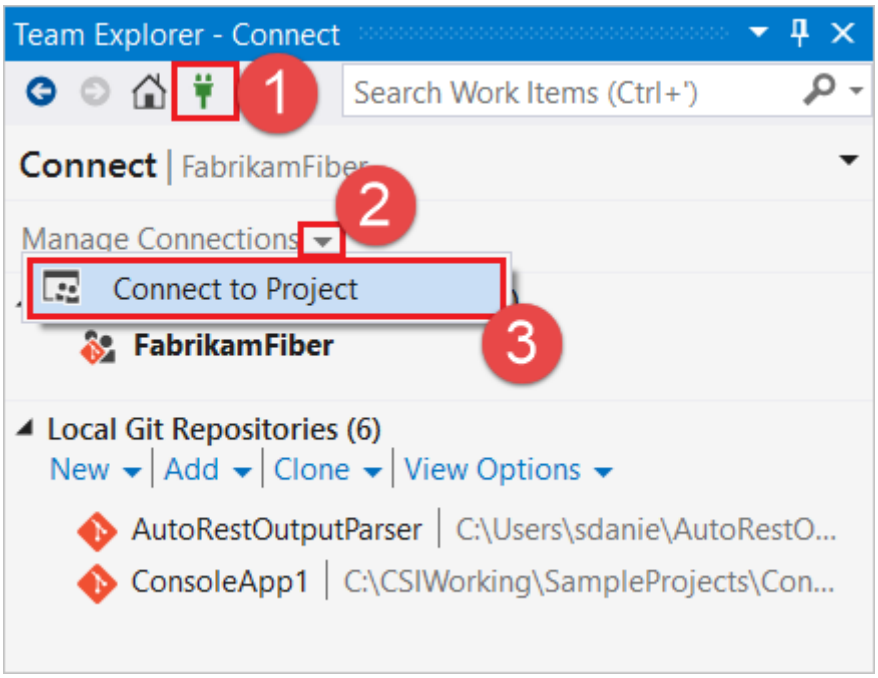


Figure 8. Managing connections.

In Connect to a Project, select the repo you want to clone from the list and select Clone.

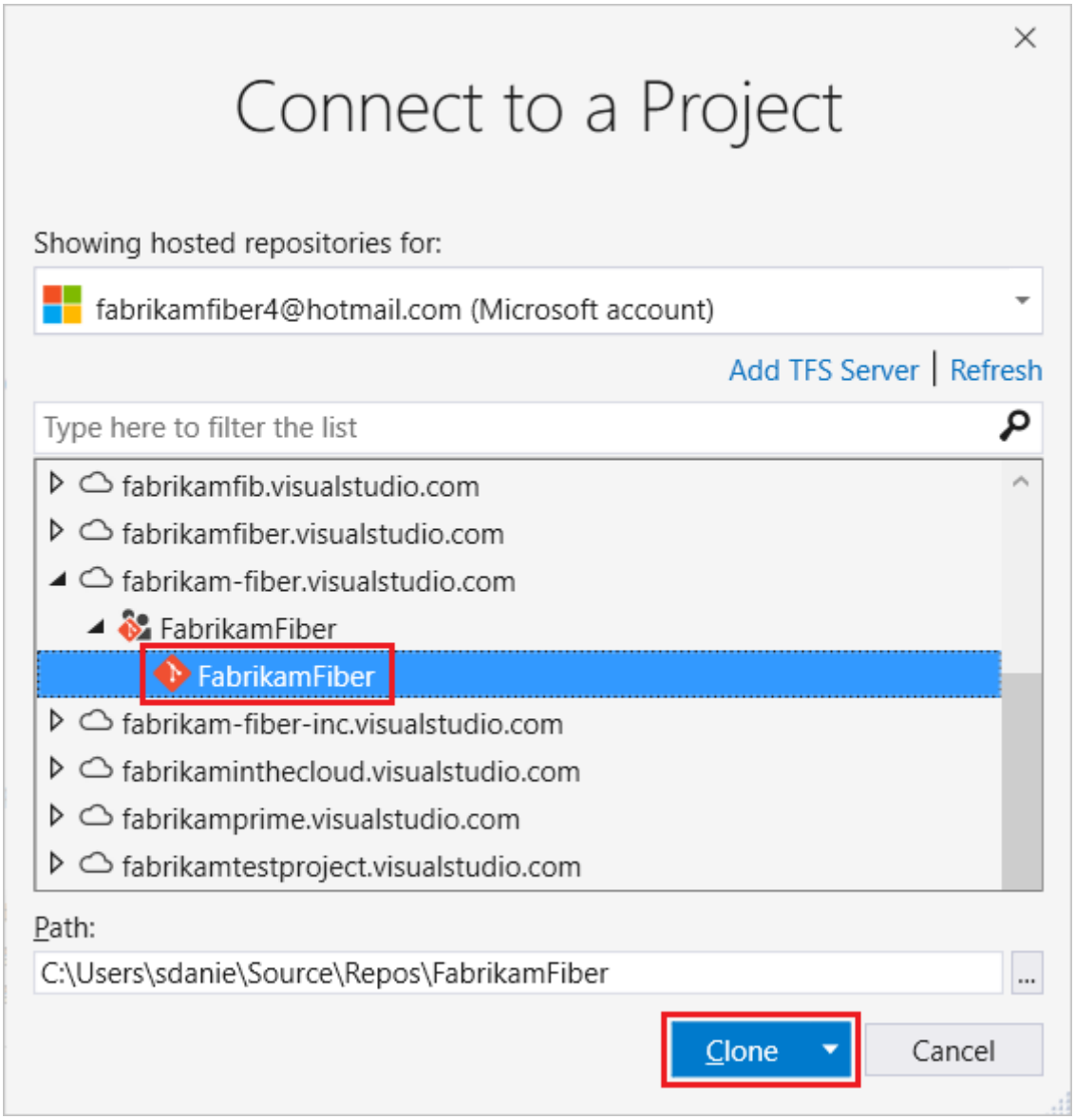


Figure 9. Connect to a project.

That creates a directory named `FabrikamFiber`, initializes a `.git` directory inside it, pulls down all the data for that repository, and checks out a working copy of the latest version. You can go into the new `FabrikamFiber` directory that was just created, you'll see the solution in there, ready to be worked on or used.

If you want to clone the repository into a directory named something other than `libgit2`, you can change the Path in the dialog.

Creating a New Empty Repository

You can also ask Visual Studio to create a new git repository in a new empty folder.

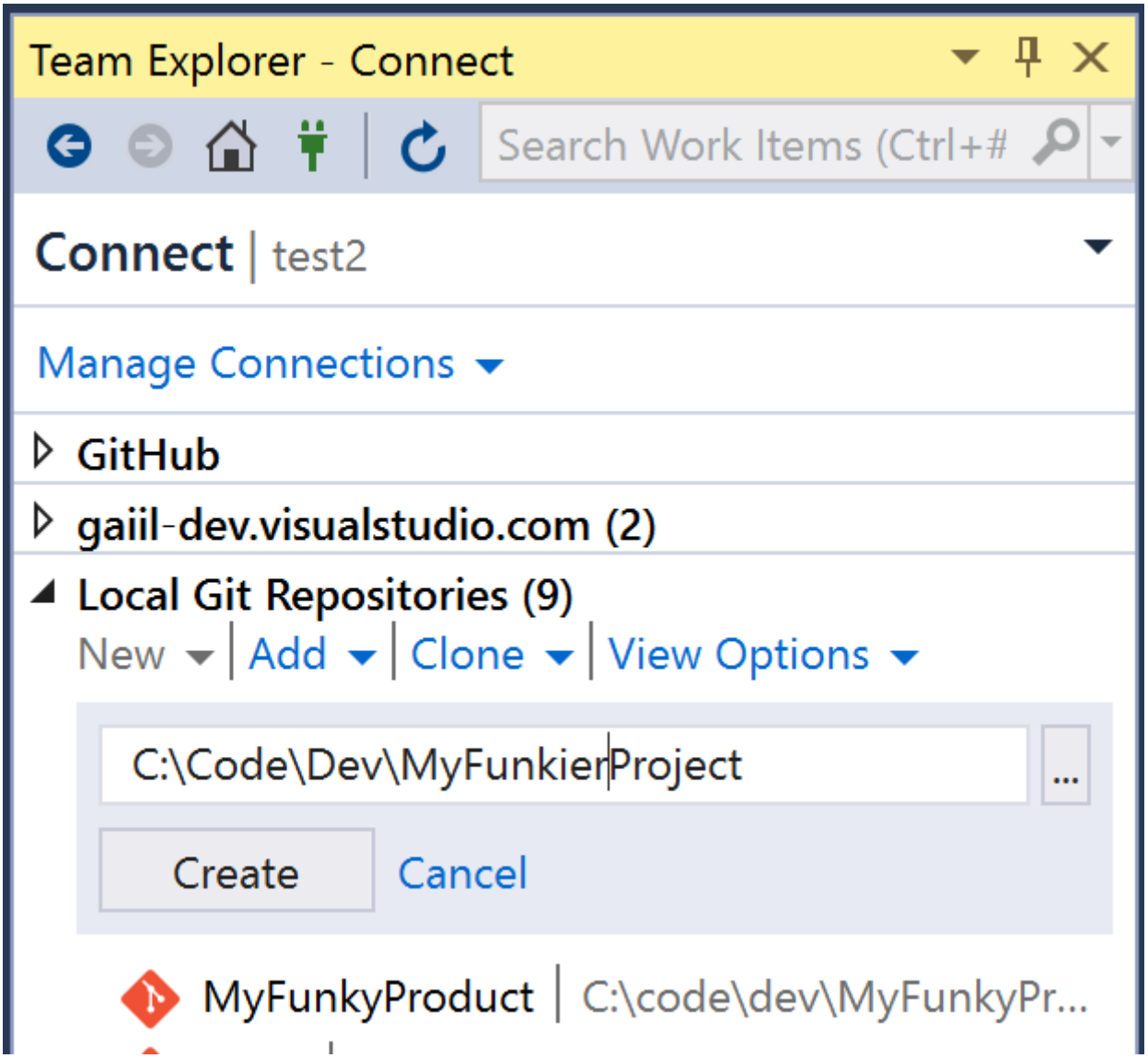


Figure 10. Create Empty Git Repository.

If you start with a blank repository, remember to set up an appropriate Git Ignore and Git Attributes files.

Git Ignore and Attributes settings

Ignore and Attributes file settings can be checked in the repository settings.

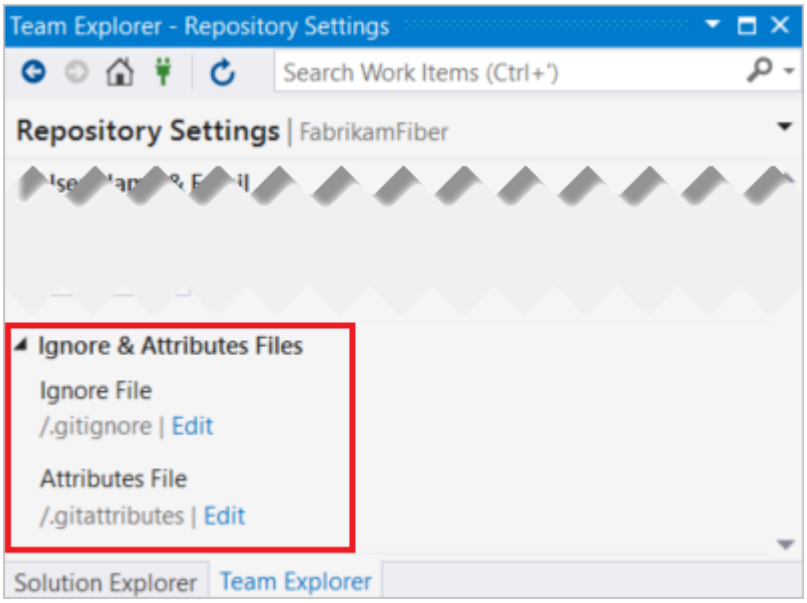


Figure 11. Ignore and Attributes Files.

Repository settings are scoped to each local Git repository on the dev machine and are normally commited so that everyone else on the team uses the same repository settings.

Ignore File: To avoid checking in temporary non-source items such as binaries, you can specify a .gitignore file. See Ignoring files and gitignore(5) Manual Page.

Attributes File: To specify options such as how the system handles line-breaks, you can specify a .gitattributes file. See Customizing Git – Git Attributes.

Recording Changes to the Repository

At this point, you should have a *bona fide* Git repository on your local machine, and a checkout or *working copy* of all of its files in front of you. Typically, you'll want to start making changes and committing snapshots of those changes into your repository each time the project reaches a state you want to record.

Remember that each file in your working directory can be in one of two states: *tracked* or *untracked*. Tracked files are files that were in the last snapshot; they can be unmodified, modified, or staged. In short, tracked files are files that Git knows about.

Untracked files are everything else — any files in your working directory that were not in your last snapshot and are not in your staging area. When you first clone a repository, all of your files will be tracked and unmodified because Git just checked them out and you haven't edited anything.

As you edit files, Git sees them as modified, because you've changed them since your last commit. As you work, you selectively stage these modified files and then commit all those staged changes, and the cycle repeats.

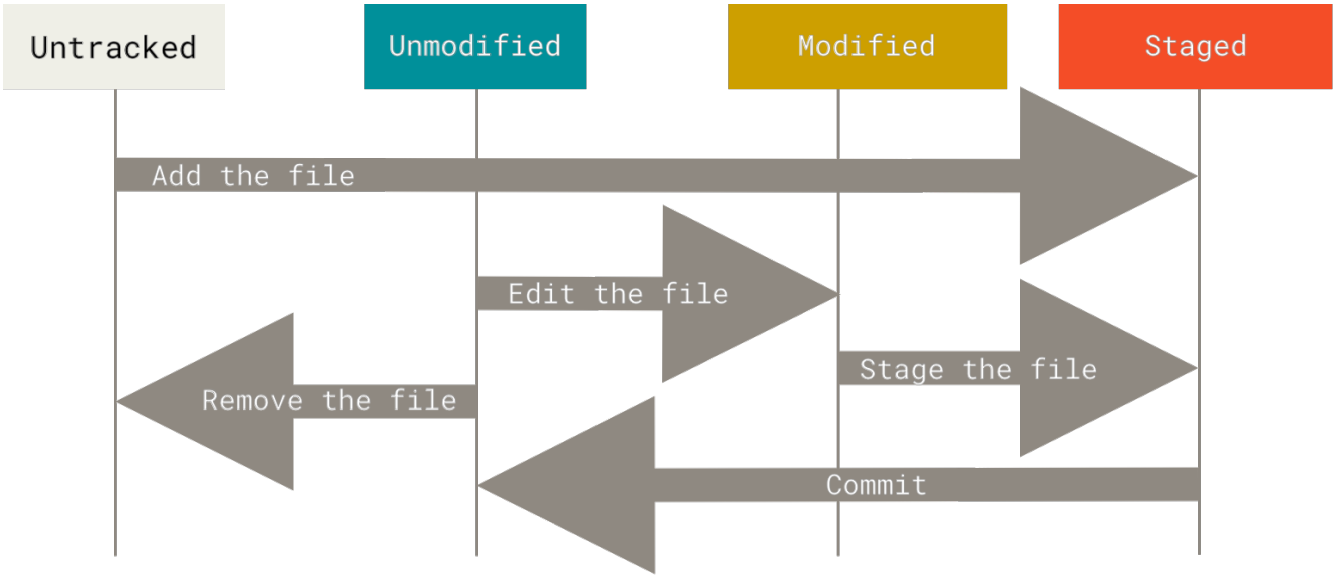


Figure 12. The lifecycle of the status of your files.

Ignored Files

Git needs to be know which files are non-source files that should not be tracked, by setting up an appropriate ".gitignore" file . Visual Studio automatically sets up a `.gitignore` suitable for Visual-Studio related non-source files when adding a project to Git.

Setting up an appropriate `.gitignore` file for a new repository before you get going is generally a good idea so you don't accidentally commit files that you really don't want in your Git repository, such as user settings and automatically generated files. Commits polluted with files that should not be tracked can may require the use of advanced Git history rewriting tools if left to fester, to avoid crippling the standard collaboration and conflict management tools.

GitHub maintains a fairly comprehensive list of good `.gitignore` file examples for dozens of projects and languages at https://github.com/github/gitignore if you want a starting point for your project.

Checking the Status of Your Files

The main tool you use to determine which files are in which state is the Changes view in Team Explorer. When you are ready to stage changes, open up this view.

Tracking New Files

In order to begin tracking a new file, you click on add or right-click on the file and **Stage** it.

Staging Modified Files

When you change (or delete) a file that was already tracked, you can right-click and **Stage** it from the **Changes** section.

If you make additional changes after staging a file, you should stage it again to have those additional changes included in the next commit. If you commit before staging again, the version of the file at the time you staged it is what will be in the subsequent historical snapshot.

Deleting Tracked Files

When you stage and commit a file deletion, the file will no longer be tracked after the next commit. This can sometimes be useful if you accidentally committed a file which should not be tracked.

Ignoring Files

If a non-source file like a user setting, a binary or other build artifact is listed under **Changes**, you can ignore the file with **Ignore this local item** or ignore the type of file with **Ignore this extension**.

Only untracked files can be ignored, not changes to files that git is already tracking.

Deleting Untracked Files

Untracked files which are not ignored can be deleted to ensure the working directory is clean working directory with no pending changes.

Try to keep your working folder clear of files which should not be tracked, by ensuring temporary files are identified by your `.gitignore`, for example using a .bak or .tmp file extension.

Viewing Your Staged and Unstaged Changes

If you want to know exactly what you changed, not just which files were changed you right-click on a modified file and choose **Compare with Unmodified...**.

Committing Your Changes

When your staging area is set up the way you want it, you can commit your changes.

[] | *vimeo*

Committing Selected Changes.

Remember that the commit records the snapshot you set up in your staging area. Anything you didn't stage is still sitting there modified; you can do another commit to add it to your history. Every time you perform a commit, you're recording a snapshot of your project that you can revert to or compare to later.

Before sharing your commits, you should ideally have a "clean" working folder with no remaining changes. In some cases, you may want to modify your `.gitignore` to recognise more files, or segregate changes into branches as in Git Branching.

Skipping the Staging Area

Although it can be amazingly useful for crafting commits exactly how you want them, staging individual changes is sometimes a bit more complex than you need in your workflow. When you have no staged changes, you can commit all changes by entering a commit message and selecting Commit All.

[] | *vimeo*

Committing All Changes.

This is convenient, but be careful; sometimes this will cause you to include unwanted changes.

Moving/Renaming Files

Unlike many other VCS systems, Git doesn't explicitly track file movement. If you rename a file in Git, no metadata is stored in Git that tells it you renamed the file. However, Git is pretty smart about figuring that out after the fact.

Git spots renames by looking at all the changes in the staging area (or between two commits). Renames will show under changes as renames once the new file and the corresponding deletion of the old file have both been staged.

Viewing the Commit History

After you have created several commits, or if you have cloned a repository with an existing commit history, you'll probably want to look back to see what has happened. We will look at a couple of ways you can review this history.

You can review the history of a single file by right-clicking on it in Solution Explorer and choosing **View History...**.

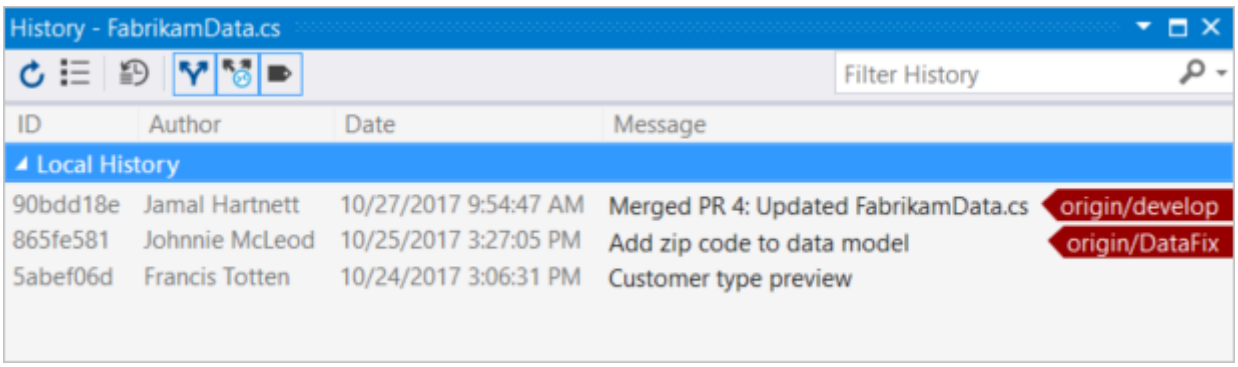


Figure 13. View History.

You can right-click and choose **Compare with previous...** or compare the versions in any two commits by selecting both, right-clicking and selecting **Compare...**.

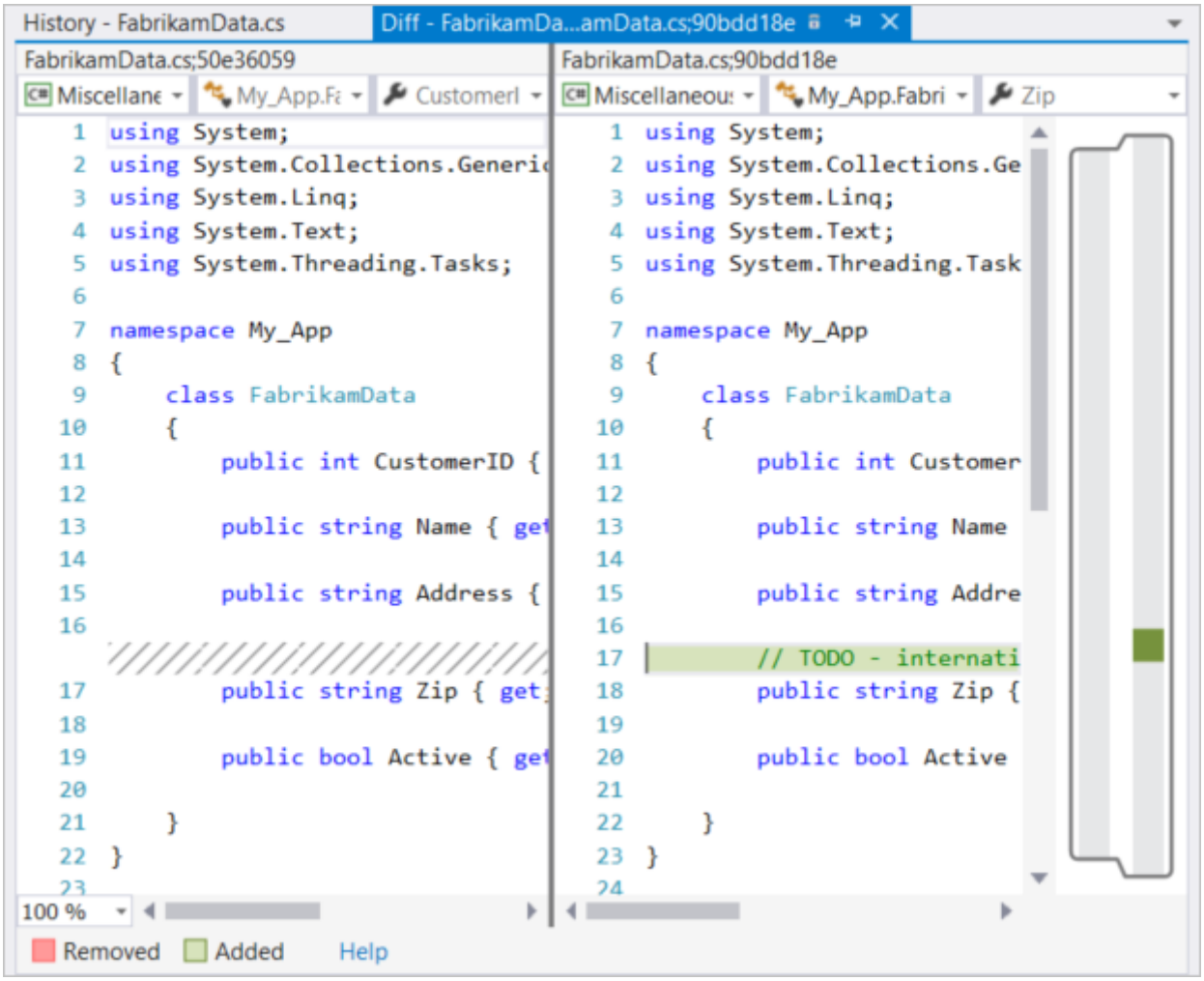


Figure 14. File Differences.

Undoing Things

At any stage, you may want to undo something. Here, we'll review a few basic tools for undoing changes that you've made. Be careful, because you can't always undo some of these undos. This is one of the few areas in Git where you may lose some work if you do it wrong.

One of the common undos takes place when you commit too early and possibly forget to add some files, or you mess up your commit message. If you want to redo that commit, you can make the additional changes you forgot and amend your previous commit. As with making a new commit, stage the changes you need, or leave all changes unstaged to commit all changes.

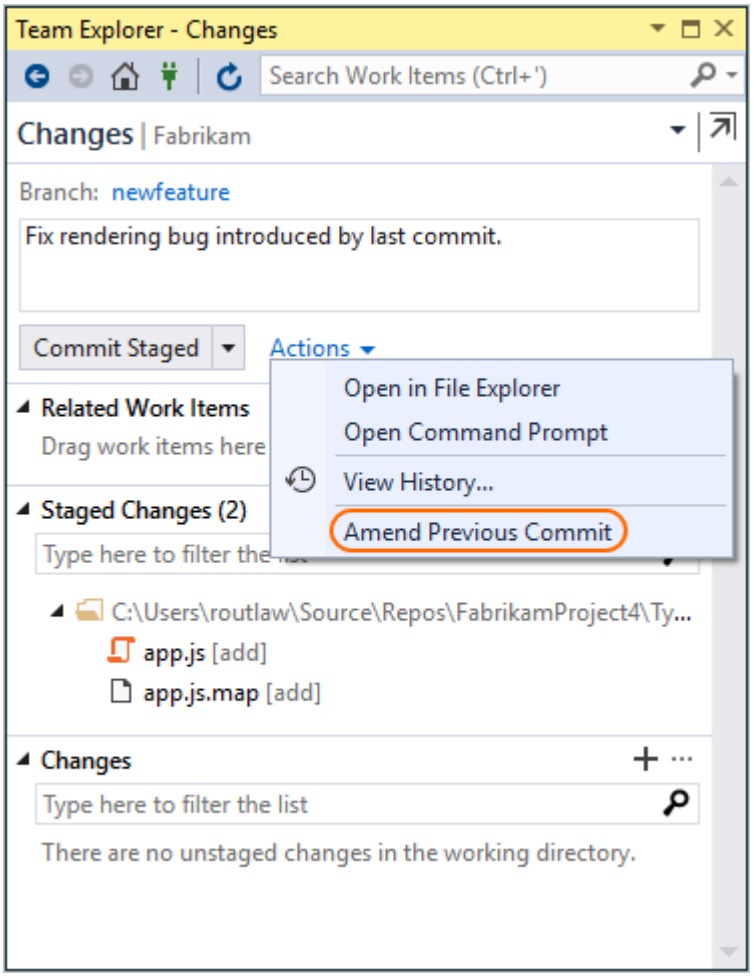


Figure 15. Amend previous commit.

If you want to amend the commit message, you can view the branch history, view the commit details, edit the commit message and click Amend Message.

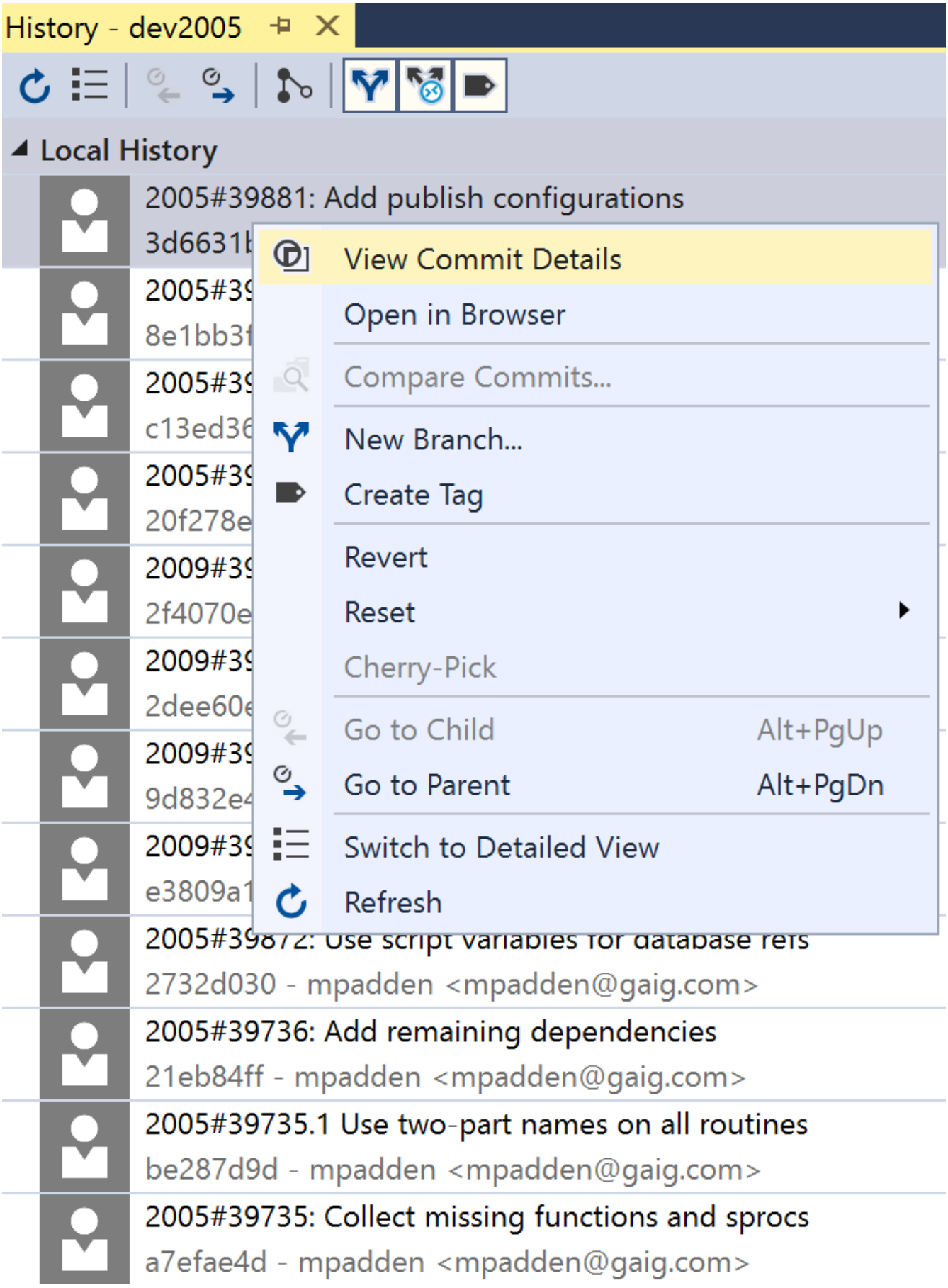


Figure 16. View previous commit details.

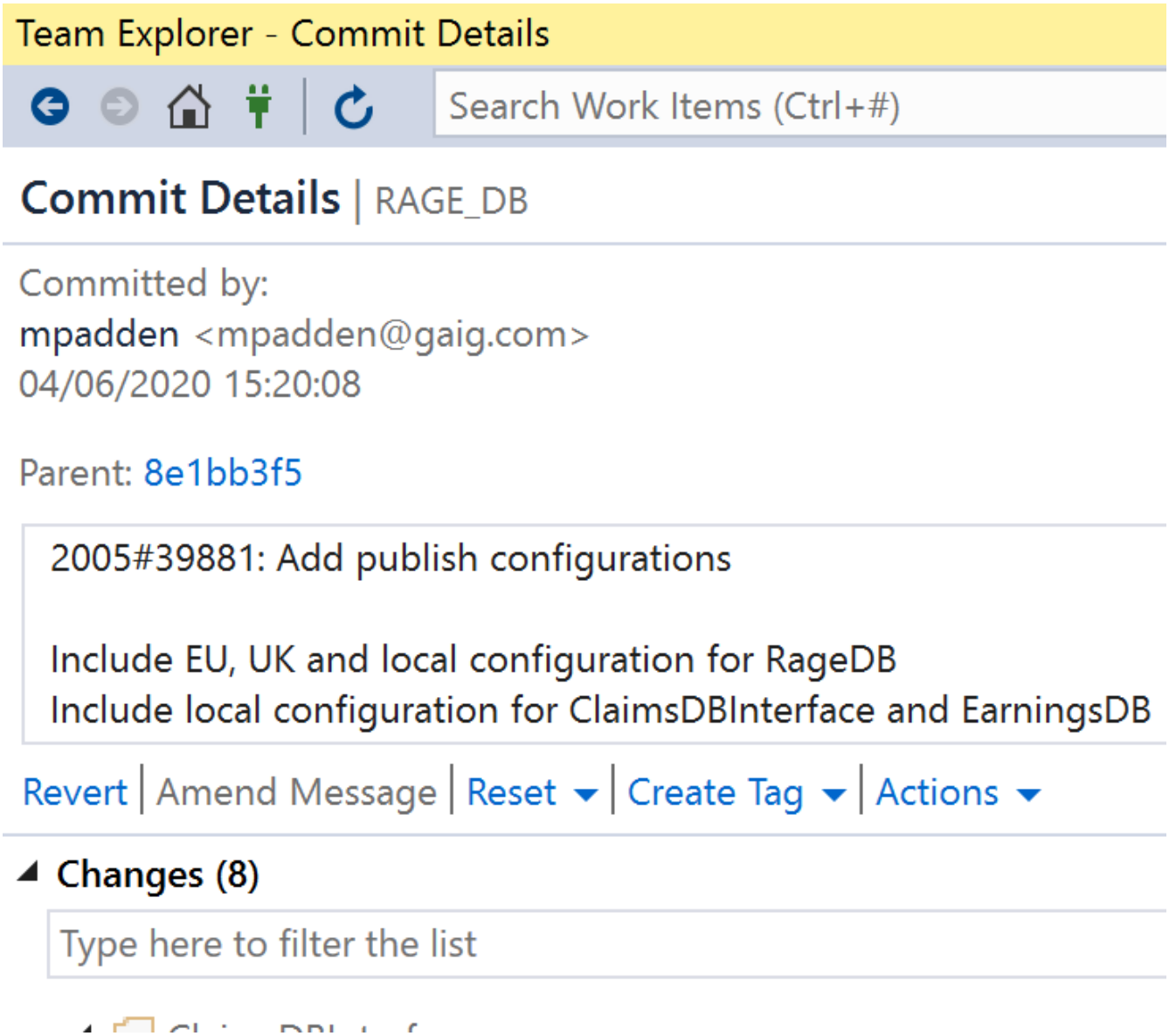


Figure 17. Amend previous commit message.

The amend message option only appears on the last commit in the current branch. It is enabled when the commit message is edited.

In either case you end up with a single commit — the second commit replaces the results of the first.

When you're amending your last commit, you're not so much fixing it as *replacing* it entirely with a new, improved commit that pushes the old commit out of the way and puts the new commit in its place. Effectively, it's as if the previous commit never happened, and it won't show up in your repository history.

The obvious value to amending commits is to make minor improvements to your last commit, without cluttering your repository history with commit messages of the form, "Oops, forgot to add a file" or "Darn, fixing a typo in last commit".

Unstaging a Staged File

Let's say accidentally staged a file and don't want to commit the changes to the file. This can be useful if you accidentally stage a file which shouldn't be tracked. How can you unstage it?

Under the **Changes** section, right-click on the file you want and click **Unstage**.

Unmodifying a Modified File

What if you realize that you don't want to keep your changes to the file? How can you easily unmodify it — restore it back to what it looked like before you modified it?

Under the **Changes** section, right-click on the file you want and click **Undo Changes...**.

[] | *vimeo*

Discard changes to a file.

Unstaging a file does not touch the file in your working directory, it mergely removes it from the staging area.

It's important to understand that **Undo Changes...** is a dangerous function. Any local changes you made to that file are gone — Git just replaced that file with the most recently-committed version. Don't use this function unless you absolutely know that you don't want those unsaved local changes.

If you would like to keep the changes you've made to that file but still need to get it out of the way for now, you can stash them in a temporary branch, which is generally a better way to go. We'll go over branching in Git Branching.

Remember, anything that is *committed* in Git can almost always be recovered. Even commits that were on branches that were deleted or commits that were overwritten with an amended commit can be recovered (see the ProGit book for data recovery). However, anything you lose that was never committed is likely never to be seen again.

Working with Remotes

To be able to collaborate on any Git project, you need to know how to manage your remote repositories. Remote repositories are versions of your project that are hosted on the Internet or network somewhere. You can have several of them, each of which generally is either read-only or read/write for you. Collaborating with others involves managing these remote repositories and pushing and pulling data to and from them when you need to share work. Managing remote repositories includes knowing how to add remote repositories, remove remotes that are no longer valid, manage various remote branches and define them as being tracked or not, and more. In this section, we'll cover some of these remote-management skills.

Remote repositories can be on your local machine.

It is entirely possible that you can be working with a "remote" repository that is, in fact, on the same host you are. The word "remote" does not necessarily imply that the repository is somewhere else on the network or Internet, only that it is elsewhere. Working with such a remote repository would still involve all the standard pushing, pulling and fetching operations as with any other remote.

Showing Your Remotes

The remote servers for a repository can be seen at the bottom of the repository settings.

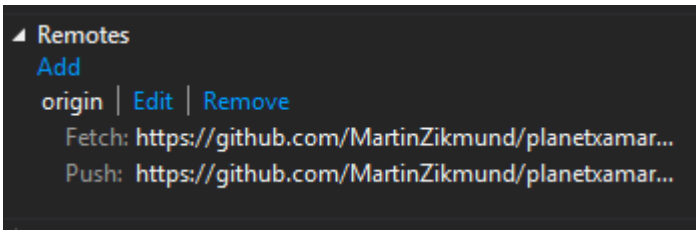


Figure 18. Remotes.

If you've cloned your repository, you should at least see `origin` — that is the default name Git gives to the server you cloned from.

Adding Remote Repositories

Closing a repository implicitly adds the `origin` remote for you. You can add more remotes at the bottom of the repository settings.

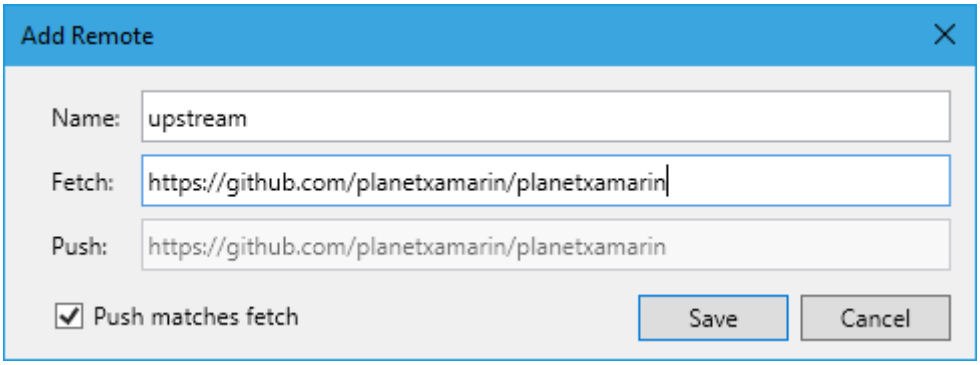


Figure 19. Add Remote.

Once you have added a remote you can **Sync** and choose to **Fetch** from that remote. This will make the branches of that remote available in branches so you can checkout or merge those changes into your fork.

(We'll go over what branches are and how to use them in much more detail in Git Branching.)

Fetching and Pulling from Your Remotes

As you just saw, to get data from your remote projects, you can **Sync** and **Fetch** incoming commits. After you do this, you should have references to all the branches from that remote, which you can merge in or inspect at any time.

If you clone a repository, this automatically adds that remote repository under the name “origin”. So, **Sync**/**Fetch** fetches any new work that has been pushed to that server since you cloned (or last fetched from) it. It’s important to note that the this only downloads the data to your local repository — it doesn’t automatically merge it with any of your work or modify what you’re currently working on. You have to merge it manually into your work when you’re ready.

If your current branch is set up to track a remote branch (see the next section and Git Branching for more information), you can **Pull** to automatically fetch and then merge that remote branch into your current branch. This may be an easier or more comfortable workflow for you; and by default, clone automatically sets up your local `master` branch to track the remote `master` branch (or whatever the default branch is called) on the server you cloned from. Choosing **Sync**/**Pull** generally fetches data from the server you originally cloned from and automatically tries to merge it into the code you’re currently working on.

Pushing to Your Remotes

When you have your project at a point that you want to share, you have to push it upstream. When you choose **Sync**/**Pull** you specifiy the remote and the branch you want to push. The default is normally pushing your `master` branch to your `origin` server.

This command works only if you cloned from a server to which you have write access and if nobody has pushed in the meantime. If you and someone else clone at the same time and they push upstream and then you push upstream, your push will rightly be rejected. You’ll have to fetch their work first and incorporate it into yours before you’ll be allowed to push. See Git Branching for more detailed information on how to push to remote servers.

Tagging

Like most VCSs, Git has the ability to tag specific points in a repository’s history as being important. Typically, people use this functionality to mark release points (`v1.0`, `v2.0` and so on). In this section, you’ll learn how to view existing tags, how to create and delete tags, and what the different types of tags are.

Viewing Your Tags

Viewing tags in Git is done from the **Tags** in the **Home** view.

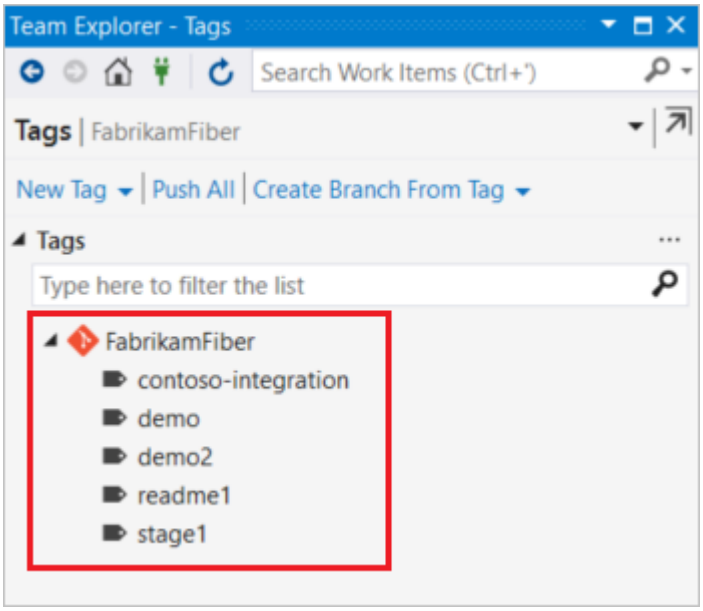


Figure 20. View Tags.

Tags are shown in alphabetical order; the order in which they are displayed has no real importance.

You can also search for tags that match a particular pattern by typing a search term into the **Type here to filter the list** box.

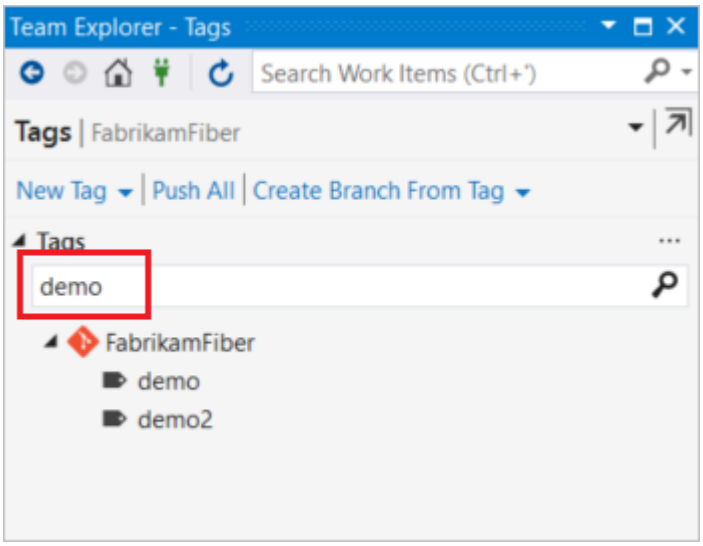


Figure 21. Filter Tags.

Creating Tags

Git supports two types of tags: *lightweight* and *annotated*.

A lightweight tag is very much like a branch that doesn't change — it's just a pointer to a specific commit.

Annotated tags, however, are stored as full objects in the Git database. They're checksummed; contain the tagger name, email, and date; have a tagging message; and can be signed and verified with GNU Privacy Guard (GPG). It's generally recommended that you create annotated tags so you can have all this information; but if you want a temporary tag or for some reason don't want to

keep the other information, lightweight tags are available too.

Annotated Tags

To create an annotated tag against the tip of the current branch, select **New Tag** in the **Tags** view, give a name to the tag and enter a tag message.

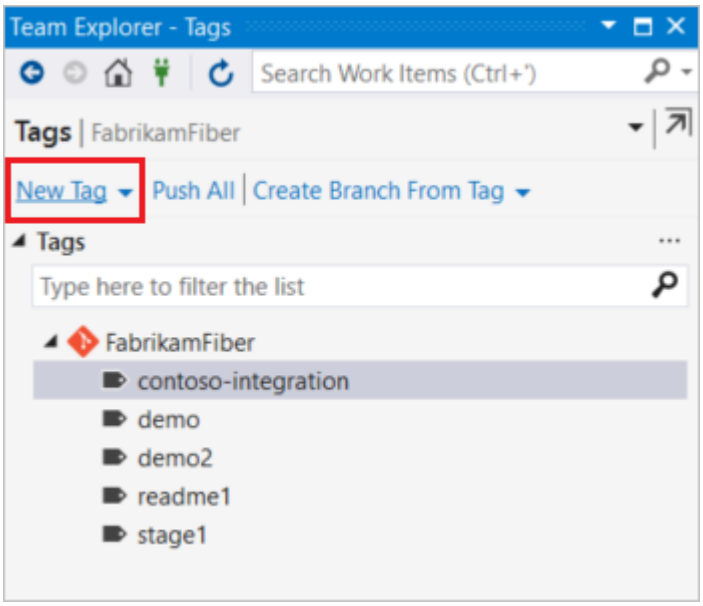


Figure 22. Create Tag.

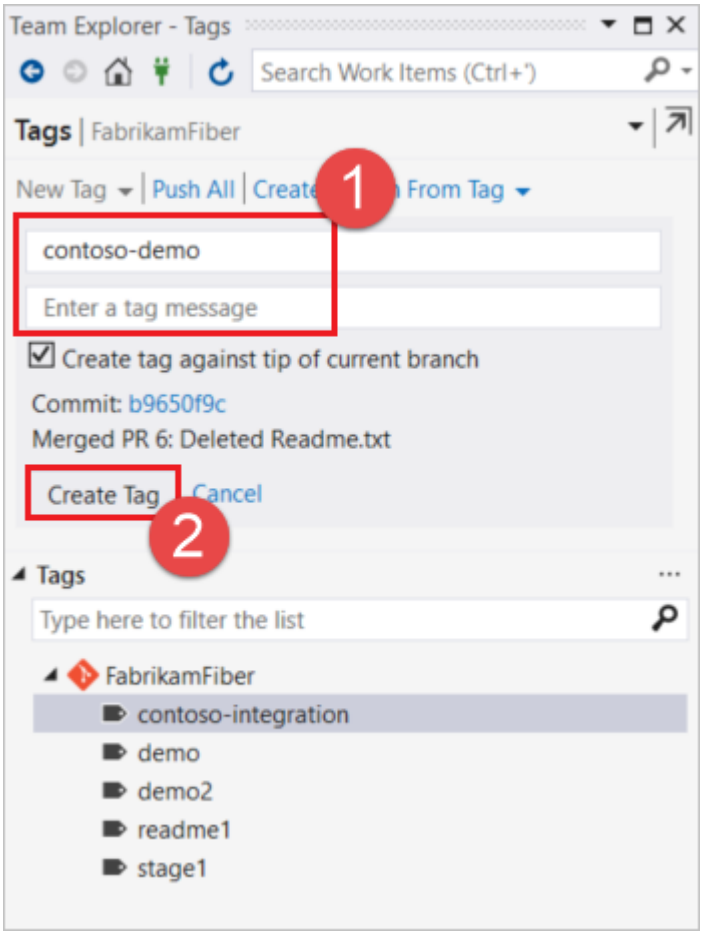


Figure 23. Create Tag Against Tip of Current Branch.

Annotated tags show a tooltip with the tag name, tagger, tag date and message.

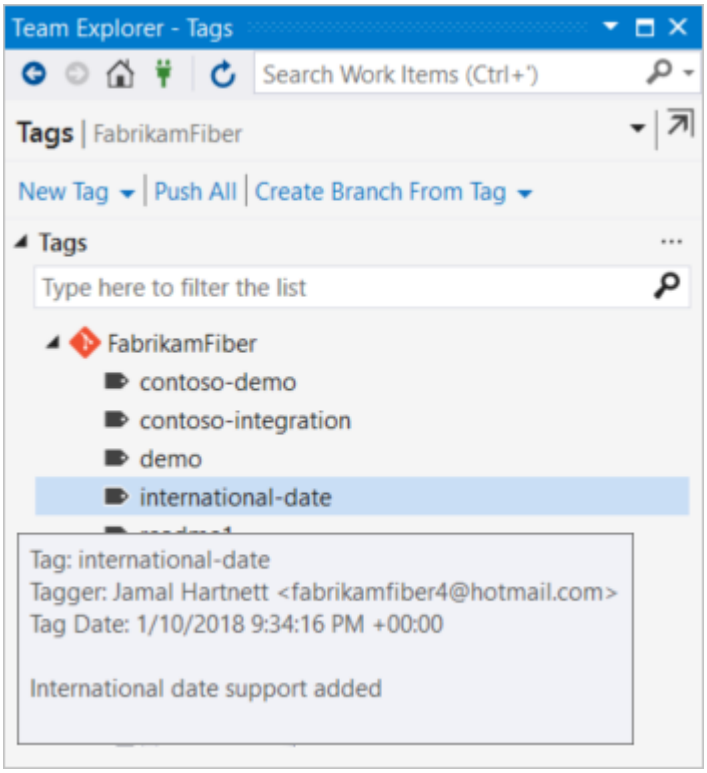


Figure 24. Tag Style.

Lightweight Tags

Another way to tag commits is with a lightweight tag. This is basically the commit checksum stored in a file — no other information is kept. To create a lightweight tag, don't supply a tag message, just provide a tag name. Lightweight tags only show the tag name, with no other details in the tooltip.

Tag Details

For more information about the tagged commit, right-clik the tag and select **View Commit Details**

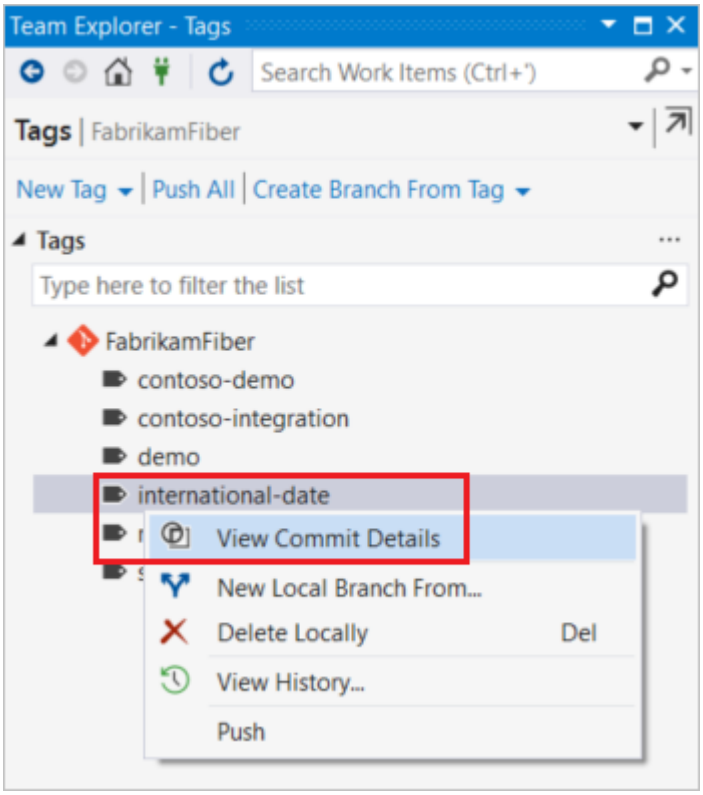


Figure 25. View Commit Details.

Tagging Later

You can create a tag against the tip of another branch by clearing the **Create tag against tip of current branch** check box and selecing a branch from the **Select a branch** drop-down.

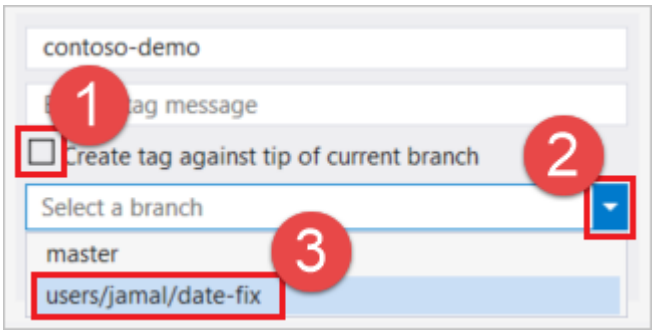


Figure 26. Create Tag From Select Branch.

You can also tag commits after you've moved past them from the history view by right-clicking the desired commit and choosing **Create Tag**.

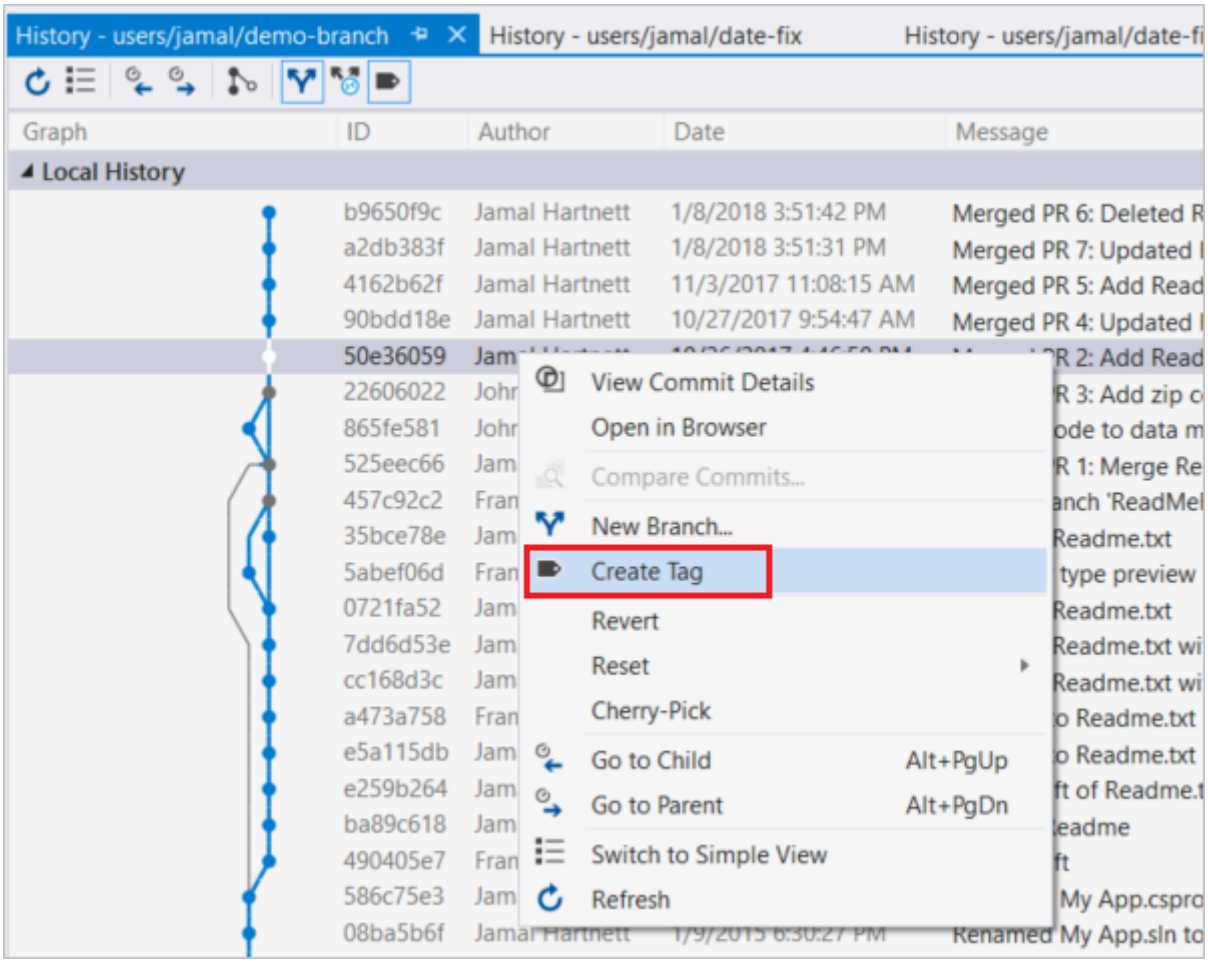


Figure 27. Create Tag From Commit.

Sharing Tags

By default, the `git push` command doesn't transfer tags to remote servers. You will have to explicitly push tags to a shared server after you have created them. This process is just like sharing remote branches — you can right click the tag and choose **Push**. If you have a lot of tags that you want to push up at once, you can also use choose **Push All**. This will transfer all of your tags to the remote server that are not already there.

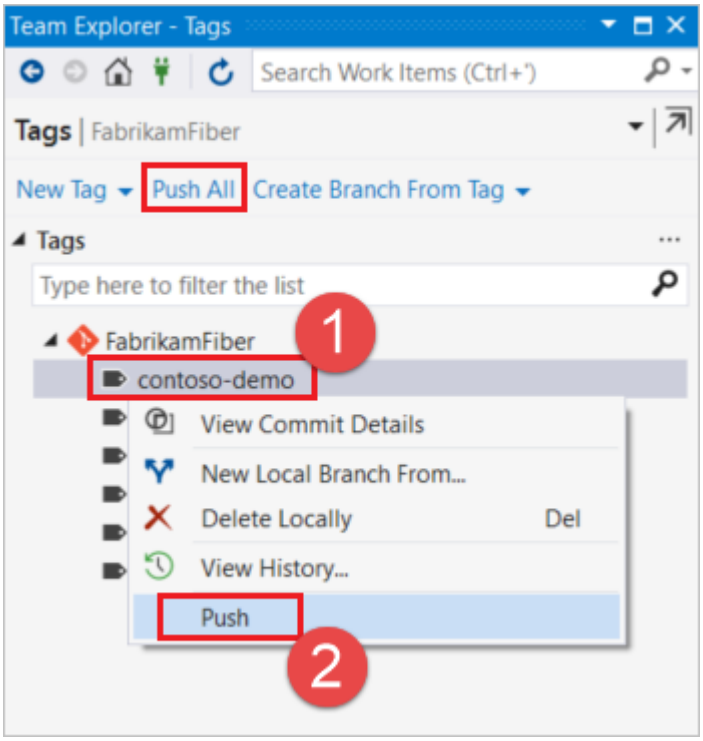


Figure 28. Share Tags.

Now, when someone else clones or pulls from your repository, they will get all your tags as well.

Push All *pushes both types of tags*

There is currently no option to push only lightweight tags.

Deleting Tags

To delete a tag on your local repository, right click the tag and choose **Delete Locally**

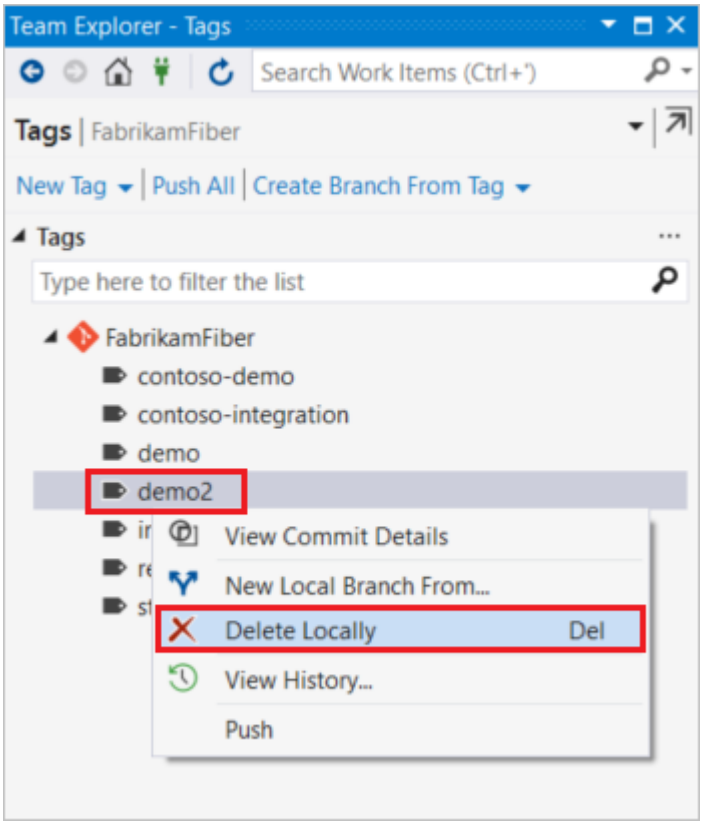


Figure 29. Delete Tag.

Checking out Tags

If you want to create a new working branch from a tag, righ-click the tag and choose **New Local Branch From** or click **Create Branch From tag**.

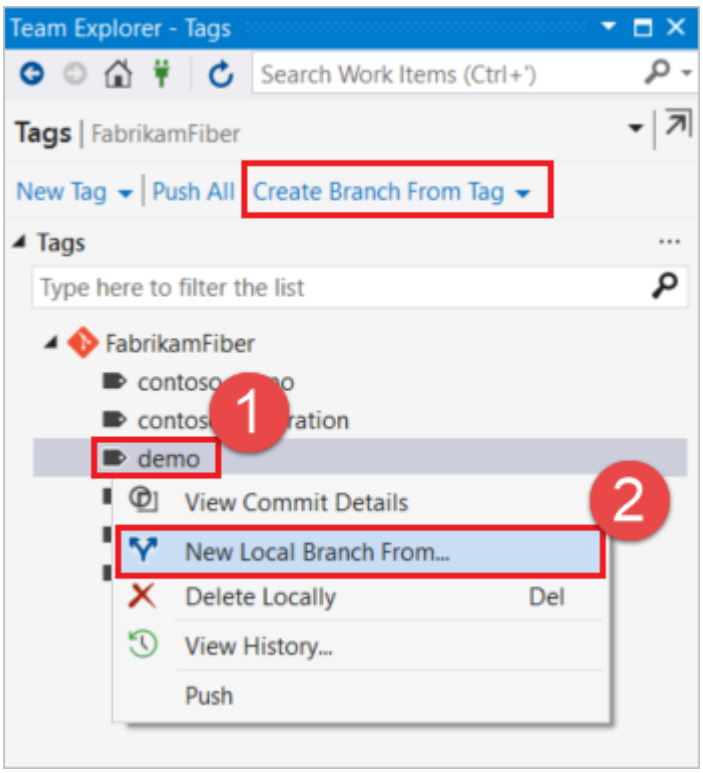


Figure 30. Create Branch From Tag.

Summary

At this point, you can do all the basic local Git operations — cloning or setting up a repository with project tracking rules, making changes, staging and committing those changes, and viewing the history of all the changes the repository has been through. Next, we'll cover Git's killer feature: its branching model.

Git Branching

Nearly every VCS has some form of branching support. Branching means you diverge from the main line of development and continue to do work without messing with that main line. In many VCS tools, this is a somewhat expensive process, often requiring you to create a new copy of your source code directory, which can take a long time for large projects.

Some people refer to Git's branching model as its "killer feature," and it certainly sets Git apart in the VCS community. Why is it so special? The way Git branches is incredibly lightweight, making branching operations nearly instantaneous, and switching back and forth between branches generally just as fast. Unlike many other VCSs, Git encourages workflows that branch and merge often, even multiple times in a day. Understanding and mastering this feature gives you a powerful and unique tool and can entirely change the way that you develop.

Branches in a Nutshell

To really understand the way Git does branching, we need to take a step back and examine how Git stores its data.

As you may remember from Getting Started, Git doesn't store data as a series of changesets or differences, but instead as a series of *snapshots*.

When you make a commit, Git stores a commit object that contains a pointer to the snapshot of the content you staged. This object also contains the author's name and email address, the message that you typed, and pointers to the commit or commits that directly came before this commit (its parent or parents): zero parents for the initial commit, one parent for a normal commit, and multiple parents for a commit that results from a merge of two or more branches.

To visualize this, let's assume that you have a directory containing three files, and you stage them all and commit. Staging the files computes a checksum for each one (the SHA-1 hash we mentioned in Getting Started), stores that version of the file in the Git repository (Git refers to them as *blobs*), and adds that checksum to the staging area:

When you create the commit, Git checksums each subdirectory and stores them as a tree object in the Git repository. Git then creates a commit object that has the metadata and a pointer to the root project tree so it can re-create that snapshot when needed.

Your Git repository now contains five objects: three *blobs* (each representing the contents of one of the three files), one *tree* that lists the contents of the directory and specifies which file names are stored as which blobs, and one *commit* with the pointer to that root tree and all the commit metadata.

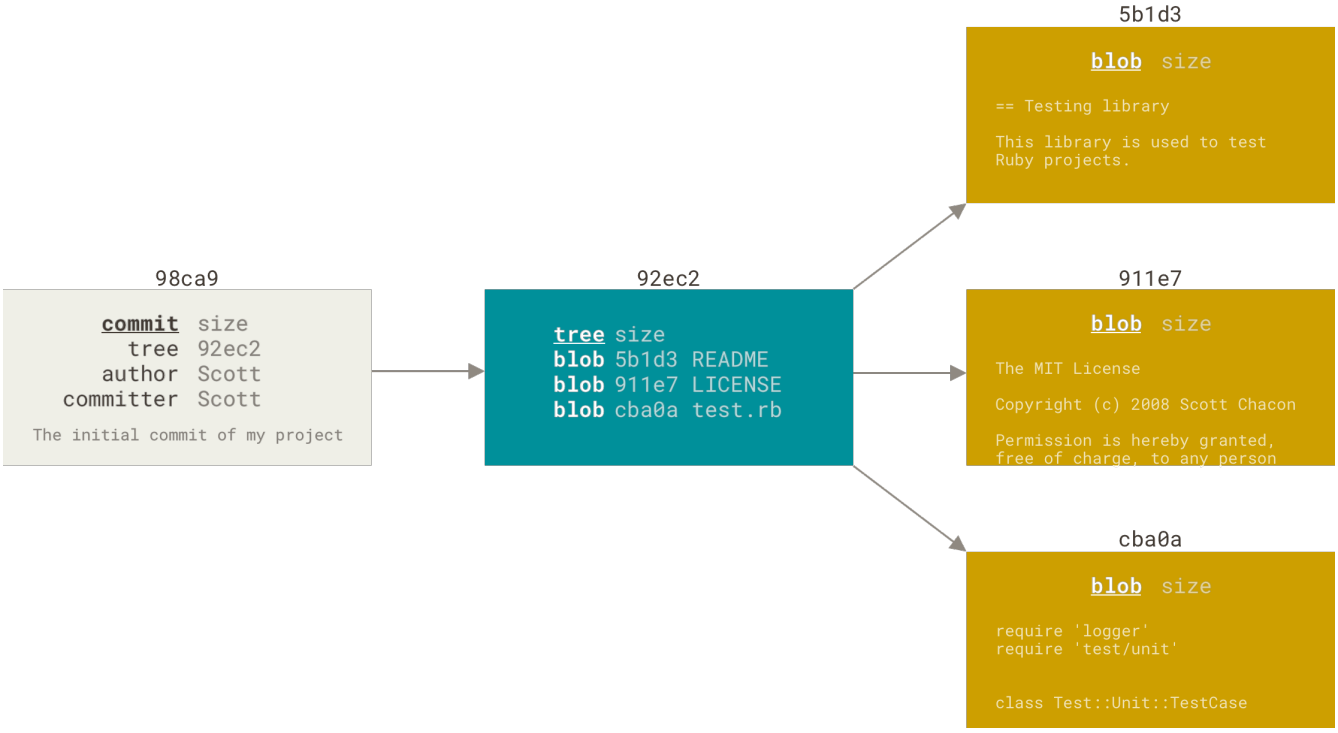


Figure 31. A commit and its tree

If you make some changes and commit again, the next commit stores a pointer to the commit that came immediately before it.

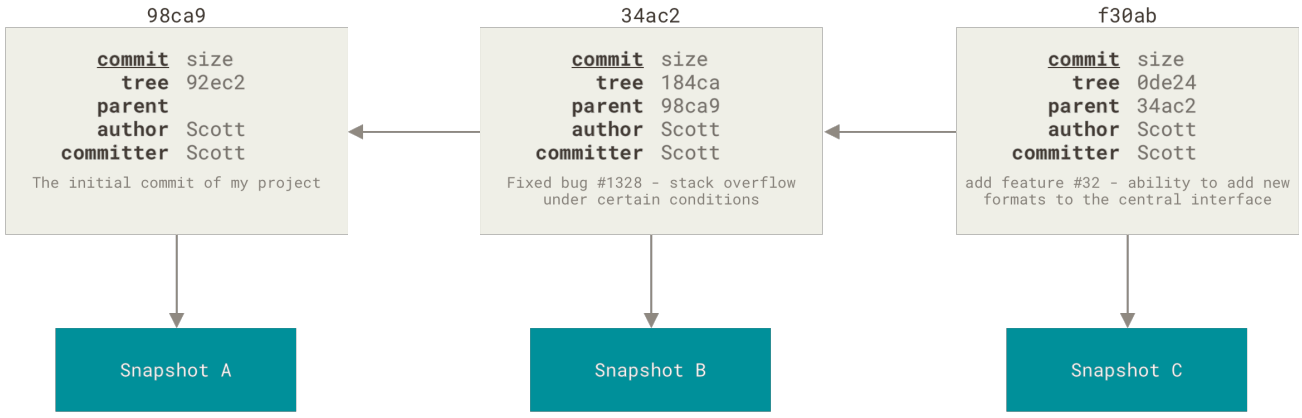


Figure 32. Commits and their parents

A branch in Git is simply a lightweight movable pointer to one of these commits. The default branch name in Git is `master`. As you start making commits, you're given a `master` branch that points to the last commit you made. Every time you commit, the `master` branch pointer moves forward automatically.

The "master" branch in Git is not a special branch. It is exactly like any other branch. The only reason nearly every repository has one is that the `git init` command creates it by default and most people don't bother to change it.

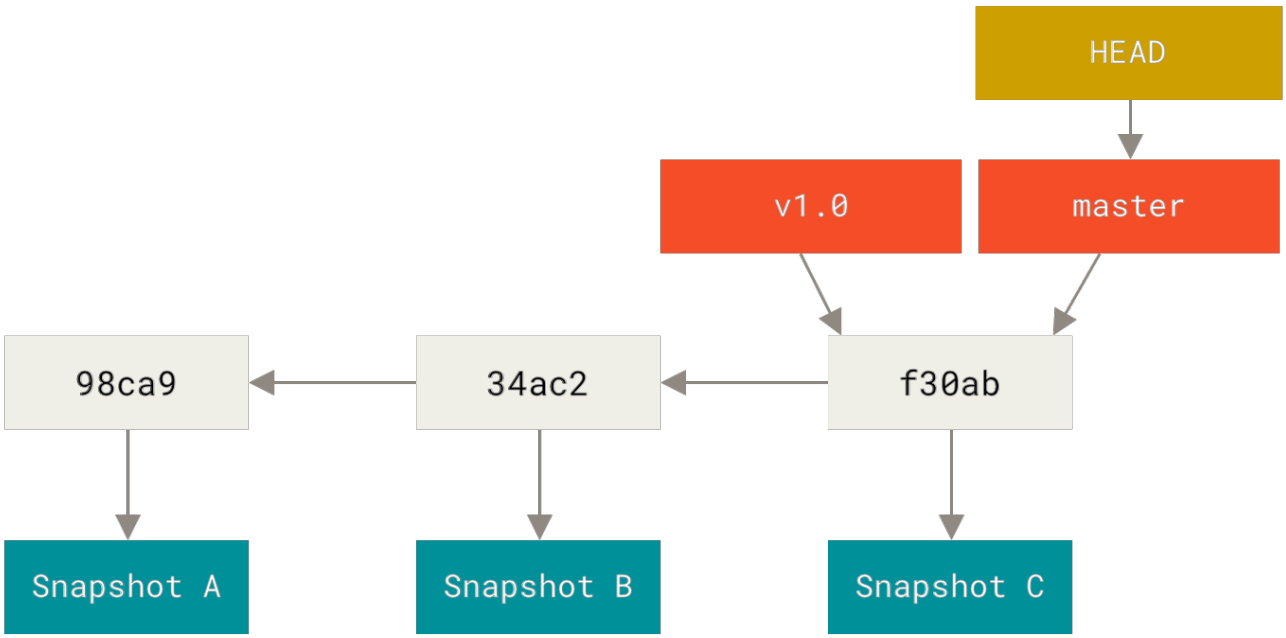


Figure 33. A branch and its commit history

Creating a New Branch

What happens when you create a new branch? Well, doing so creates a new pointer for you to move around. You create a branch from the Team Explorer **Branches** view, but right-clicking on the parent branch (usually `master`) as a starting point and choosing **New Local Branch From...**.

[] | *vimeo*

Create Branch.

This creates a new pointer to the tip of the existing branch. Git keeps track of what branch you are on with an internal `HEAD` pointer.

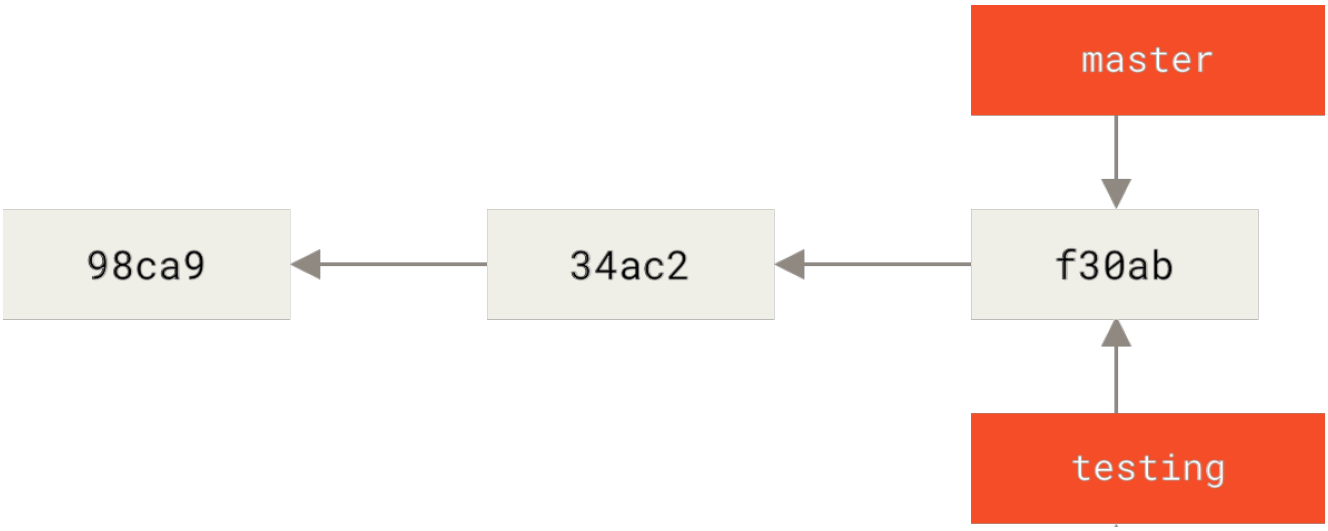


Figure 34. Two branches pointing into the same series of commits

Switching Branches

To switch to an existing branch, you **Checkout** the branch you want to work on .

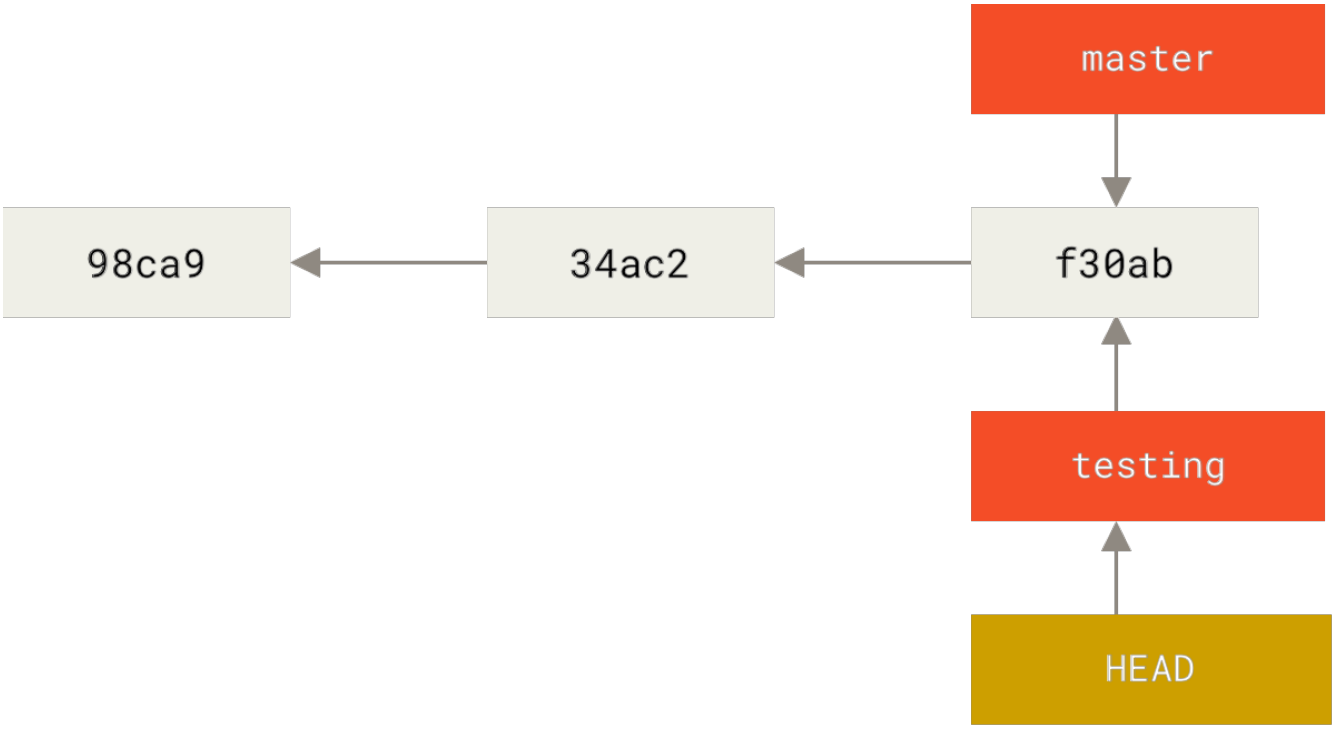


Figure 35. HEAD points to the current branch

What is the significance of that? When you do another commit, the head branch moves forward.

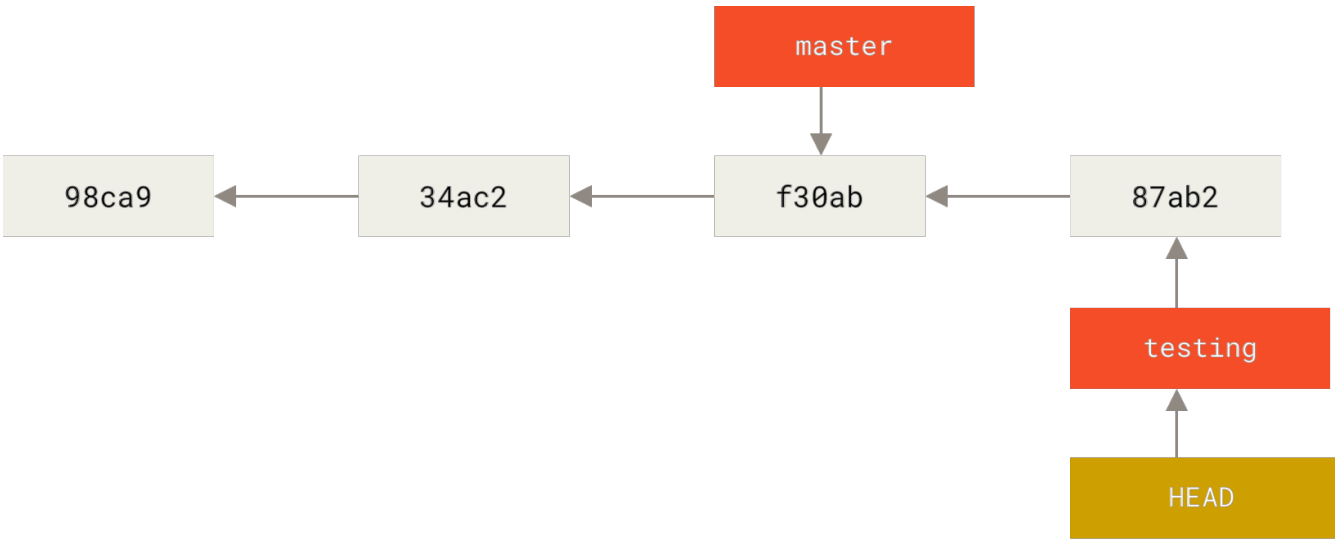


Figure 36. The HEAD branch moves forward when a commit is made

Figure 37. HEAD moves when you checkout.

The checkout did two things. It moved the HEAD pointer back to point to the `master` branch, and it restored the files in your working directory back to the snapshot that `master` points to. This also means the changes you make from this point forward will diverge from an older version of the project. It essentially rewinds the work you've done in your `testing` branch so you can go in a different direction.

Switching branches changes files in your working directory

It's important to note that when you switch branches in Git, files in your working directory will change. If you switch to an older branch, your working directory will be restored to what it looked like the last time you committed on that branch. If Git cannot do it cleanly, it will not let you switch at all.

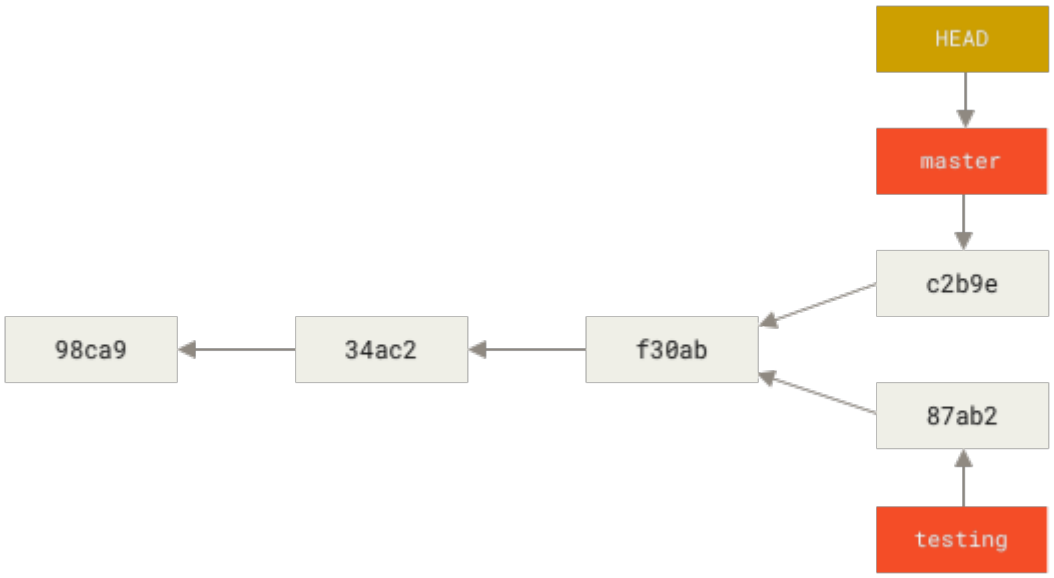


Figure 38. Divergent history.

Because a branch in Git is actually a simple file that contains the SHA checksum of the commit it points to, branches are cheap to create and destroy. Creating a new branch is as quick and simple as writing the checksum and a newline to a file.

This is in sharp contrast to the way most older VCS tools branch, which involves copying all of the project's files into a second directory. This can take several seconds or even minutes, depending on the size of the project, whereas in Git the process is always instantaneous. Also, because we're recording the parents when we commit, finding a proper merge base for merging is automatically done for us and is generally very easy to do. These features help encourage developers to create and use branches often.

Let's see why you should do so.

Basic Branching and Merging

When the work on your branch is complete, the simplest way to share it is to merge it with master.

Basic Merging

Suppose you've decided that your work is complete and ready to be merged into your `master` branch. In order to do that, you'll merge your branch into `master`. In Team Explorer **Branches**,

checkout master and click **Merge**.

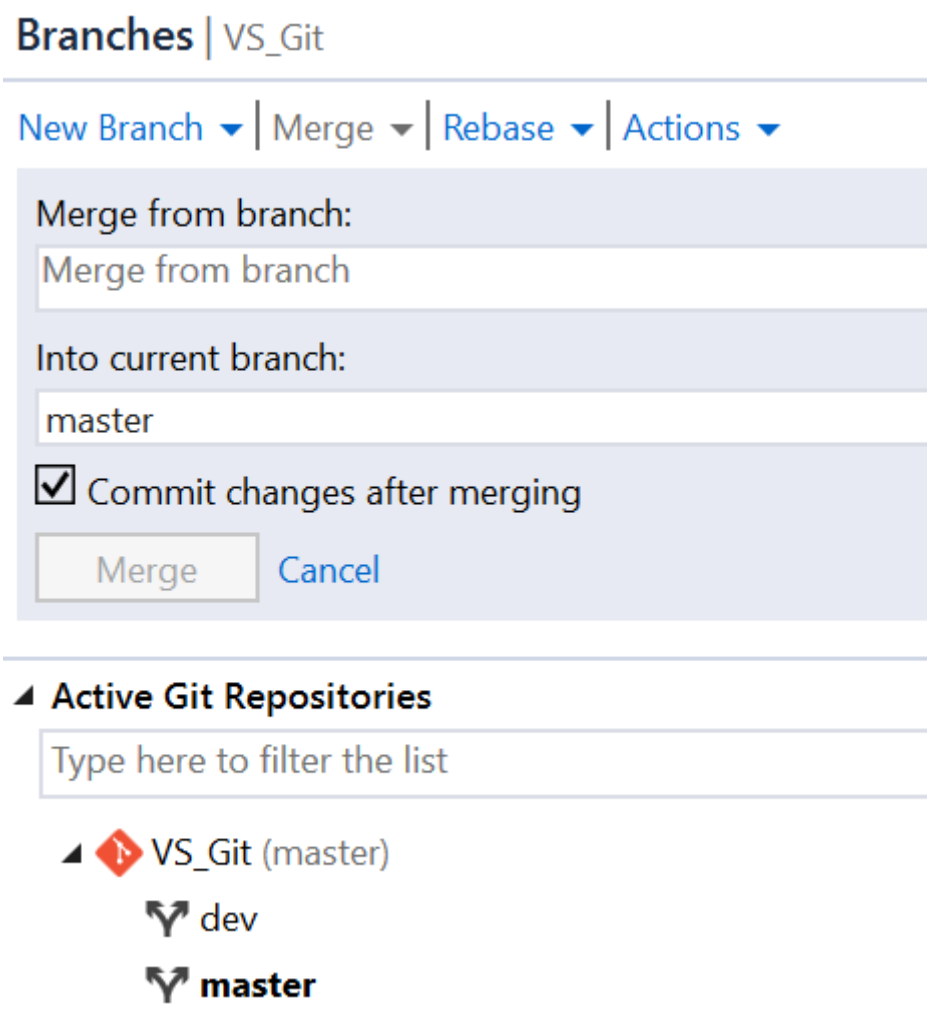


Figure 39. Simple Merge.

So long as the master branch has not diverged from your branch, this will create a "fast-forward" commit. When you try to merge one commit with a commit that can be reached by following the first commit's history, Git simplifies things by moving the pointer forward because there is no divergent work to merge together — this is called a "fast-forward."

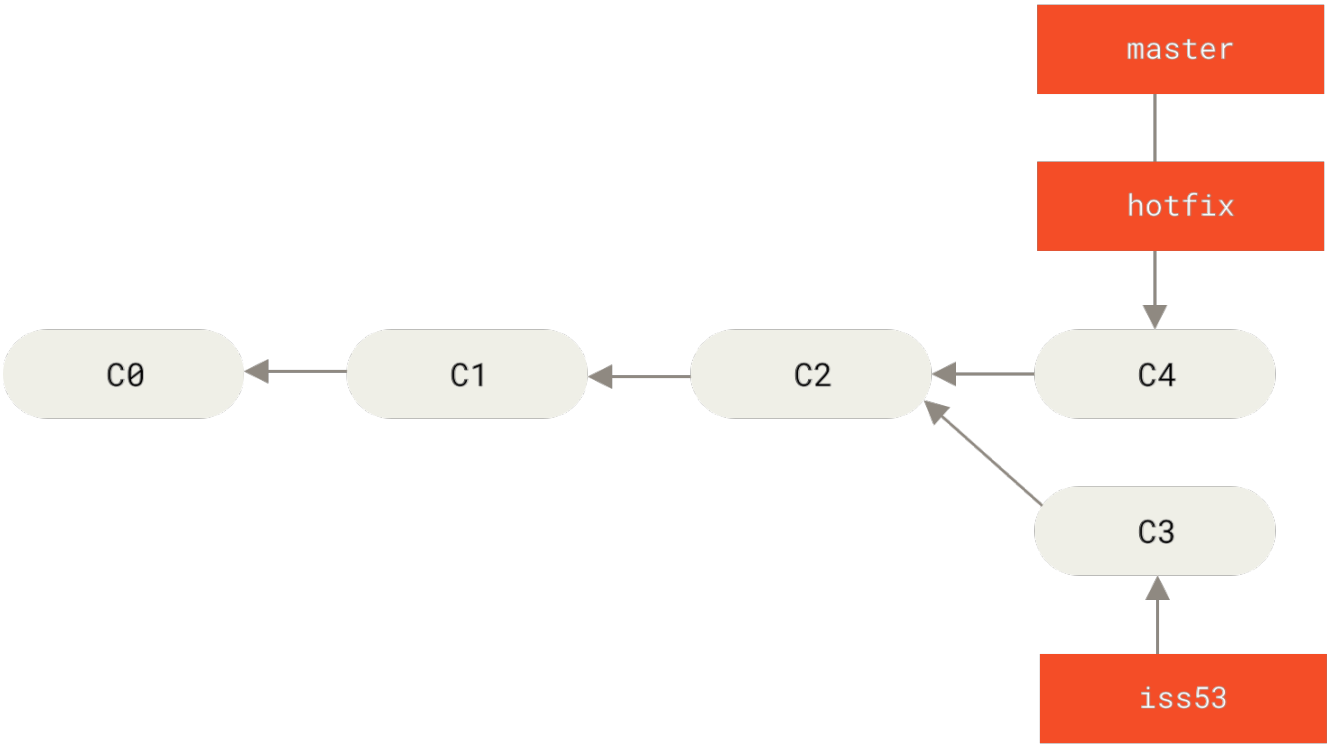


Figure 40. `master` is fast-forwarded to `hotfix`

If your development history has diverged from some older point, the commit on the branch you're on isn't a direct ancestor of the branch you're merging in, and Git has to do some work. In this case, Git does a simple three-way merge, using the two snapshots pointed to by the branch tips and the common ancestor of the two.

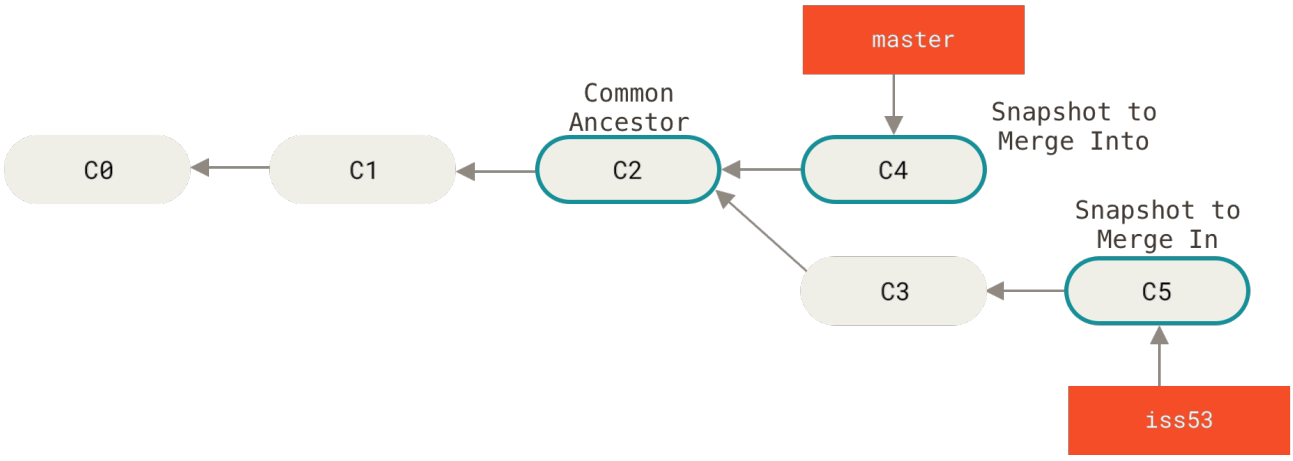


Figure 41. Three snapshots used in a typical merge

Instead of just moving the branch pointer forward, Git creates a new snapshot that results from this three-way merge and automatically creates a new commit that points to it. This is referred to as a merge commit, and is special in that it has more than one parent.

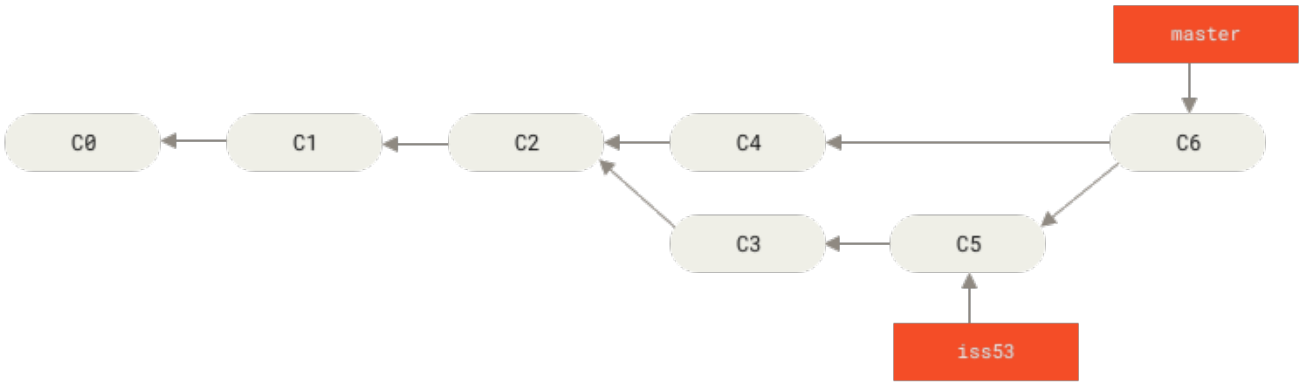


Figure 42. A merge commit.

Once your work is merged in, you have no further need for the branch. You can close the issue in the tracking system and delete the branch.

To delete the branch, make sure it isn't checked out, then right-click on the branch in the Team Explorer **Branches** view and click delete.

[] | *vimeo*

Delete Branch.

This option can also delete branches that have yet not been merged. If you have unpublished changes, Visual Studio will ask for confirmation in case you lose work.

Basic Merge Conflicts

Occasionally, this process doesn't go smoothly. If you changed the same part of the same file differently in the two branches you're merging, Git won't be able to merge them cleanly. A conflict notification will appear. Click **Conflicts** to start to resolve the conflicts.

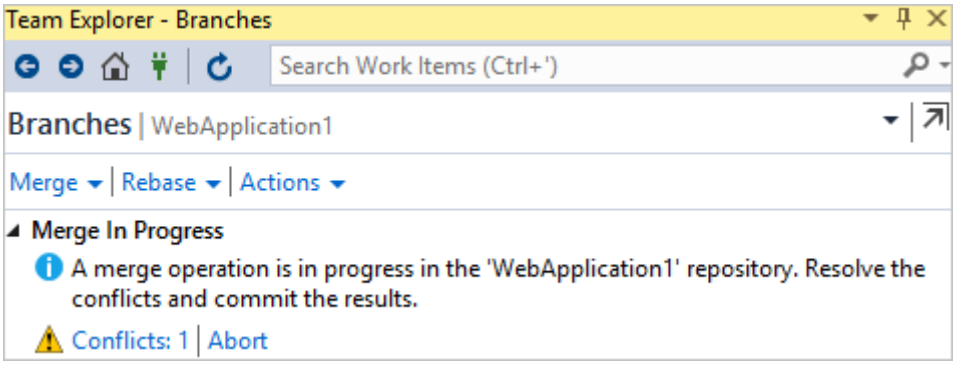


Figure 43. Merge Prompt.

This will bring up a list of unresolved merge conflicts. You can accept the branch you are merging from with the **Take Source** button or keep the chnages in the branch you are merging into with **Keep Target**, or resolve the conflicts manually by selectting **Merge**. use the checkboxes next to the modified lines to select between remote and local changes entirely, or edit the results directly in the

Result editor. When done, click **Accept Merge**, and repeat the process for all conflcting files. Once all conflicts are resolved yo will be able to commit the changes to create the merge commit.

[] | *vimeo*

Resolve Merge Conflicts.

Branch Management

Now that you've created, merged, and deleted some branches, let's look at some branch-management tools that will come in handy when you begin using branches all the time.

The Team Explorer **Branches** view does more than just create and delete branches. Opening the view will give you a simple listing of your current branches:

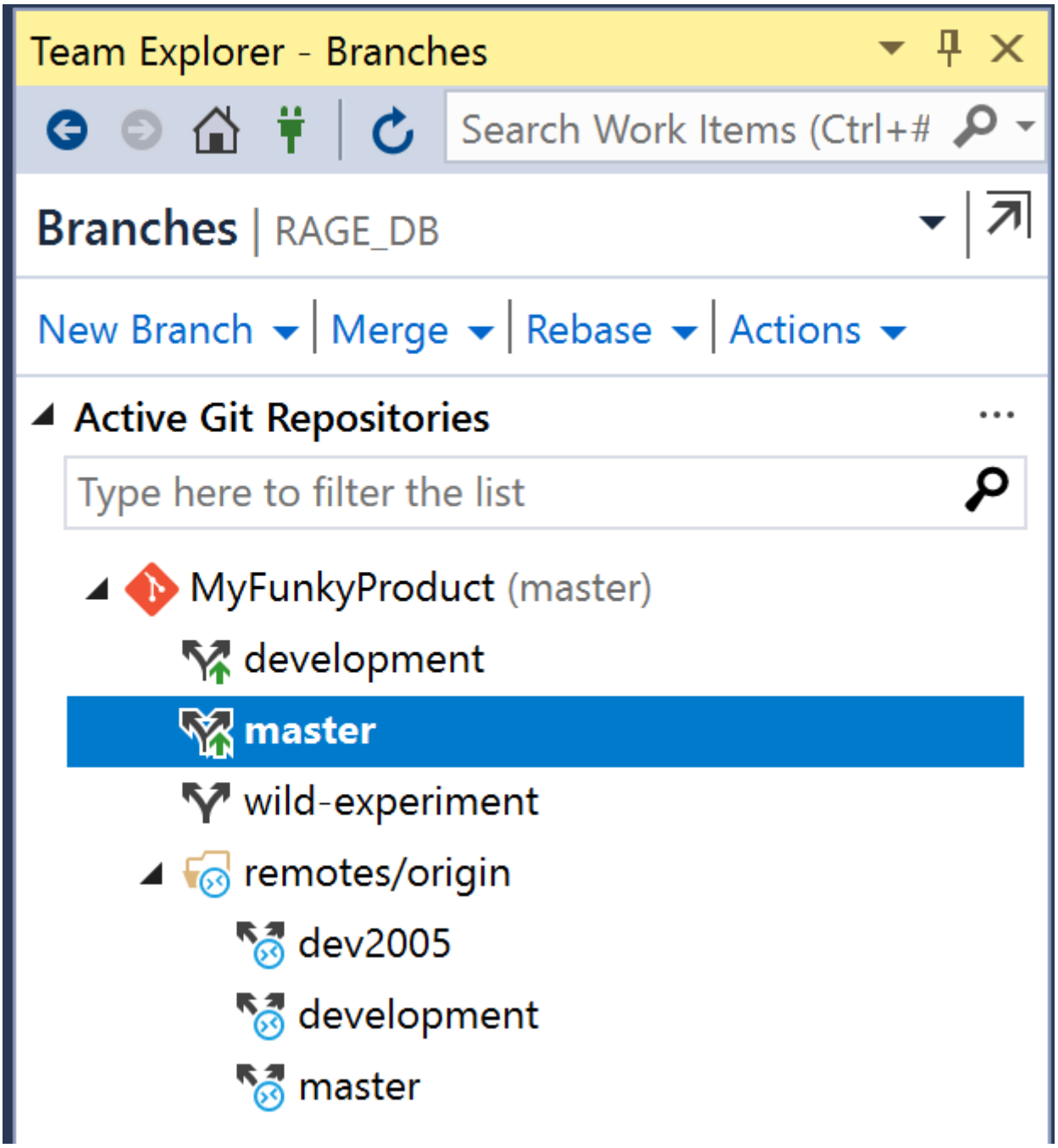


Figure 44. View Branches.

Notice `master` branch is in bold: this indicates the branch that you currently have checked out (i.e., the branch that `HEAD` points to). This means that if you commit at this point, the `master` branch will be moved forward with your new work.

You can also filter the view to show branches names matching a particular pattern.

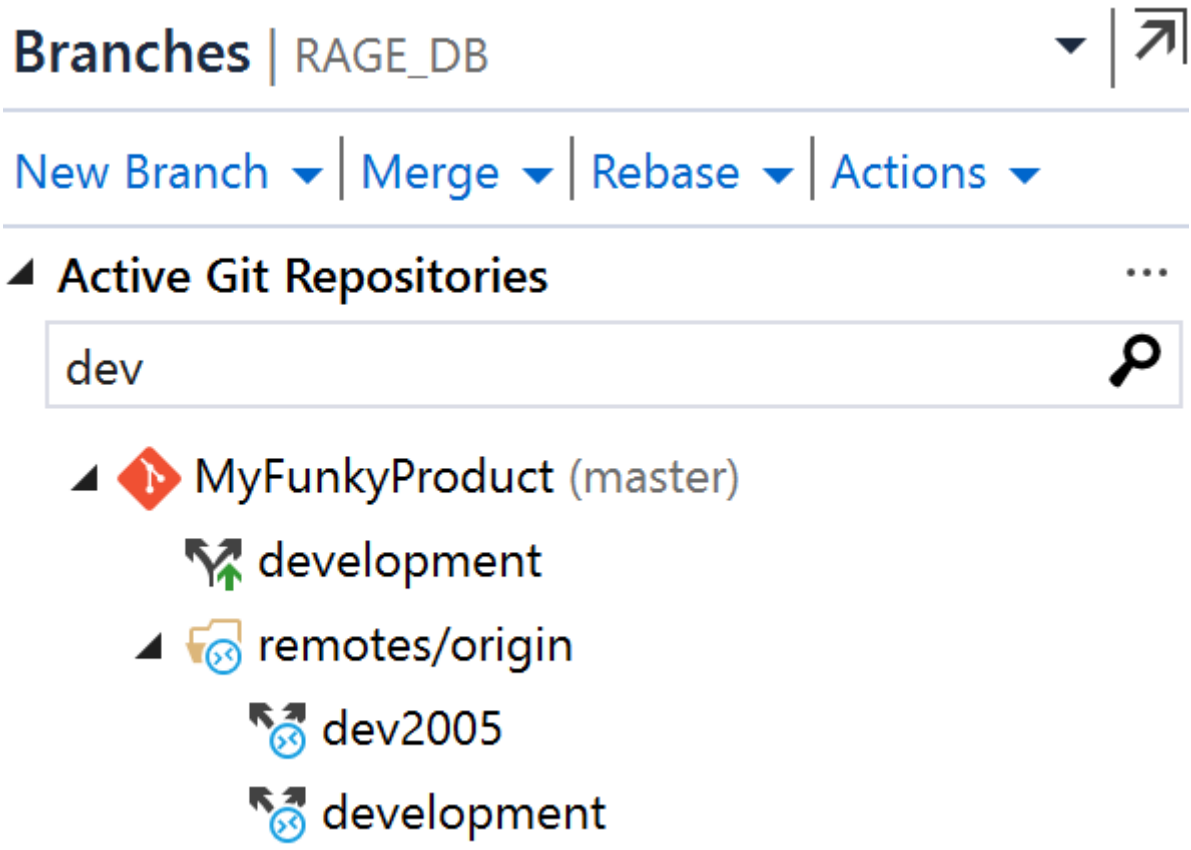


Figure 45. View Filtered Branches.

You can delete unmerged branches that you don't want to keep, and branches on remote repositories.

Visual Studio will first confirm if it is ok to continue before deleting a branch containing unmerged work or a remote branch.

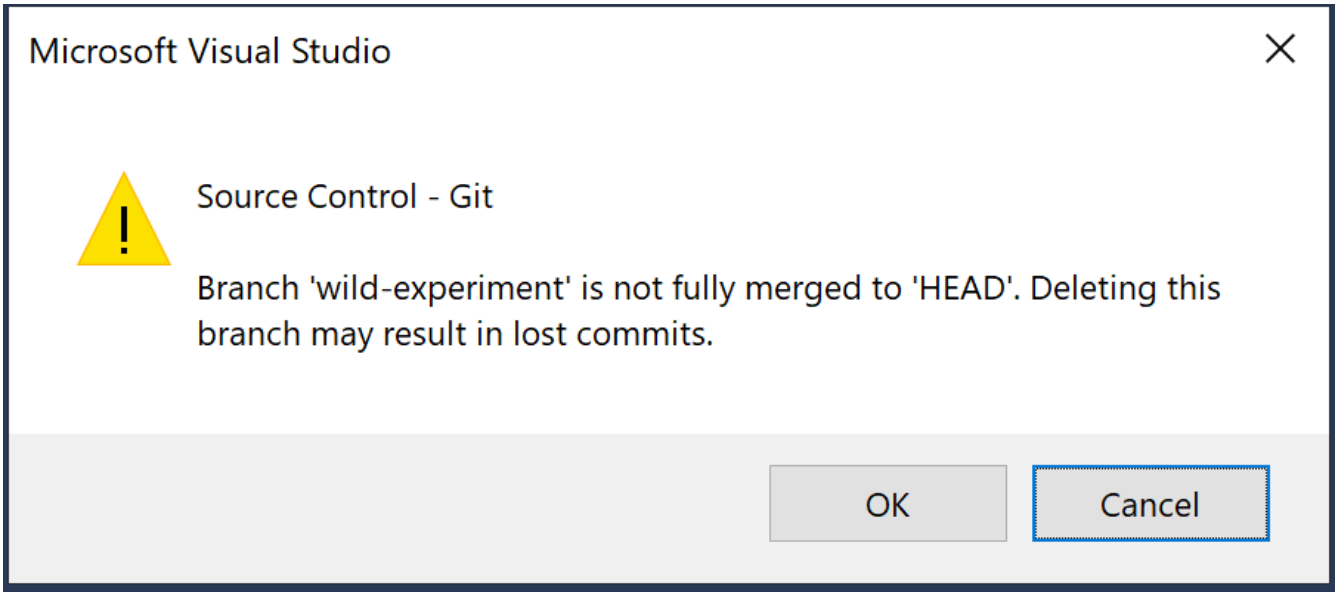


Figure 46. Delete Unmerged Branch.

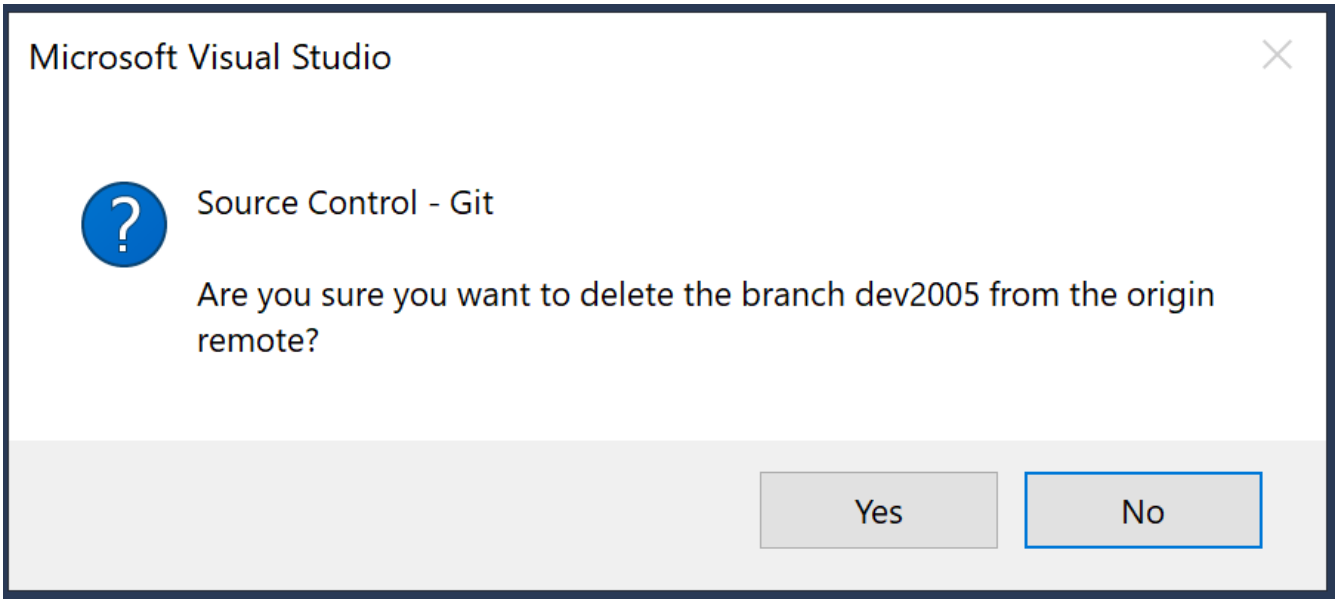


Figure 47. Delete Remote Branch.

Branching Workflows

Now that you have the basics of branching and merging down, what can or should you do with them? In this section, we'll cover some common workflows that this lightweight branching makes possible, so you can decide if you would like to incorporate them into your own development cycle.

Long-Running Branches

Because Git uses a simple three-way merge, merging from one branch into another multiple times over a long period is generally easy to do. This means you can have several branches that are always open and that you use for different stages of your development cycle; you can merge regularly from some of them into others.

Many Git developers have a workflow that embraces this approach, such as having only code that is entirely stable in their `master` branch — possibly only code that has been or will be released. They

have another parallel branch named `develop` or `next` that they work from or use to test stability — it isn't necessarily always stable, but whenever it gets to a stable state, it can be merged into `master`. It's used to pull in topic branches (short-lived branches, aka ticket or issue branch) when they're ready, to make sure they pass all the tests and don't introduce bugs.

In reality, we're talking about pointers moving up the line of commits you're making. The stable branches are farther down the line in your commit history, and the bleeding-edge branches are farther up the history.

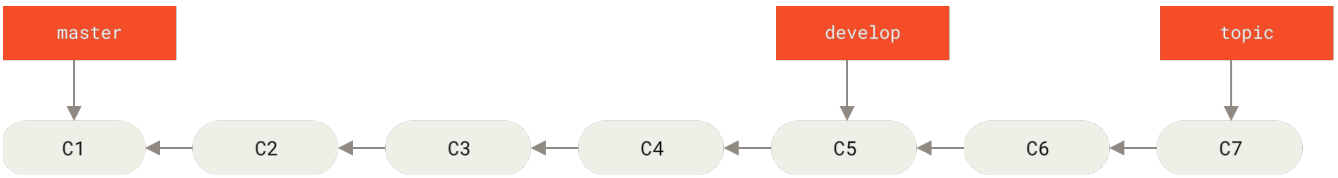


Figure 48. A linear view of progressive-stability branching

It's generally easier to think about them as work silos, where sets of commits graduate to a more stable silo when they're fully tested.

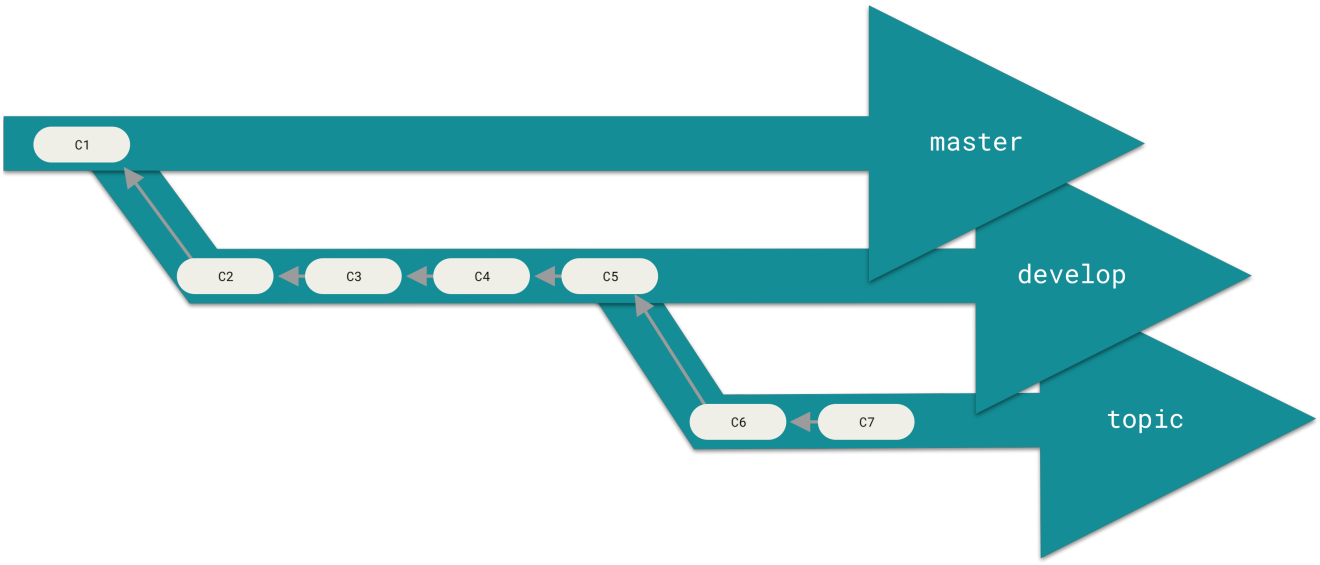


Figure 49. A "silo" view of progressive-stability branching

You can keep doing this for several levels of stability. Some larger projects also have a `proposed` or `pu` (proposed updates) branch that has integrated branches that may not be ready to go into the `next` or `master` branch. The idea is that your branches are at various levels of stability; when they reach a more stable level, they're merged into the branch above them. Again, having multiple long-running branches isn't necessary, but it's often helpful, especially when you're dealing with very large or complex projects.

Topic Branches

Topic branches, however, are useful in projects of any size. A topic branch is a short-lived branch that you create and use for a single particular feature or related work. This is something you've likely never done with a VCS before because it's generally too expensive to create and merge branches. But in Git it's common to create, work on, merge, and delete branches several times a day.

This technique allows you to context-switch quickly and completely — because your work is

separated into silos where all the changes in that branch have to do with that topic, it's easier to see what has happened during code review and such. You can keep the changes there for minutes, days, or months, and merge them in when they're ready, regardless of the order in which they were created or worked on.

Consider an example of doing some work (on `master`), branching off for an issue (`iss91`), working on it for a bit, branching off the second branch to try another way of handling the same thing (`iss91v2`), going back to your `master` branch and working there for a while, and then branching off there to do some work that you're not sure is a good idea (`dumbidea` branch). Your commit history will look something like this:

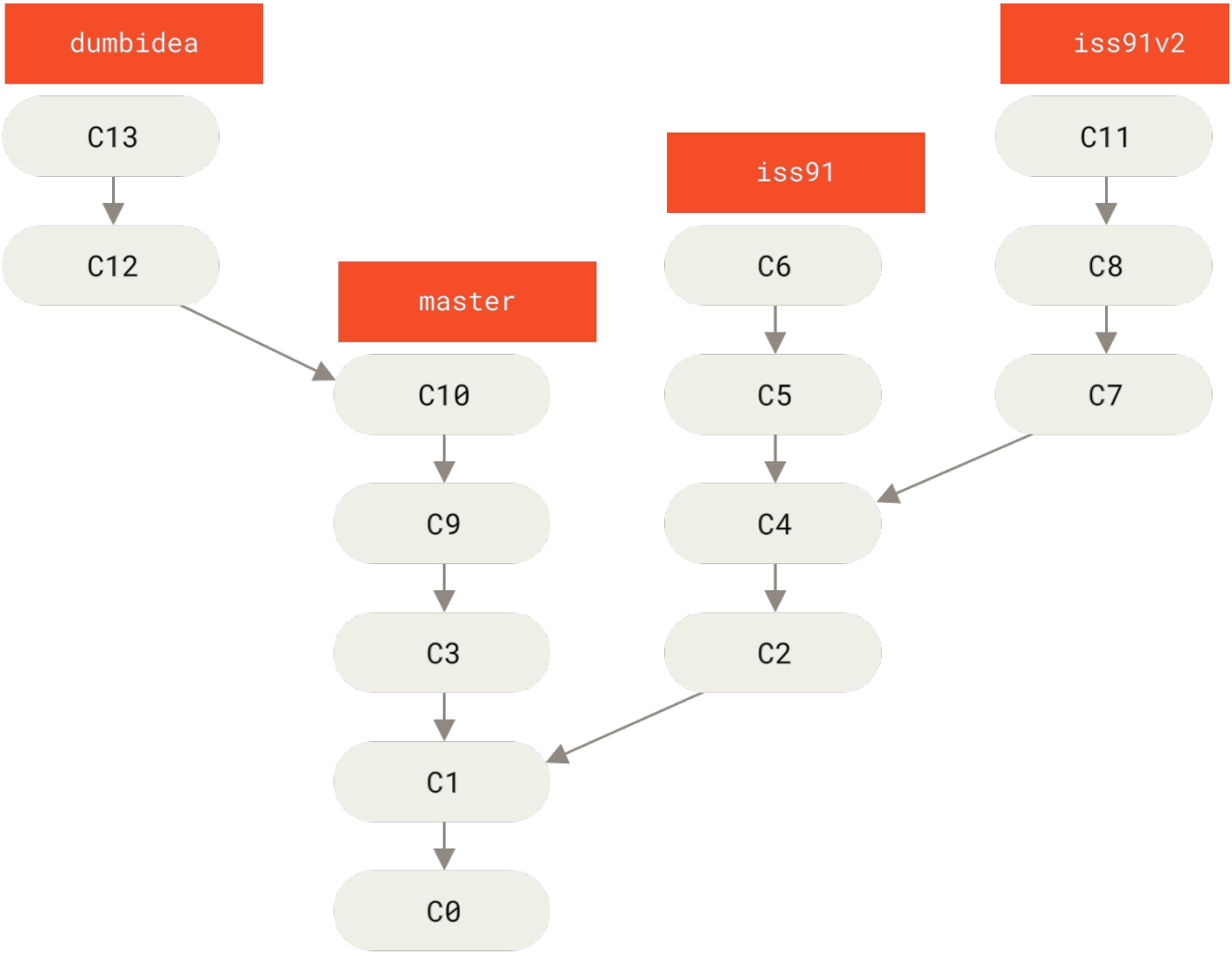


Figure 50. Multiple topic branches

Now, let's say you decide you like the second solution to your issue best (`iss91v2`); and you showed the `dumbidea` branch to your coworkers, and it turns out to be genius. You can throw away the original `iss91` branch (losing commits `C5` and `C6`) and merge in the other two. Your history then looks like this:

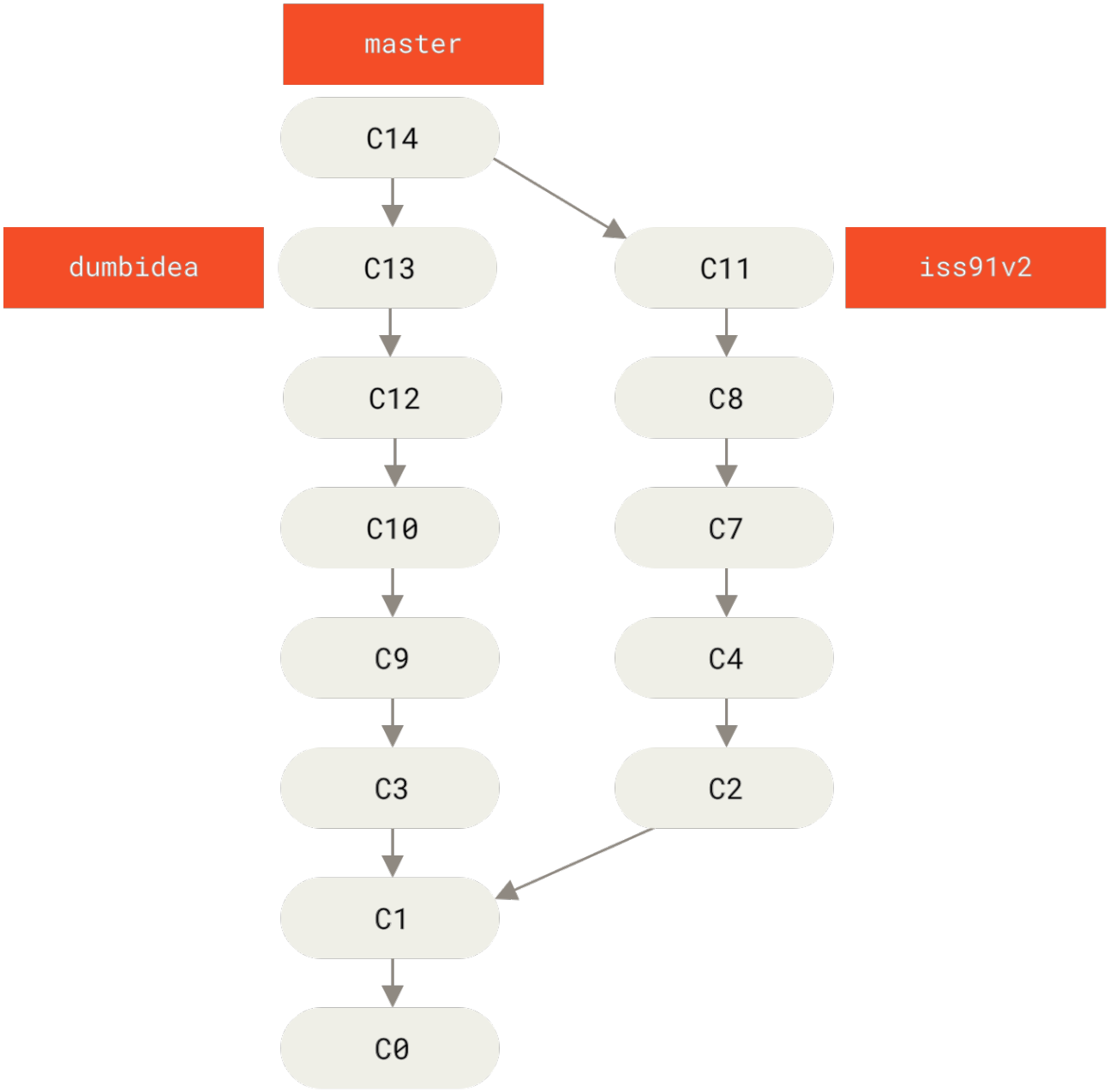


Figure 51. History after merging `dumbidea` *and* `iss91v2`

There is more detail about the various possible workflows for Git projects at Pro Git Chapter 5, so before you decide which branching scheme your next project will use, be sure to read that chapter.

It's important to remember when you're doing all this that these branches are completely local. When you're branching and merging, everything is being done only in your Git repository — there is no communication with the server.

Remote Branches

Remote-tracking branches are references to the state of remote branches. They're local references that you can't move; Git moves them for you whenever you do any network communication, to make sure they accurately represent the state of the remote repository. Think of them as bookmarks, to remind you where the branches in your remote repositories were the last time you connected to them.

If you wanted to see what the `master` branch on your `origin` remote looked like as of the last time you communicated with it, you would check the *master` branch under the 'origin* remote. If you were working on an issue with a partner and they pushed up an `iss53` branch, you might have your own local `iss53` branch, and the branch on the server would be represented by the remote-tracking `iss53` under the `origin` remote.

This may be a bit confusing, so let's look at an example. Let's say you have a Git server on your network at `git.ourcompany.com`. If you clone from this, Git's `clone` command automatically names it `origin` for you, pulls down all its data, creates a pointer to where its `master` branch is, and names it `origin/master` locally. Git also gives you your own local `master` branch starting at the same place as origin's `master` branch, so you have something to work from.

"origin" is not special

While "master" is the default name for a starting branch when you create a repository, and "origin" is the default name for a remote when you clone a repository, neither have any special meaning in Git, and you can have a starting branch and remote with different names.

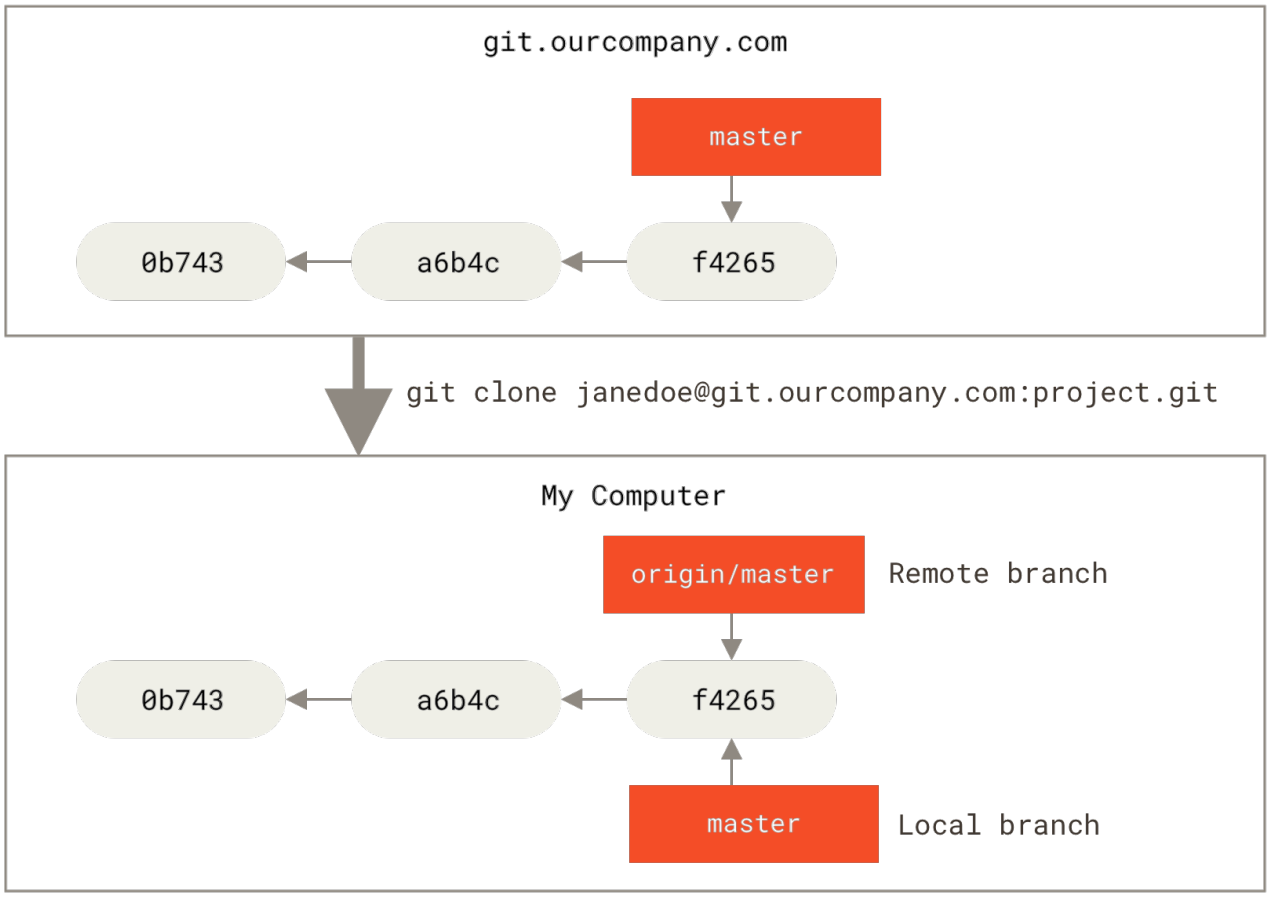


Figure 52. Server and local repositories after cloning

If you do some work on your local `master` branch, and, in the meantime, someone else pushes to `git.ourcompany.com` and updates its `master` branch, then your histories move forward differently. Also, as long as you stay out of contact with your `origin` server, your `origin/master` pointer doesn't move.

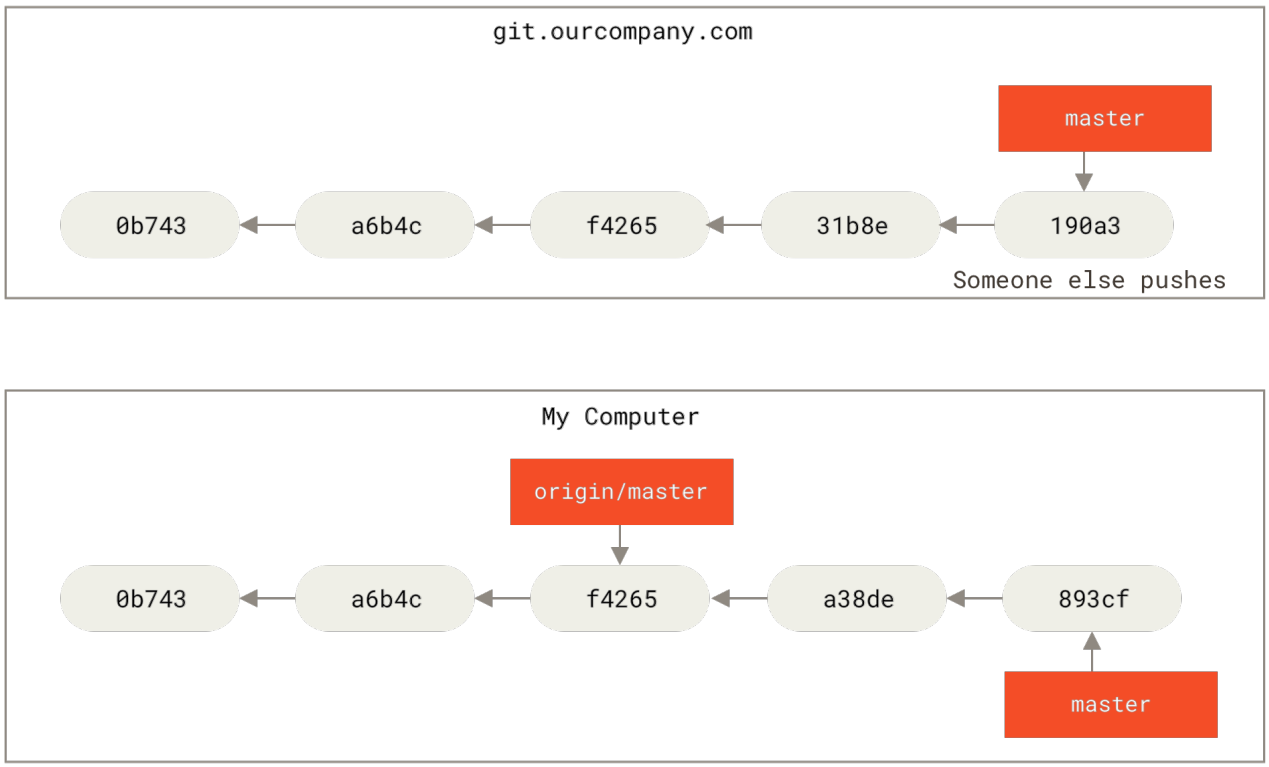


Figure 53. Local and remote work can diverge

To synchronize your work with a given remote, you **Fetch** changes from it. This command looks up the remote server (in this case, it's `git.ourcompany.com`), fetches any data from it that you don't yet have, and updates your local database, moving your the remote tracking pointer to its new, more up-to-date position.

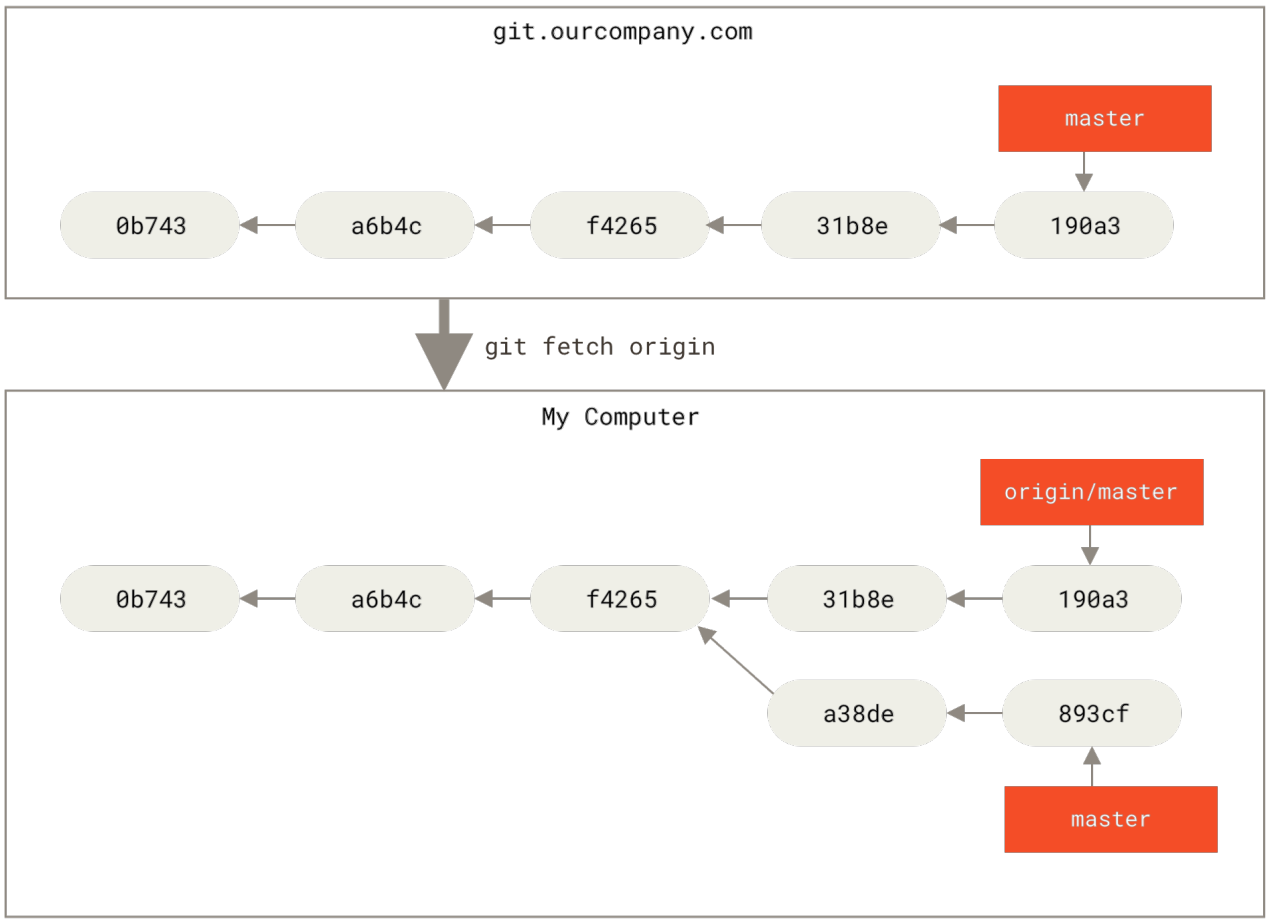


Figure 54. `git fetch` updates your remote-tracking branches

To demonstrate having multiple remote servers and what remote branches for those remote projects look like, let's assume you have another internal Git server that is used only for development by one of your sprint teams. This server is at `git.team1.ourcompany.com`. You can add it as a new remote reference to the project you're currently working on in the repository settings as we covered in Git Basics. Name this remote `teamone`, which will be your shortname for that whole URL.

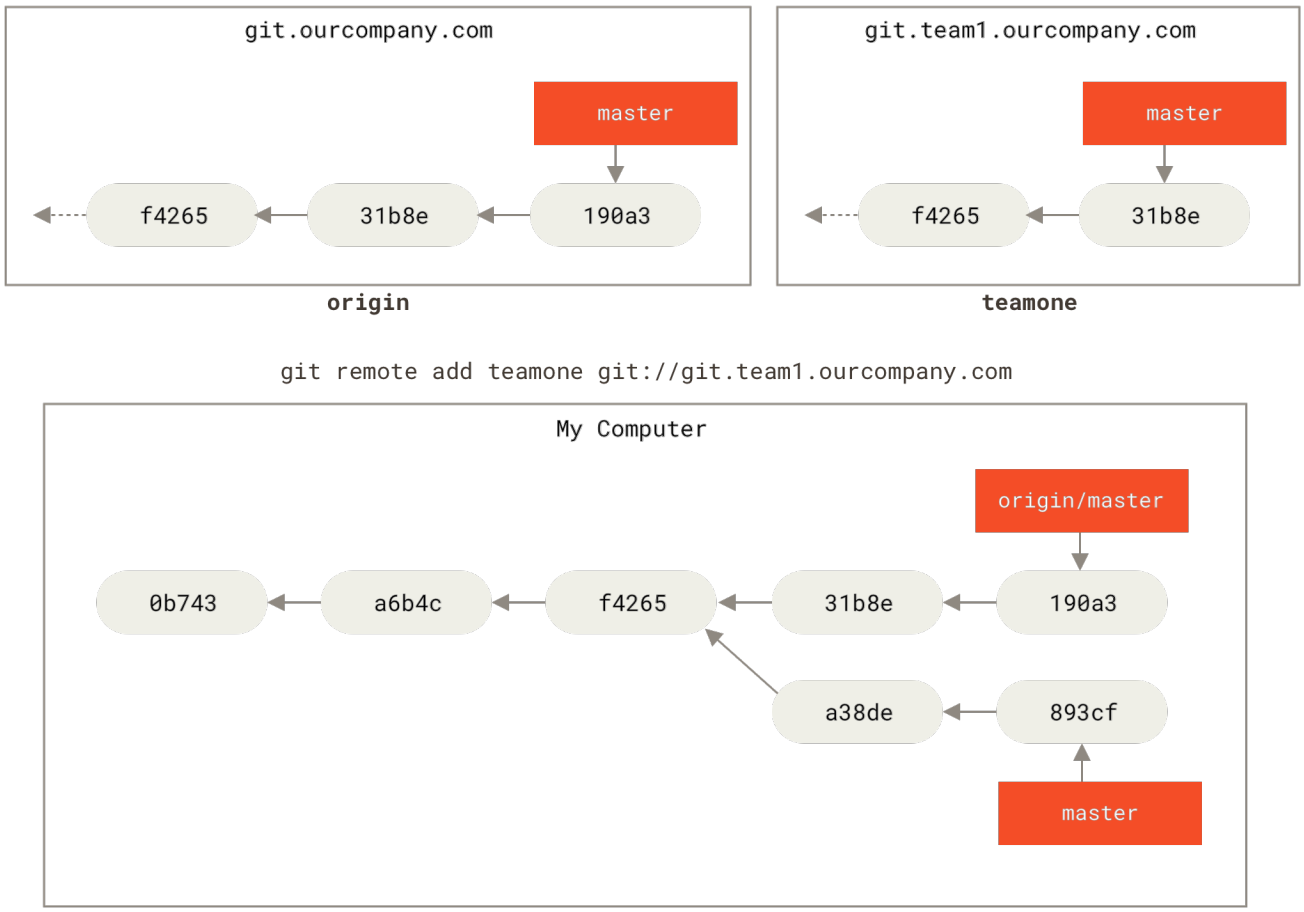


Figure 55. Adding another server as a remote

Now, you can run **Fetch** from `teamone` to fetch everything the remote `teamone` server has that you don't have yet. Because that server has a subset of the data your `origin` server has right now, Git fetches no data but sets a remote-tracking branch `master' under the `teamone` remote to point to the commit that `teamone` has as its `master` branch.

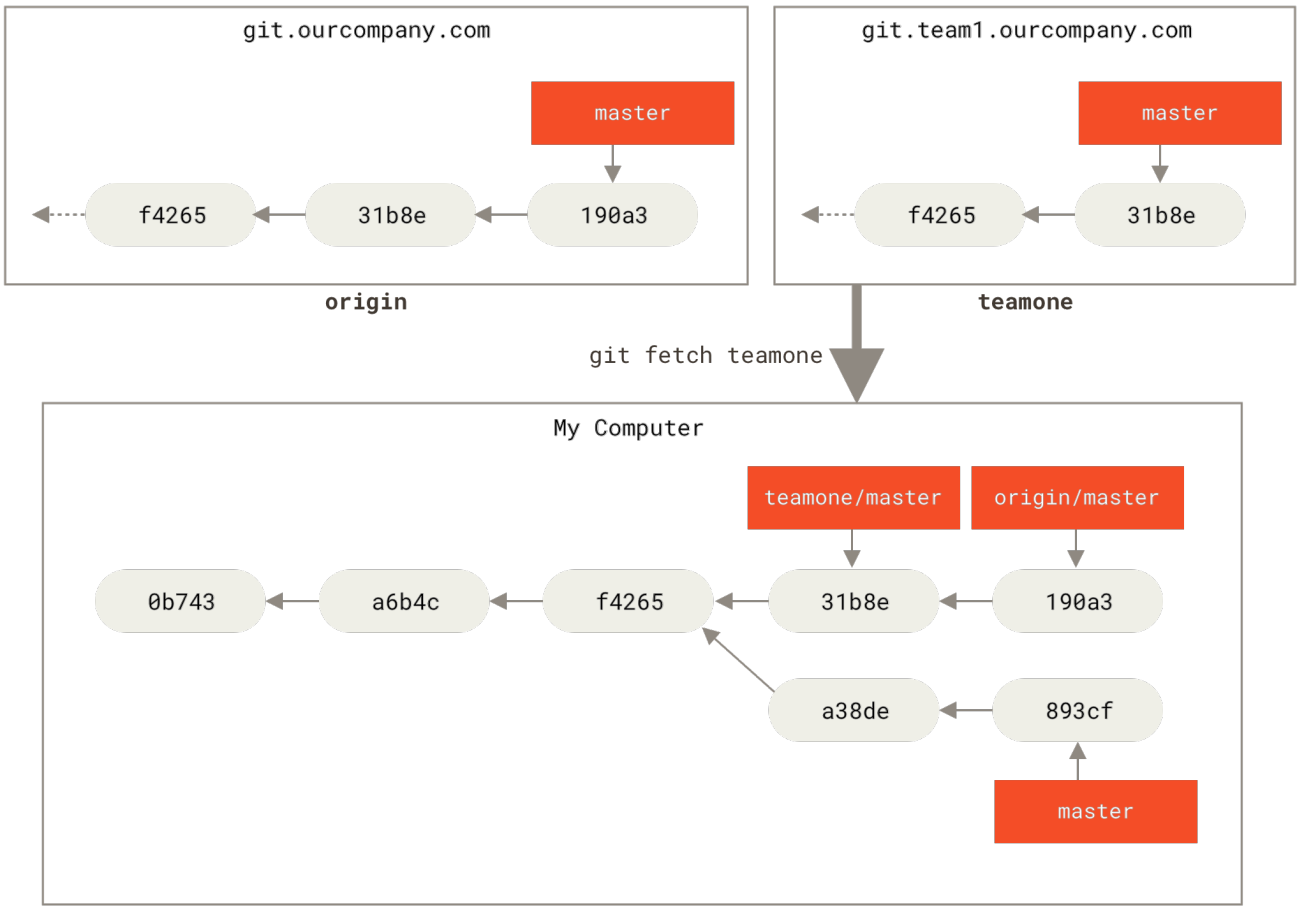


Figure 56. Remote-tracking branch for `teamone/master`

Pushing

When you want to share a branch with the world, you need to push it up to a remote to which you have write access. Your local branches aren't automatically synchronized to the remotes you write to — you have to explicitly push the branches you want to share. That way, you can use private branches for work you don't want to share, and push up only the topic branches you want to collaborate on.

If you have a branch named `serverfix` that you want to work on with others, you can push it up the same way you pushed your first branch. :

The next time one of your collaborators fetches from the server, they will get a reference to where the server's version of `serverfix` is under the *serverfix* branch undeer the remote `origin`

It's important to note that when you do a fetch that brings down new remote-tracking branches, you don't automatically have local, editable copies of them. In other words, in this case, you don't have a new `serverfix` branch — you have only an `origin/serverfix` pointer that you can't modify.

You can merge this work into your current working branch, or checkout the branch to work on it locally.

Tracking Branches

Checking out a local branch from a remote-tracking branch automatically creates what is called a

“tracking branch” (and the branch it tracks is called an “upstream branch”). Tracking branches are local branches that have a direct relationship to a remote branch. If you’re on a tracking branch and do a **Pull**, Git automatically knows which server to fetch from and which branch to merge in.

When you clone a repository, it generally automatically creates a `master` branch that tracks `origin/master`. However, you can set up other tracking branches if you wish — ones that track branches on other remotes, or don’t track the `master` branch.

In the most common case is where the branch name you’re trying to checkout (a) doesn’t exist and (b) exactly matches a name on only one remote, Git will create a tracking branch for you

Pulling

While **Fetch** will fetch all the changes on the server that you don’t have yet, it will not modify your working directory at all. It will simply get the data for you and let you merge it yourself. However, you can do a **Pull**, which is essentially a **Fetch** immediately followed by a **Merge** in most cases. If you have a tracking branch set up as demonstrated in the last section, either by explicitly setting it or by having it created for you by the `clone` or `checkout` commands, **Pull** will look up what server and branch your current branch is tracking, fetch from that server and then try to merge in that remote branch.

Generally it’s better to simply do the **Fetch** and **Merge** explicitly as the magic of **Pull** can often be confusing.

Deleting Remote Branches

Suppose you’re done with a remote branch — say you and your collaborators are finished with a feature and have merged it into your remote’s `master` branch (or whatever branch your stable codeline is in). You can delete a remote branches the same way as local branches, by finding the branch in Team Explorer’s **Branches** view under the relevant remote, right-clicking and selecting **Delete**

Basically all this does is remove the pointer from the server. The Git server will generally keep the data there for a while until a garbage collection runs, so if it was accidentally deleted, it’s often easy to recover.

Rebasing

In Git, there are two main ways to integrate changes from one branch into another: the `merge` and the `rebase`. In this section you’ll learn what rebasing is, how to do it, why it’s a pretty amazing tool, and in what cases you won’t want to use it.

The Basic Rebase

This simple history shows diverging work and original branches.

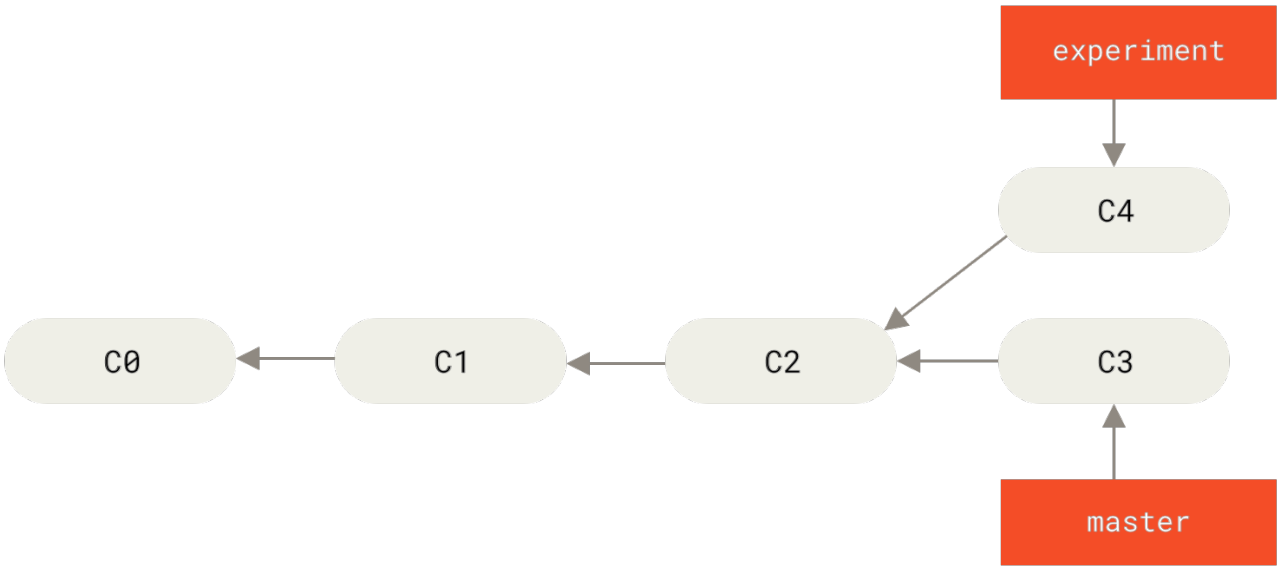


Figure 57. Simple divergent history

The easiest way to integrate the branches, is with a merge. It performs a three-way merge between the two latest branch snapshots (`C3` and `C4`) and the most recent common ancestor of the two (`C2`), creating a new snapshot (and commit).

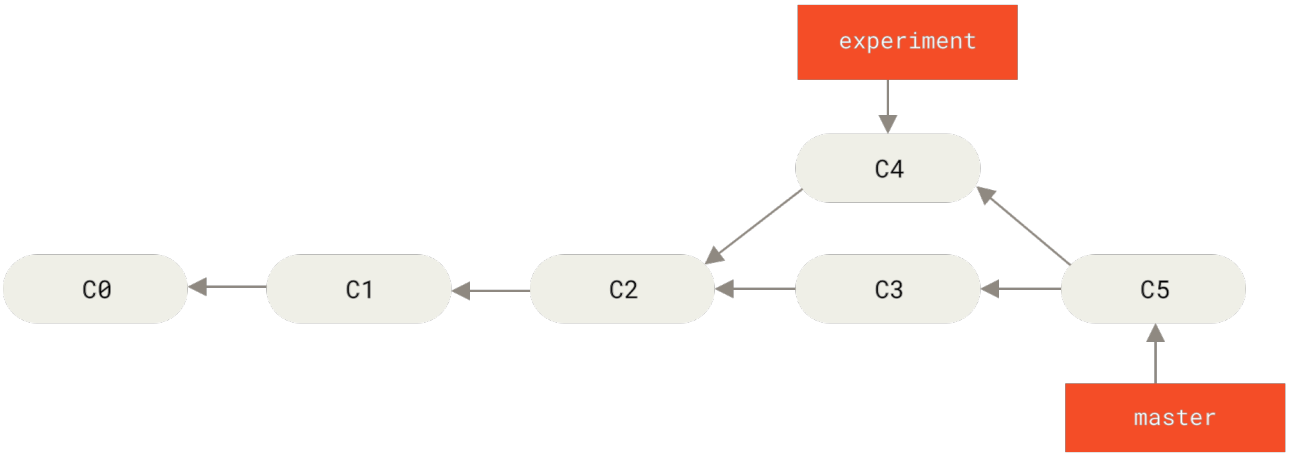


Figure 58. Merging to integrate diverged work history

However, this creates a non-linear merge commit *C5* joining two commits sub-streams. Another way is to take the patch of the change that was introduced in `C4` and reapply it on top of `C3`. In Git, this is called *rebasing*. With the `rebase` command, you can take all the changes that were committed on one branch and replay them on a different branch.

In Team Explorer, go to the **Branches** view and click **Rebase**. Choose the branch you want to rebase, which defaults to the current branch, and the branch your changes should be replayed on top of.

[] | *vimeo*

Rebasing.

This operation works by going to the common ancestor of the two branches (the one you're on and the one you're rebasing onto), getting the diff introduced by each commit of the branch you're on,

saving those diffs to temporary files, resetting the current branch to the same commit as the branch you are rebasing onto, and finally applying each change in turn.

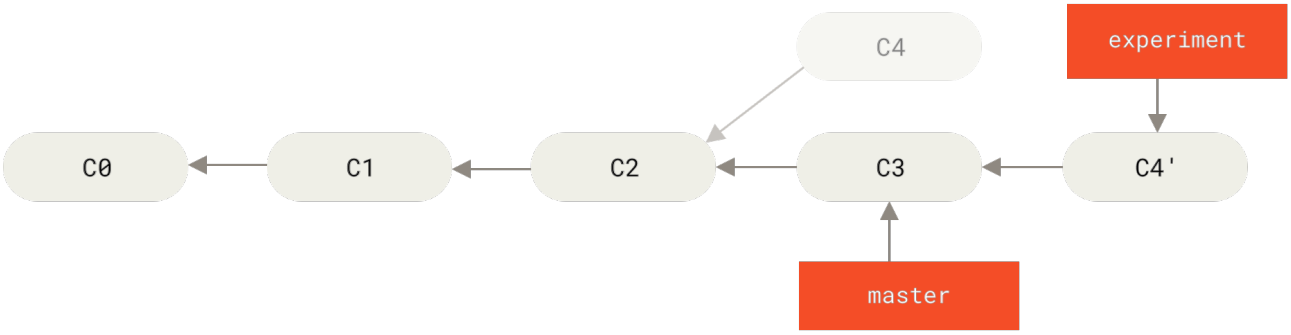


Figure 59. Rebasing the change introduced in `C4` onto `C3`

At this point, you can go back to the `master` branch and do a fast-forward merge.

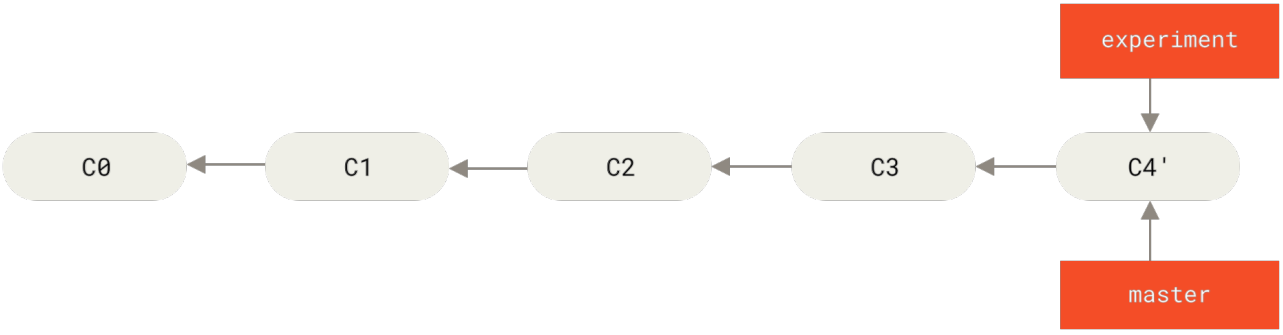


Figure 60. Fast-forwarding the `master` branch

Now, the snapshot pointed to by `C4'` is exactly the same as the one that was pointed to by `C5` in the merge example. There is no difference in the end product of the integration, but rebasing makes for a cleaner history. If you examine the log of a rebased branch, it looks like a linear history: it appears that all the work happened in series, even when it originally happened in parallel. This mirrors the behaviour of a VCS where the files for the changes in `C3` were locked when they were checked out: the changes in `C4` are applied to the version checked in by 'C3`, except that no locks are used.

Note that the content of snapshot pointed to by the final commit you end up with, whether it's the last of the rebased commits for a rebase or the final merge commit after a merge, is the same snapshot — it's only the history that is different. Rebasing replays changes from one line of work onto another in the order they were introduced, whereas merging takes the endpoints and merges them together.

The Perils of Rebasing

Ahh, but the bliss of rebasing isn't without its drawbacks, which can be summed up in a single line:

Do not rebase commits that exist outside your repository and that people may have based work on.

If you follow that guideline, you'll be fine. If you don't, people will hate you, and you'll be scorned

by friends and family.

When you rebase stuff, you're abandoning existing commits and creating new ones that are similar but different. If you push commits somewhere and others pull them down and base work on them, and then you rewrite those commits with `rebase` and push them up again, your collaborators will have to re-merge their work and things will get messy when you try to pull their work back into yours.

Let's look at an example of how rebasing work that you've made public can cause problems. Suppose you clone from a central server and then do some work off that. Your commit history looks like this:

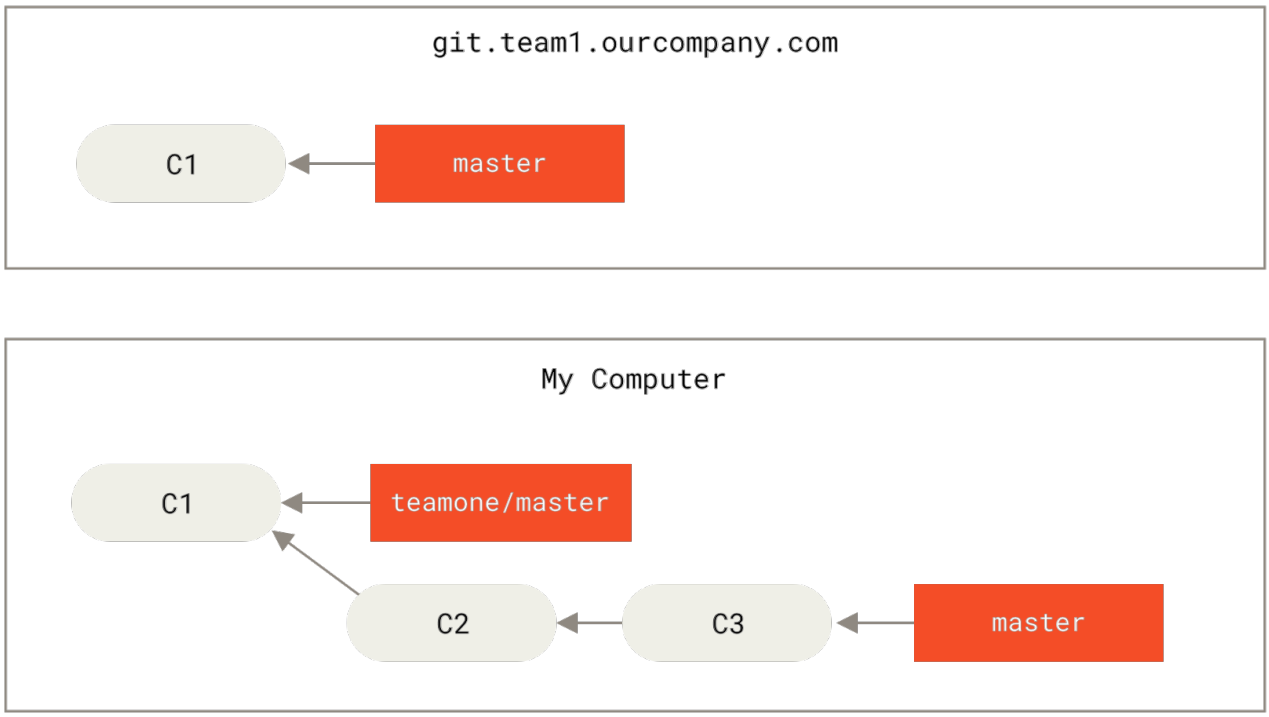


Figure 61. Clone a repository, and base some work on it

Now, someone else does more work that includes a merge, and pushes that work to the central server. You fetch it and merge the new remote branch into your work, making your history look something like this:

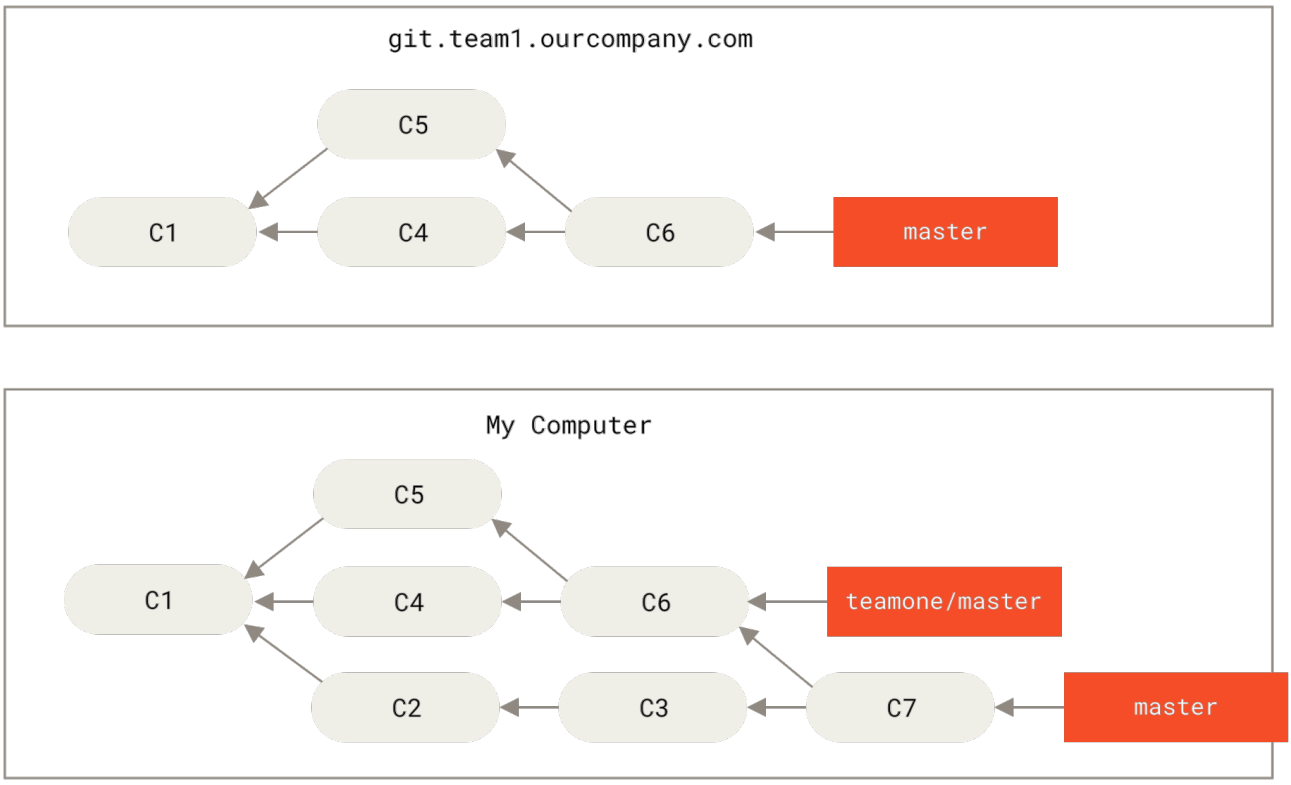


Figure 62. Fetch more commits, and merge them into your work

Next, the person who pushed the merged work decides to go back and rebase their work instead; they do a force **Push** to overwrite the history on the server. You then fetch from that server, bringing down the new commits.

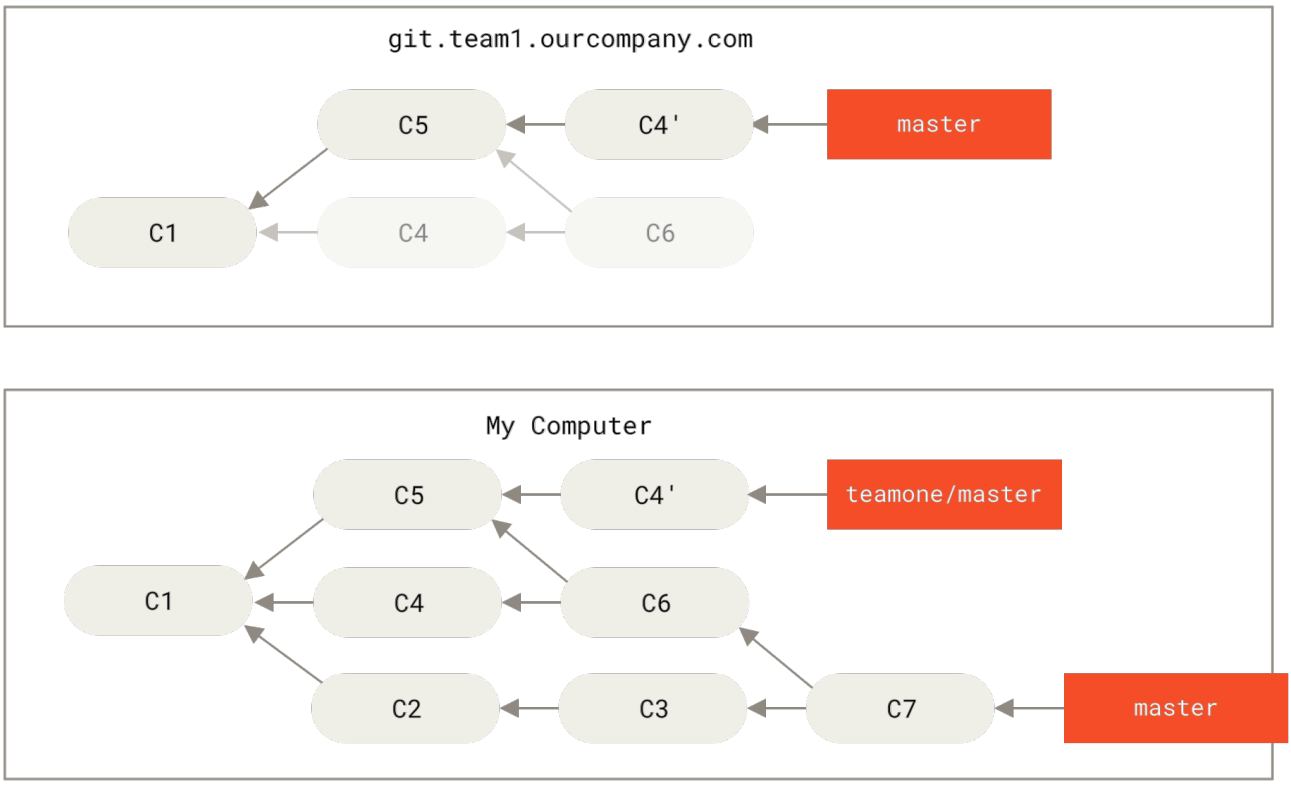


Figure 63. Someone pushes rebased commits, abandoning commits you've based your work on

Now you're both in a pickle. If you **Pull**, you'll create a merge commit which includes both lines of history, and your repository will look like this:

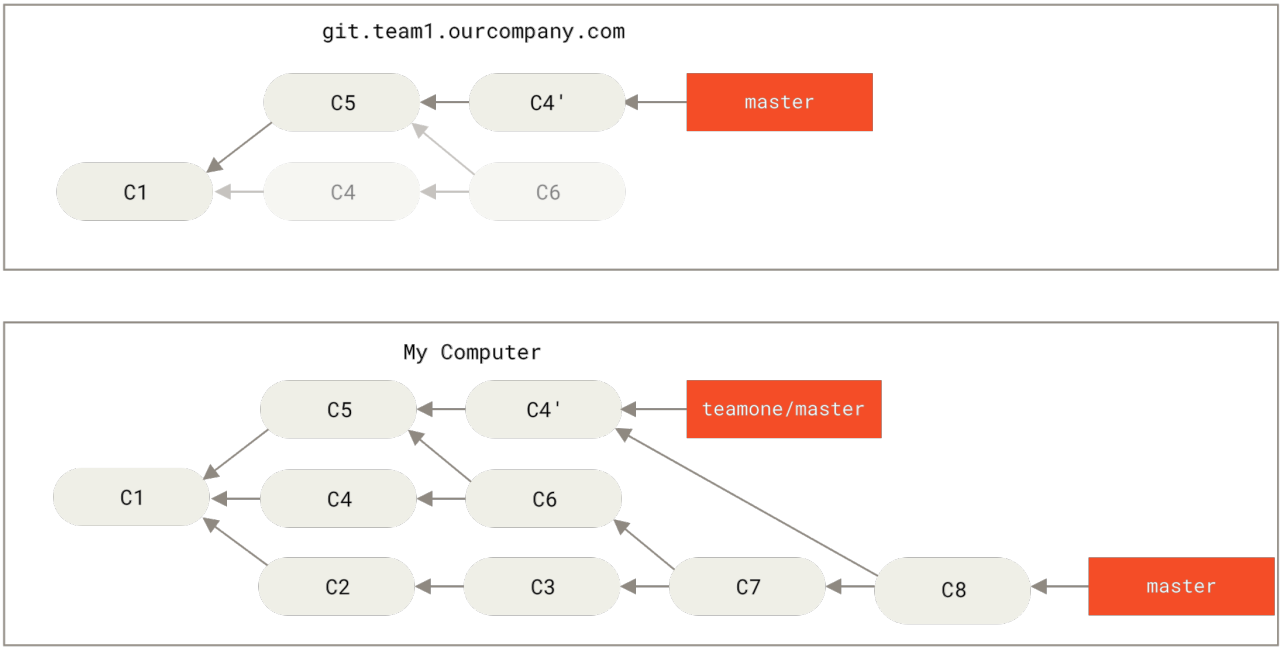


Figure 64. You merge in the same work again into a new merge commit

Your history will include two commits that have the same author, date, and message, which will be confusing. Furthermore, if you push this history back up to the server, you'll reintroduce all those rebased commits to the central server, which can further confuse people. It's pretty safe to assume that the other developer doesn't want `C4` and `C6` to be in the history; that's why they rebased in the first place.

Rebase When You Rebase

If you **do** find yourself in a situation like this, Git has some further magic that might help you out. If someone on your team force pushes changes that overwrite work that you've based work on, your challenge is to figure out what is yours and what they've rewritten.

It turns out that in addition to the commit SHA checksum, Git also calculates a checksum that is based just on the patch introduced with the commit. This is called a "patch-id".

If you pull down work that was rewritten and rebase it on top of the new commits from your partner, Git can often successfully figure out what is uniquely yours and apply them back on top of the new branch.

For instance, in the previous scenario, if instead of doing a merge when we're at Someone pushes rebased commits, abandoning commits you've based your work on we run `git rebase teamone/master`, Git will:

- Determine what work is unique to our branch (C2, C3, C4, C6, C7)
- Determine which are not merge commits (C2, C3, C4)
- Determine which have not been rewritten into the target branch (just C2 and C3, since C4 is the same patch as C4')

- Apply those commits to the top of `teamone/master`

So instead of the result we see in You merge in the same work again into a new merge commit, we would end up with something more like Rebase on top of force-pushed rebase work..

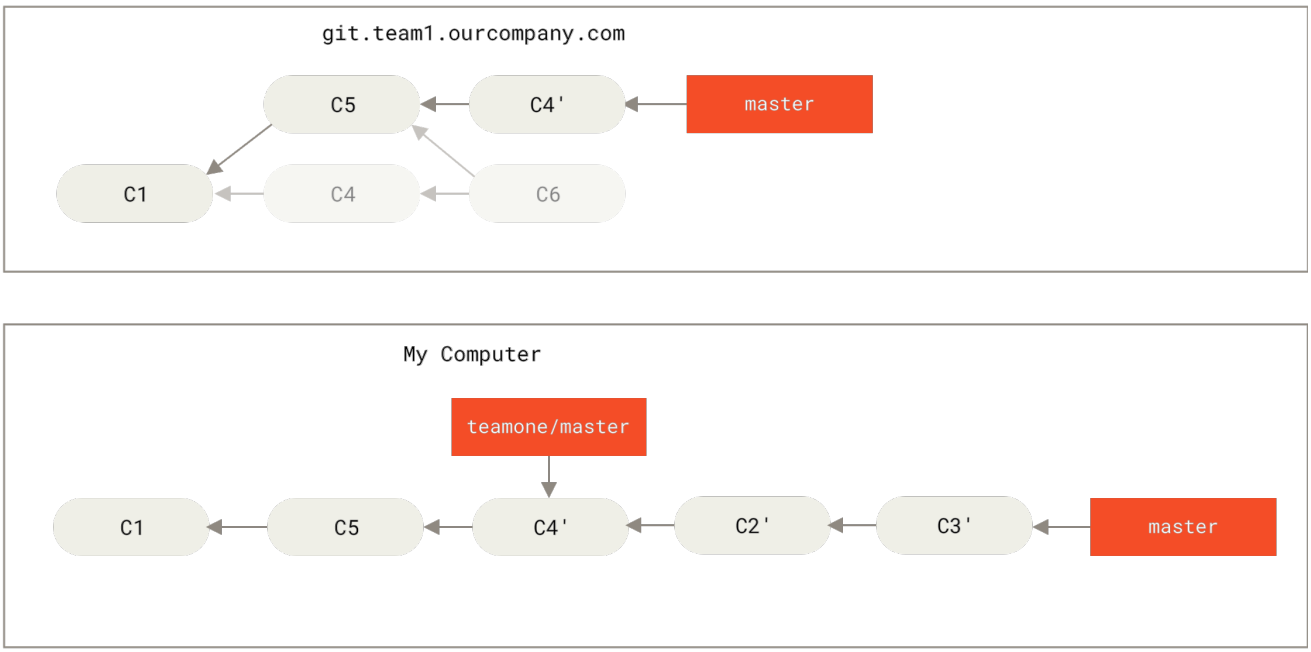


Figure 65. Rebase on top of force-pushed rebase work.

This only works if `C4` and `C4'` that your partner made are almost exactly the same patch. Otherwise the rebase won't be able to tell that it's a duplicate and will add another C4-like patch (which will probably fail to apply cleanly, since the changes would already be at least somewhat there).

You could do it manually with a **Fetch** followed by a rebase onto the remote repository master in this case.

Or you can configure your global settings so that **Pull** always rebases the local branch by setting **Rebase local branch when pulling** to `True`.

If you only ever rebase commits that have never left your own computer, you'll be just fine. If you rebase commits that have been pushed, but that no one else has based commits from, you'll also be fine. If you rebase commits that have already been pushed publicly, and people may have based work on those commits, then you may be in for some frustrating trouble, and the scorn of your teammates.

If you or a partner does find it necessary at some point, make sure everyone knows to run `git pull --rebase` to try to make the pain after it happens a little bit simpler.

Rebase vs. Merge

Now that you've seen rebasing and merging in action, you may be wondering which one is better. Before we can answer this, let's step back a bit and talk about what history means.

One point of view on this is that your repository's commit history is a *record of what actually happened.* It's a historical document, valuable in its own right, and shouldn't be tampered with.

From this angle, changing the commit history is almost blasphemous; you're *lying* about what actually transpired. So what if there was a messy series of merge commits? That's how it happened, and the repository should preserve that for posterity.

The opposing point of view is that the commit history is the *story of how your project was made*. You wouldn't publish the first draft of a book, and the manual for how to maintain your software deserves careful editing. This is the camp that uses tools like `rebase` and `filter-branch` to tell the story in the way that's best for future readers.

Now, to the question of whether merging or rebasing is better: hopefully you'll see that it's not that simple. Git is a powerful tool, and allows you to do many things to and with your history, but every team and every project is different. Now that you know how both of these things work, it's up to you to decide which one is best for your particular situation.

Bear in mind that even your original repository commit history will not - and should not - capture every edit and undo. To continue the book analogy, when sharing revisions with your editor you would not enumerate every single stage of your revisions.

In general the way to get the best of both worlds is to rebase local changes you've made but haven't shared yet before you push them in order to clean up your story, but never rebase anything you've pushed somewhere.

From a practical perspective, you are aiming to share a series of self-contained commits with changes that your collaborators will be able to integrate easily with their work - either individually or together as appropriate - using the git toolkit.

Summary

We've covered basic branching and merging in Git. You should feel comfortable creating and switching to new branches, switching between branches and merging local branches together. You should also be able to share your branches by pushing them to a shared server, working with others on shared branches and rebasing your branches before they are shared. Next, we'll cover what you'll need to run your own Git repository-hosting server.

Git Tools

By now, you've learned most of the day-to-day commands and workflows that you need to manage or maintain a Git repository for your source code control. You've accomplished the basic tasks of tracking and committing files, and you've harnessed the power of the staging area and lightweight topic branching and merging.

Now you'll explore a couple of tools you may not necessarily use on a day-to-day basis but can often be useful.

Reset Demystified

Before moving on to more specialized tools, let's talk about **Reset** and **Checkout**. These commands are two of the most confusing parts of Git when you first encounter them. They do so many things that it seems hopeless to actually understand them and employ them properly. For this, we recommend a simple metaphor.

The Three Trees

An easier way to think about **Reset** and **Checkout** is through the mental frame of Git being a content manager of three different trees. By "tree" here, we really mean "collection of files", not specifically the data structure. (There are a few cases where the index doesn't exactly act like a tree, but for our purposes it is easier to think about it this way for now.)

Git as a system manages and manipulates three trees in its normal operation:

Tree	Role
HEAD	Last commit snapshot, next parent
Index	Proposed next commit snapshot
Working Directory	Sandbox

The HEAD

HEAD is the pointer to the current branch reference, which is in turn a pointer to the last commit made on that branch. That means HEAD will be the parent of the next commit that is created. It's generally simplest to think of HEAD as the snapshot of *your last commit on that branch*.

The Index

The ***index*** is your *proposed next commit*. We've also been referring to this concept as Git's "Staging Area" as this is what Git looks at when you commit your changes.

Git populates this index with a list of all the file contents that were last checked out into your working directory and what they looked like when they were originally checked out. You then replace some of those files with new versions of them, and **Commit** converts that into the tree for a new commit.

The index is not technically a tree structure — it's actually implemented as a flattened manifest — but for our purposes it's close enough.

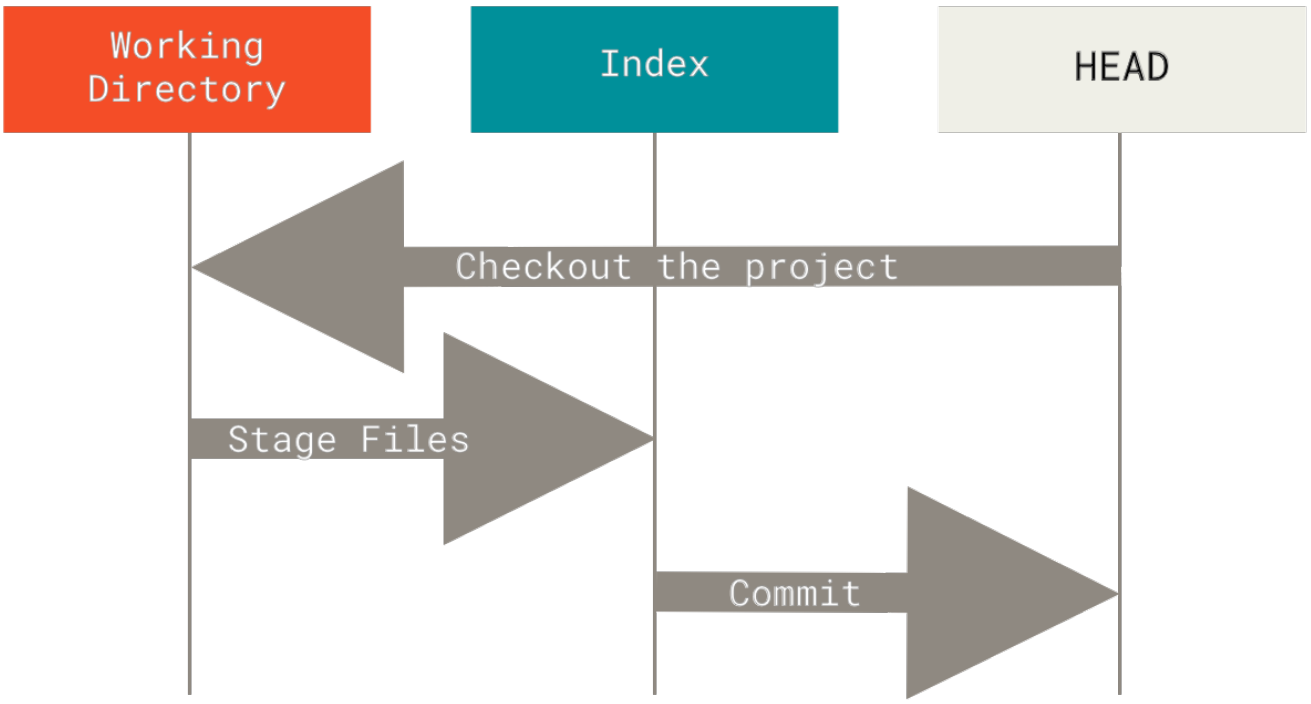
The Working Directory

Finally, you have your ***working directory*** (also commonly referred to as the "working tree"). The other two trees store their content in an efficient but inconvenient manner, inside the `.git` folder. The working directory unpacks them into actual files, which makes it much easier for you to edit them. Think of the working directory as a *sandbox*, where you can try changes out before committing them to your staging area (index) and then to history.

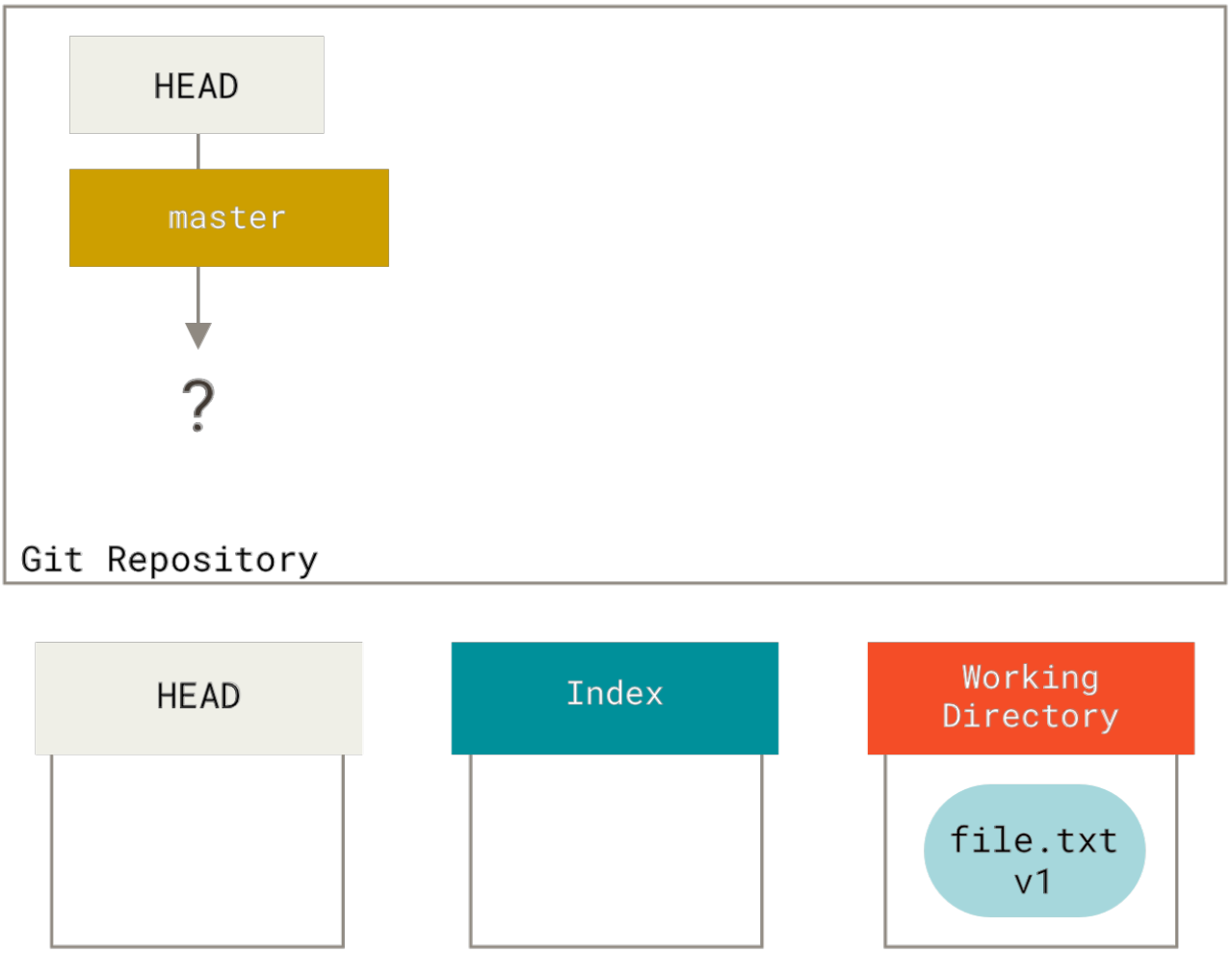
```
$ tree
.
├──── README
├──── Rakefile
└──── lib
    └──── simplegit.rb

1 directory, 3 files
```

The Workflow

Git's typical workflow is to record snapshots of your project in successively better states, by manipulating these three trees.

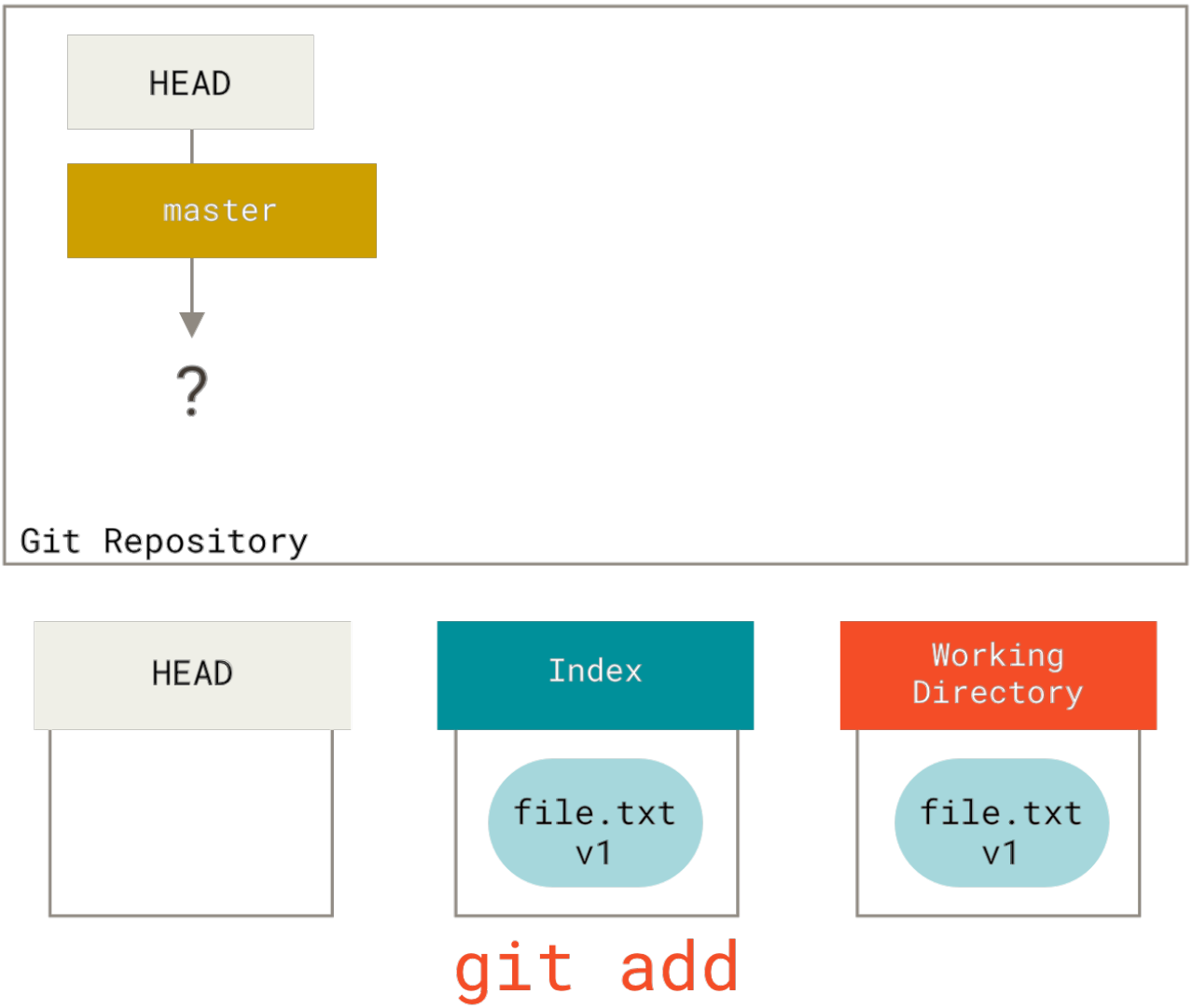


Let's visualize this process: say you create a new repository, and add a single file into the working directory. We'll call this **v1** of the file, and we'll indicate it in blue. The new repository will have a HEAD reference which points to the unborn `master` branch.

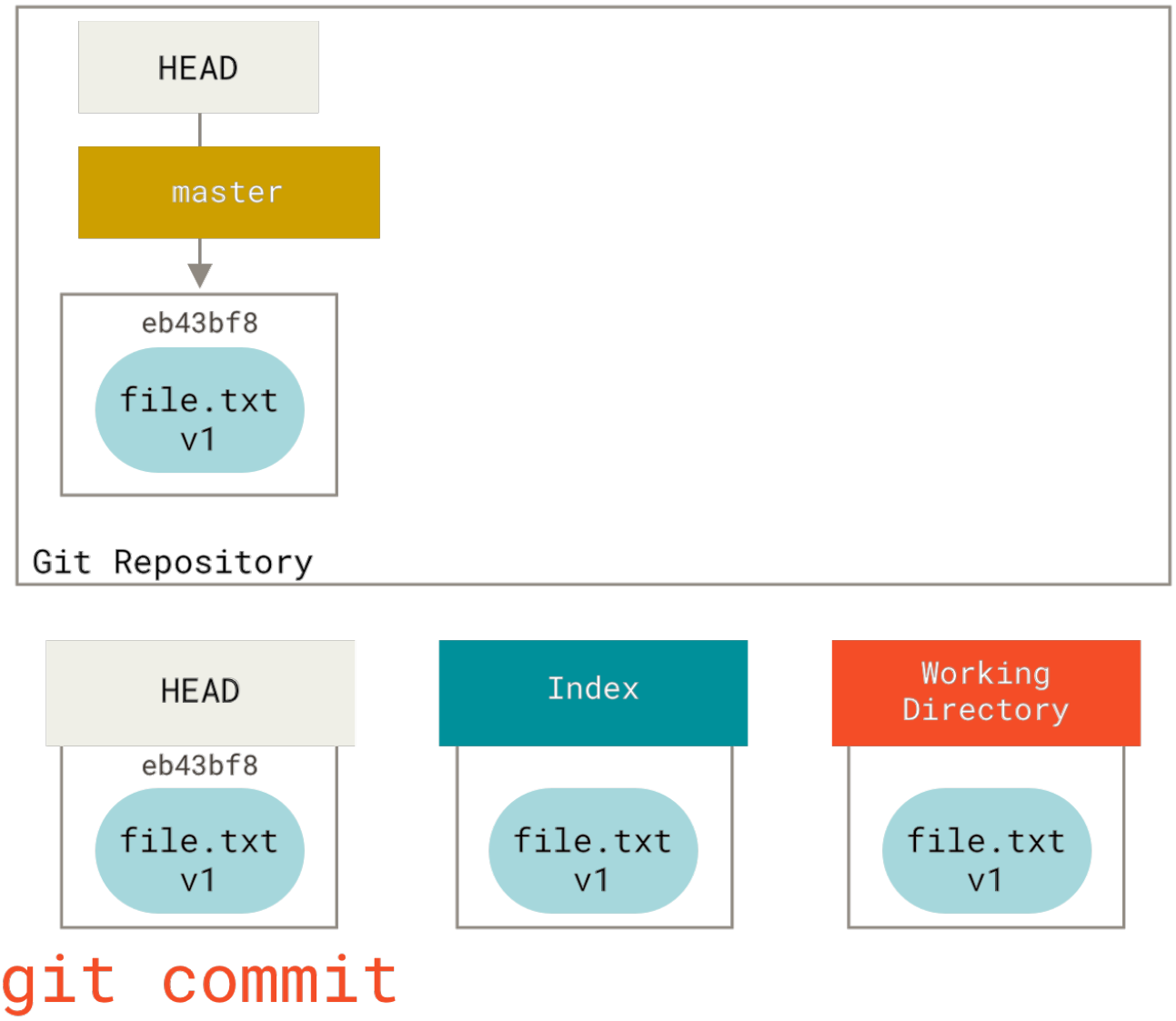


At this point, only the working directory tree has any content.

Now we want to commit this file, so we use **Stage** to take content in the working directory and copy it to the index.

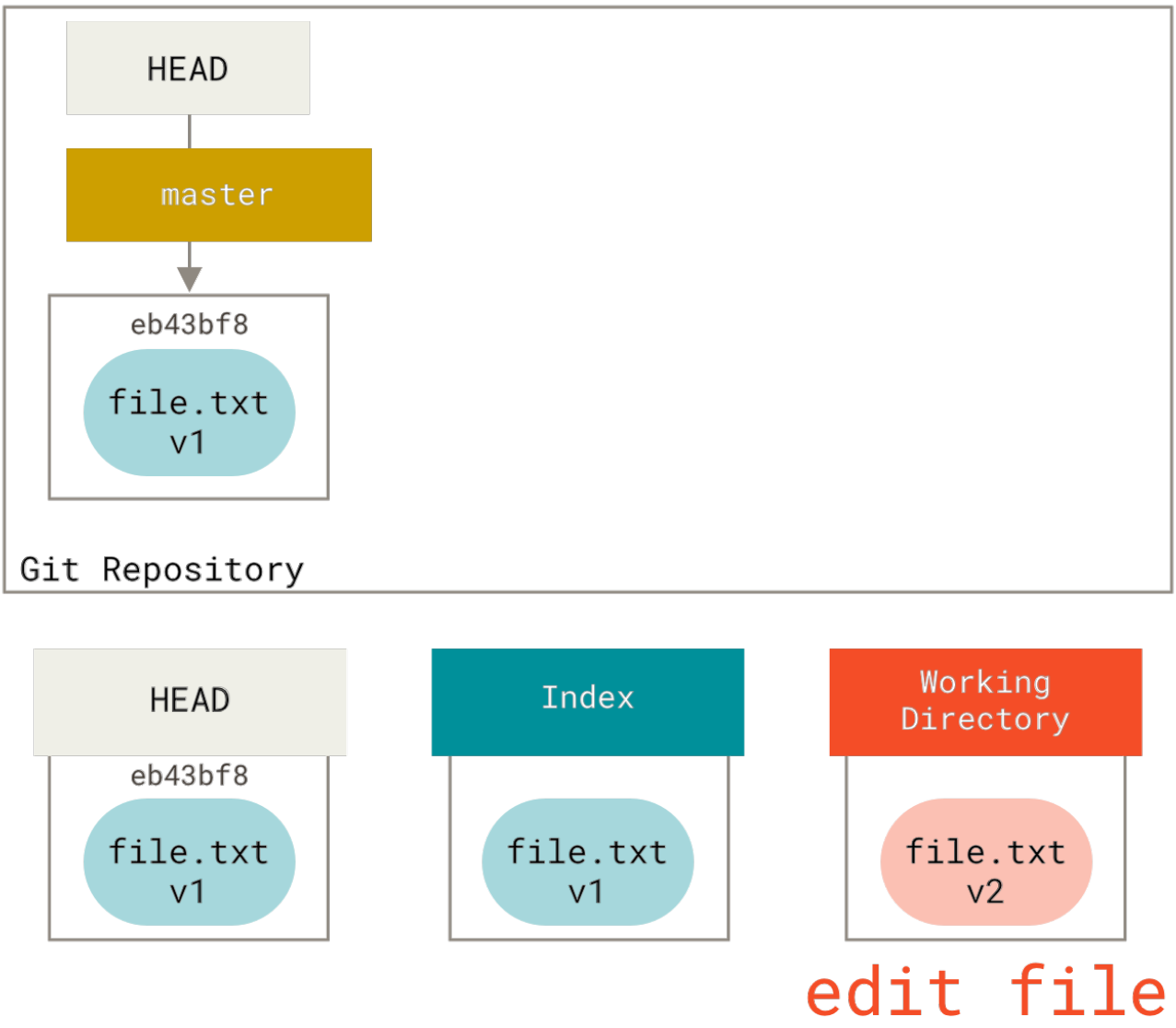


Then we **Commit**, which takes the contents of the index and saves it as a permanent snapshot, creates a commit object which points to that snapshot, and updates `master` to point to that commit.

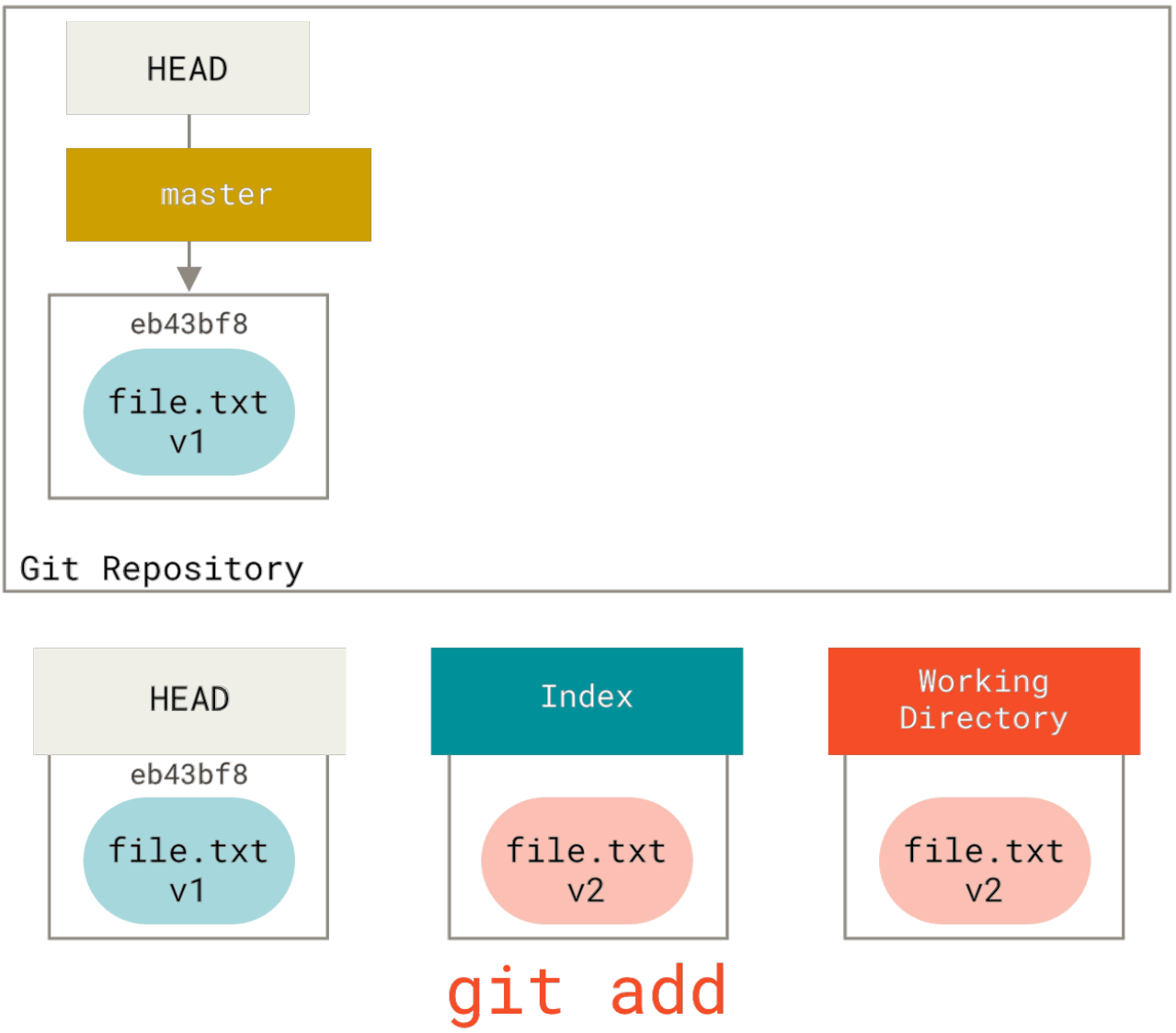


Team Explorer **Changes**, will show no changes, because all three trees are the same.

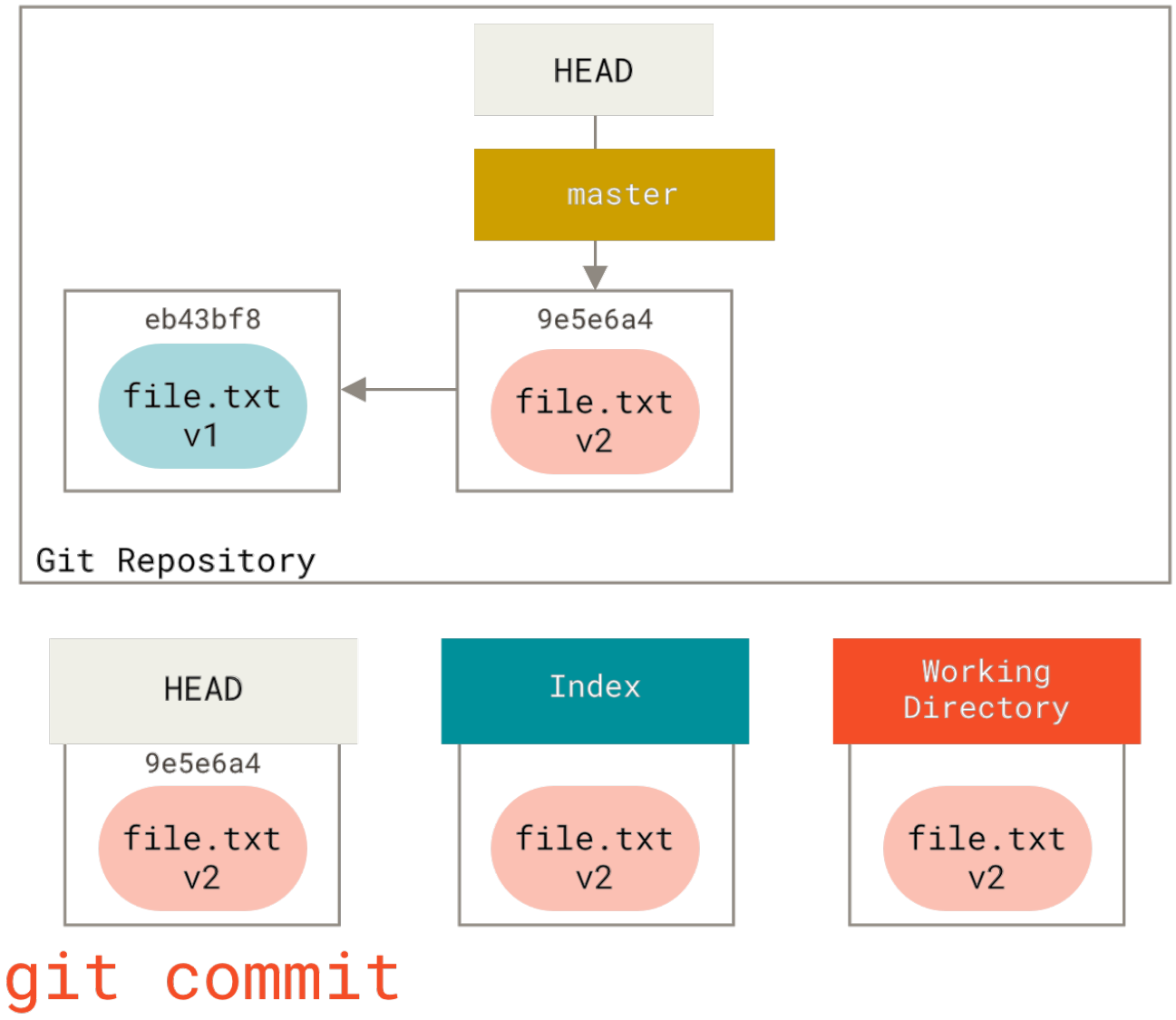
Now we want to make a change to that file and commit it. We'll go through the same process; first, we change the file in our working directory. Let's call this **v2** of the file, and indicate it in red.



Team Explorer **Changes**, right now, will show the file under “Changes”, because that entry differs between the index and the working directory. Next we click **add** on it to stage it into our index.



At this point, Team Explorer **Changes**, shows the file under “Staged Changes” because the index and HEAD differ — that is, our proposed next commit is now different from our last commit. Finally, we run **Commit** to finalize the commit.



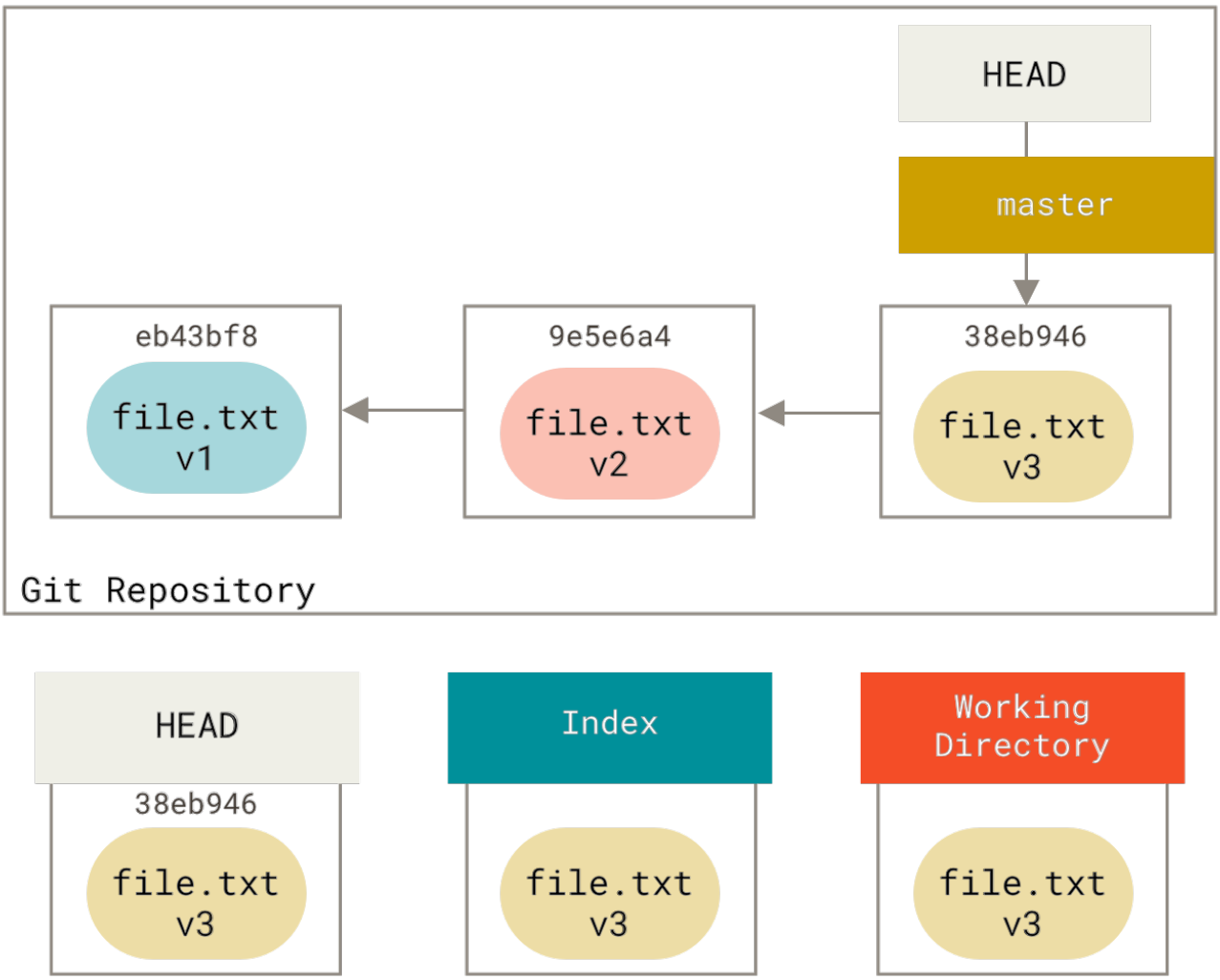
Now Team Explorer **Changes** will show no staged changes or changes, because all three trees are the same again.

Switching branches or cloning goes through a similar process. When you checkout a branch, it changes **HEAD** to point to the new branch ref, populates your **index** with the snapshot of that commit, then copies the contents of the **index** into your **working Directory**.

The Role of Reset

The effect of **Reset** makes more sense when viewed in this context.

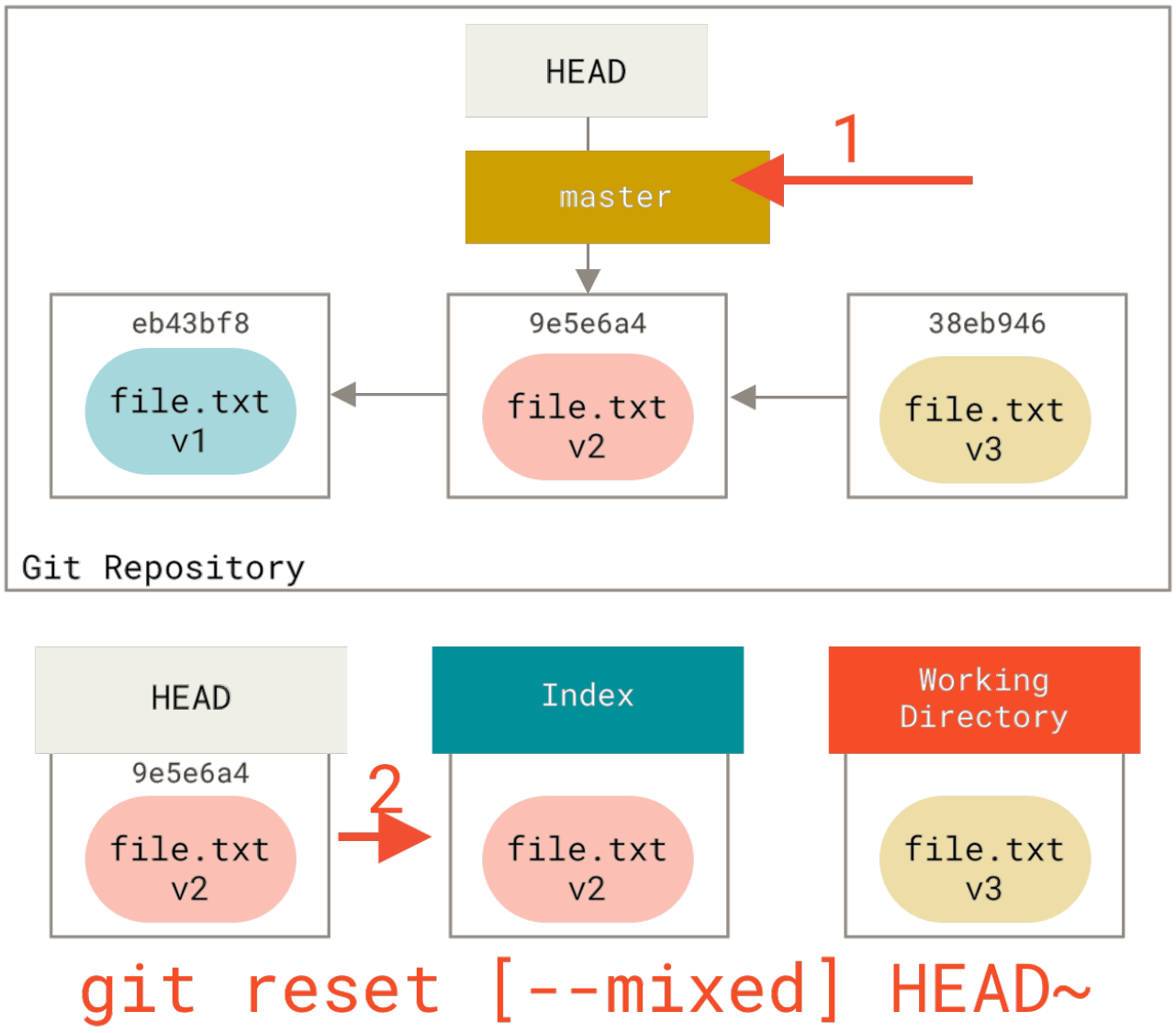
For the purposes of these examples, let's say that we've modified `file.txt` again and committed it a third time. So now our history looks like this:



Let's now walk through exactly what **Reset** does when you use it. It directly manipulates these three trees in a simple and predictable way. A mixed reset updates ***HEAD*** and ***Index*** without updating the working directory, a hard reset updates all three.

Keep Changes: Move HEAD and Index (--mixed)

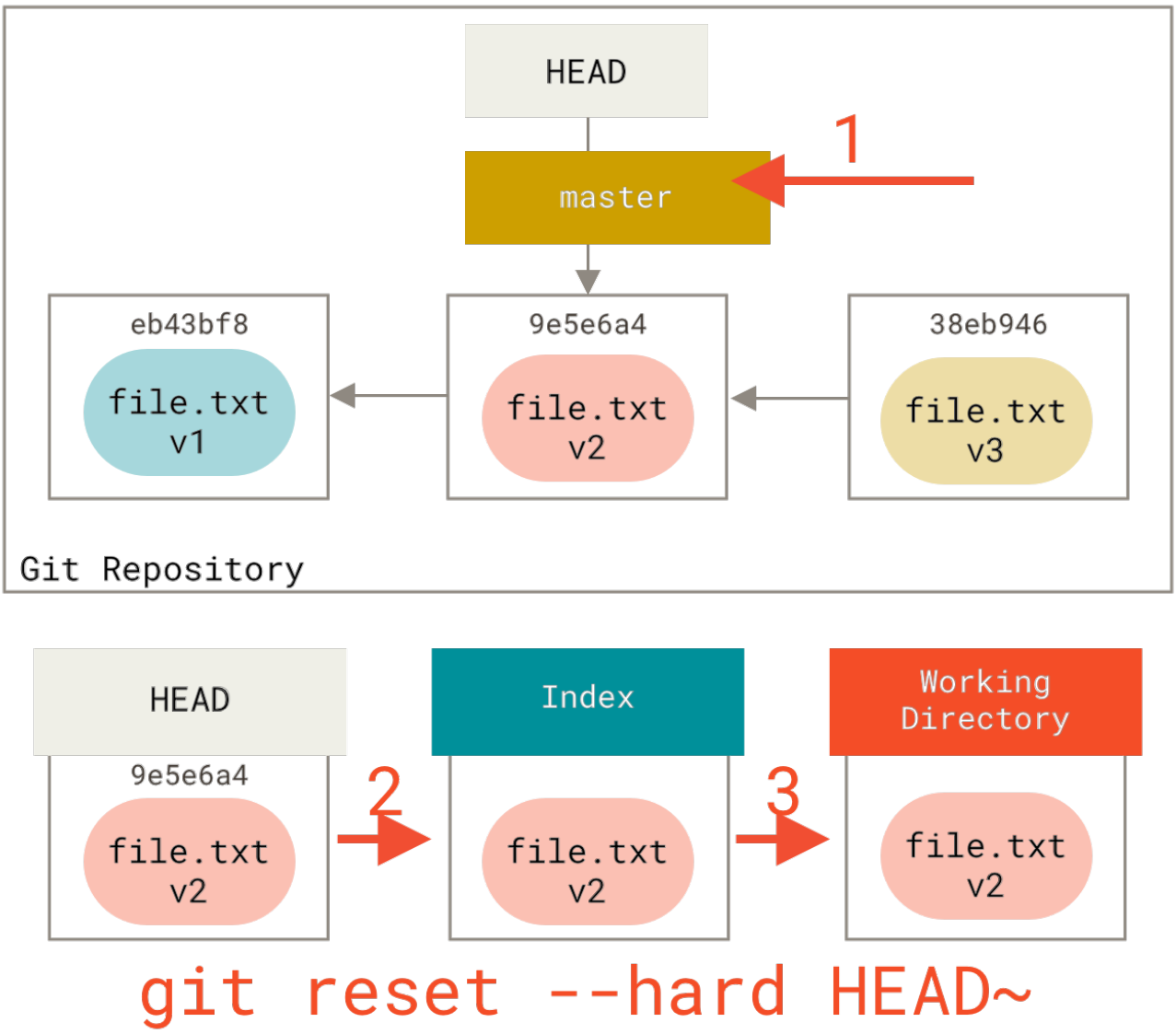
The first thing **Reset** will do is move what HEAD points to. This isn't the same as changing HEAD itself (which is what `checkout` does); **Reset** moves the branch that HEAD is pointing to. This means if HEAD is set to the `master` branch (i.e. you're currently on the `master` branch), resetting to commit `9e5e6a4` will start by making `master` point to `9e5e6a4`. Thene **Reset** updates the index with the contents of whatever snapshot HEAD now points to.



Now take a second to look at that diagram and realize what happened: it undid your last `commit`, and also unstaged everything. You rolled the branch back to before staging and committing all your changes.

Delete Changes: Updating the Working Directory (--hard)

The last thing that **Reset** will do is to make the working directory look like the index. If you use the `--hard` option, it will continue to this stage.



So let's think about what just happened. You undid the commit, staged changes, **and** all the work you did in your working directory.

This is the only dangerous variant of **Reset**, and one of the very few cases where Git will actually destroy data. Any **Reset**>*Keep Changes* can be pretty easily undone, but **Reset**>*Delete Changes* cannot, since it forcibly overwrites files in the working directory. In this particular case, we still have the **v3** version of our file in a commit in our Git DB, and we could get it back by looking at our `reflog` with the git command line. If we had not committed it, Git still would have overwritten the file and it would be unrecoverable.

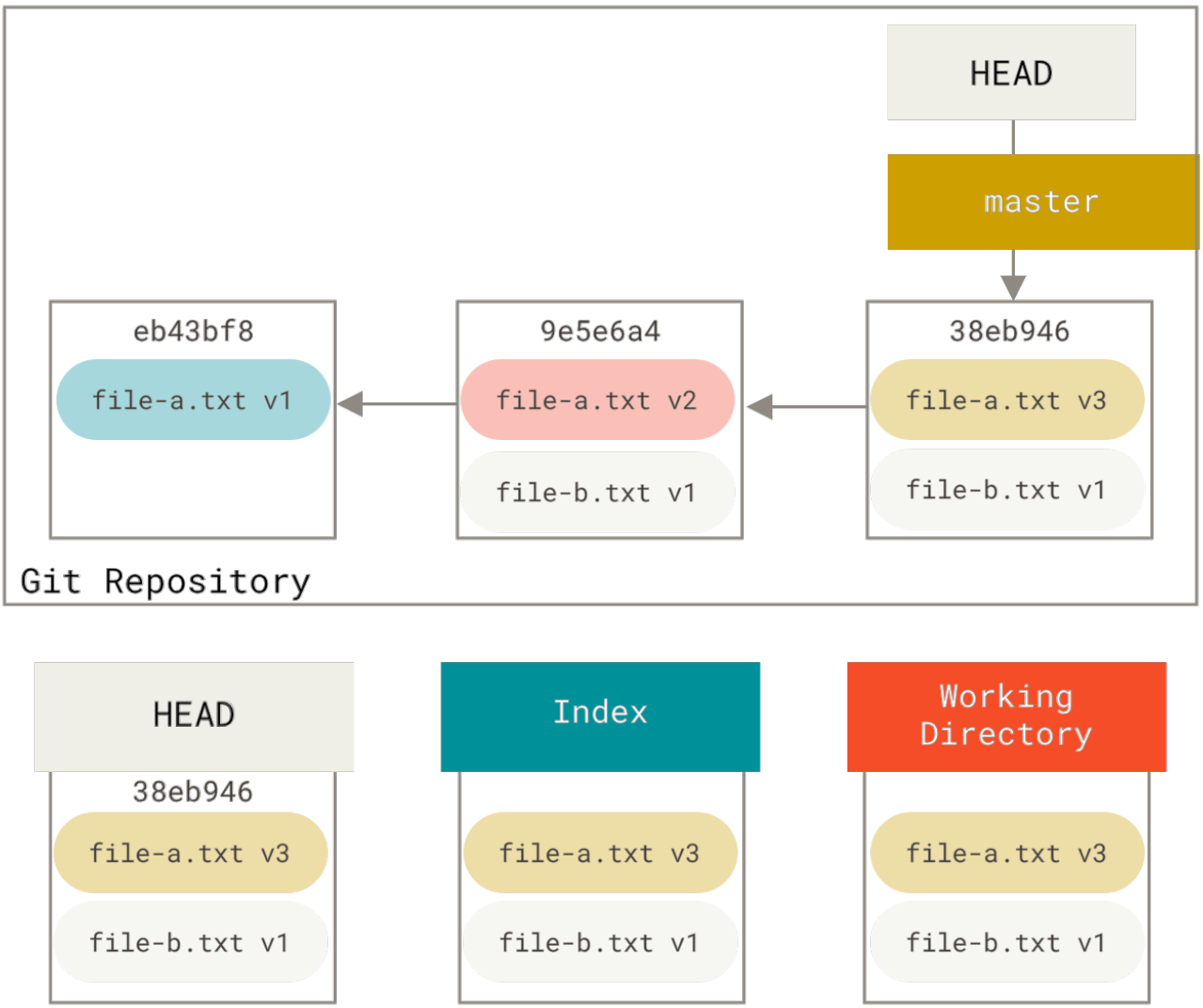
Squashing

Let's look at how to do something interesting with this newfound power — squashing commits.

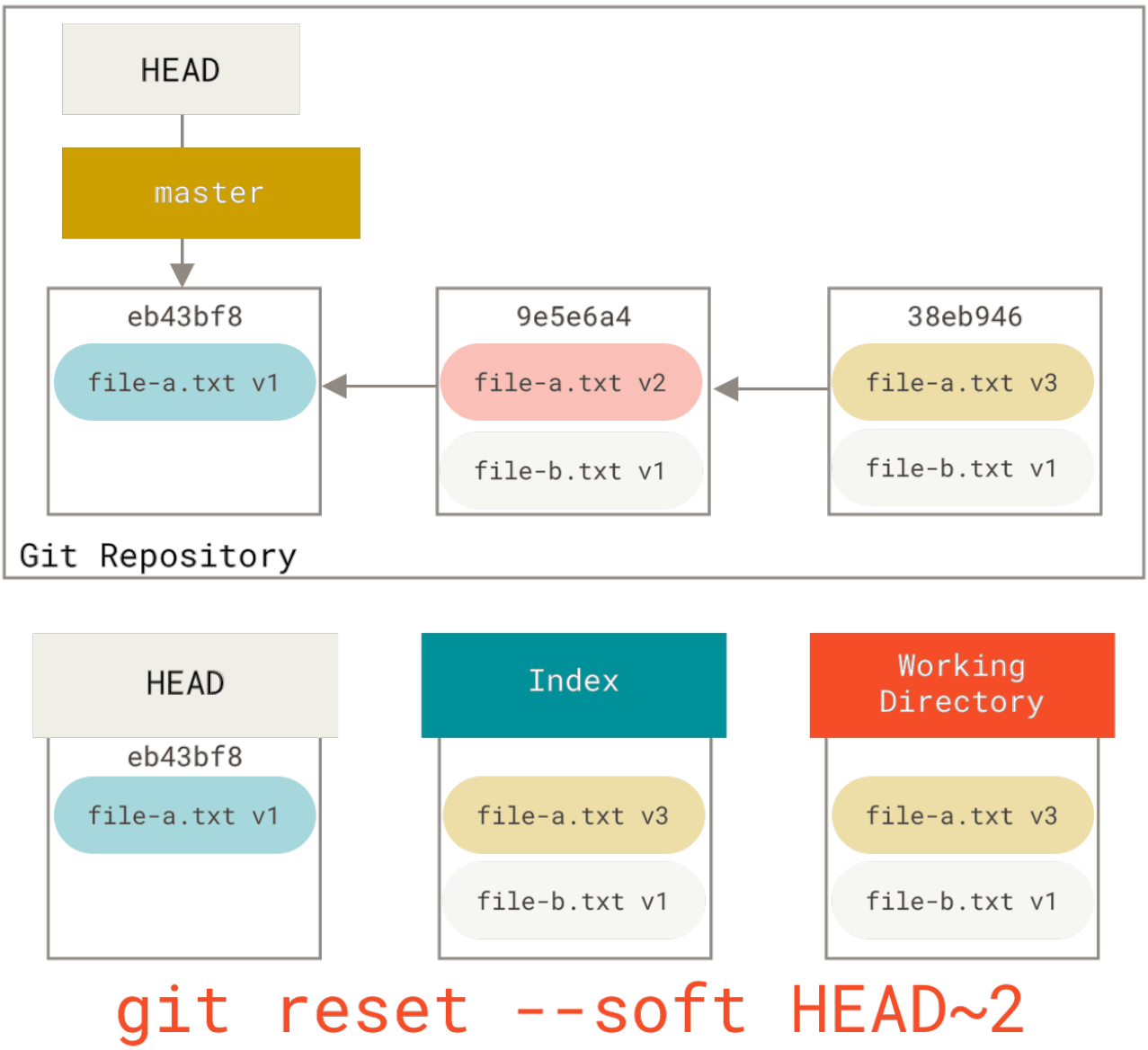
Say you have a series of commits with messages like "oops.", "WIP" and "forgot this file". You can use **Reset** to quickly and easily squash them into a single commit that makes you look really smart.

Let's say you have a project where the first commit has one file, the second commit added a new

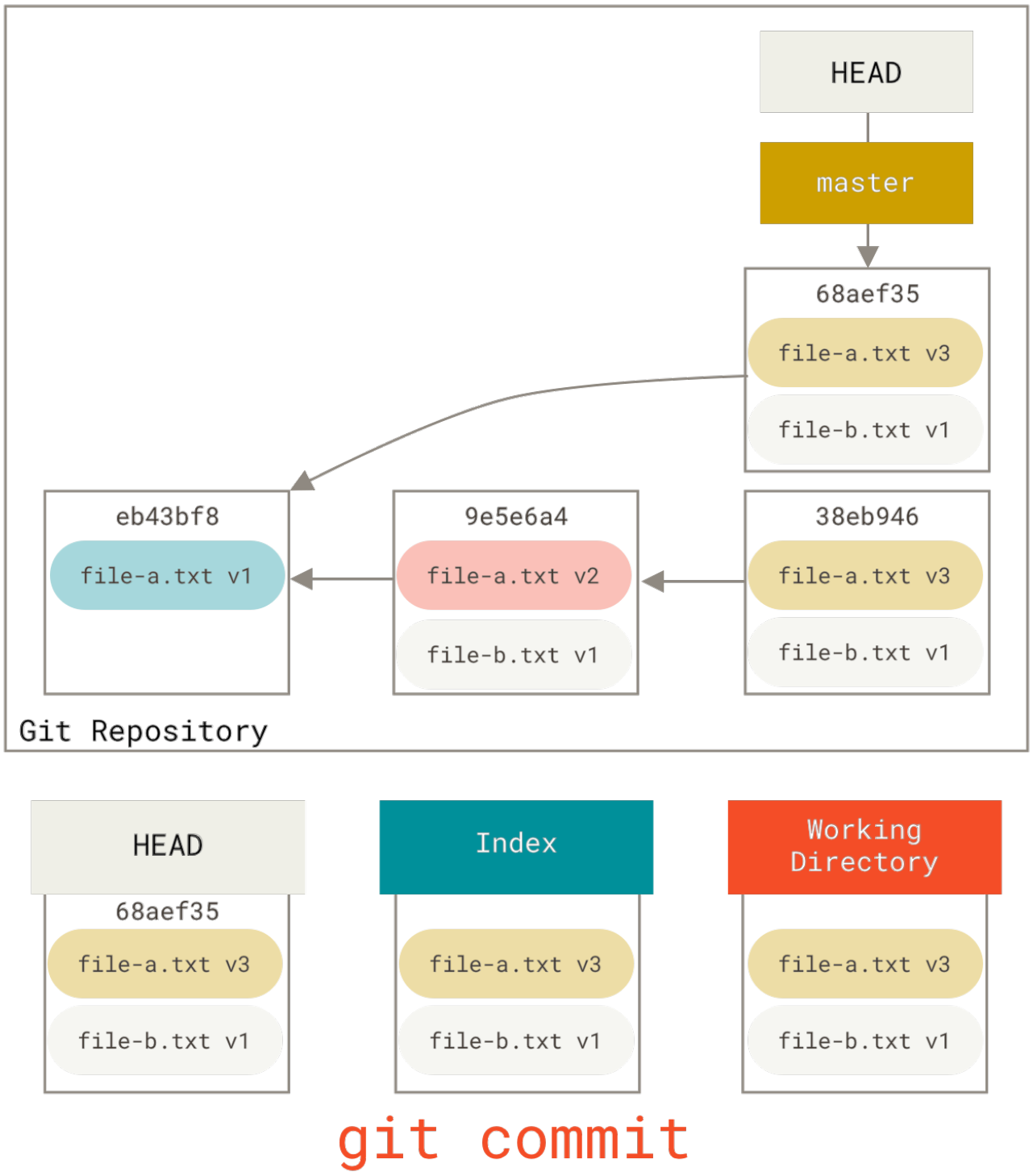
file and changed the first, and the third commit changed the first file again. The second commit was a work in progress and you want to squash it down.



You can run **Reset**>*Keep Changes* to move the branch back to the last commit you want to keep:



And then simply run **Commit** the changes again:



Now you can see that your reachable history, the history you would push, now looks like you had one commit with `file-a.txt` v1, then a second that both modified `file-a.txt` to v3 and added `file-b.txt`. The commit with the v2 version of the file is no longer in the history.

Remember to ensure that there are no uncommitted changes in your working directory before squashing commits with this method, to avoid accidentally including additional changes.

Check It Out

Finally, you may wonder what the difference between **checkout** and `reset` is. Like `reset`, `checkout`

manipulates the three trees, and it is a bit different depending on whether you give the command a file path or not.

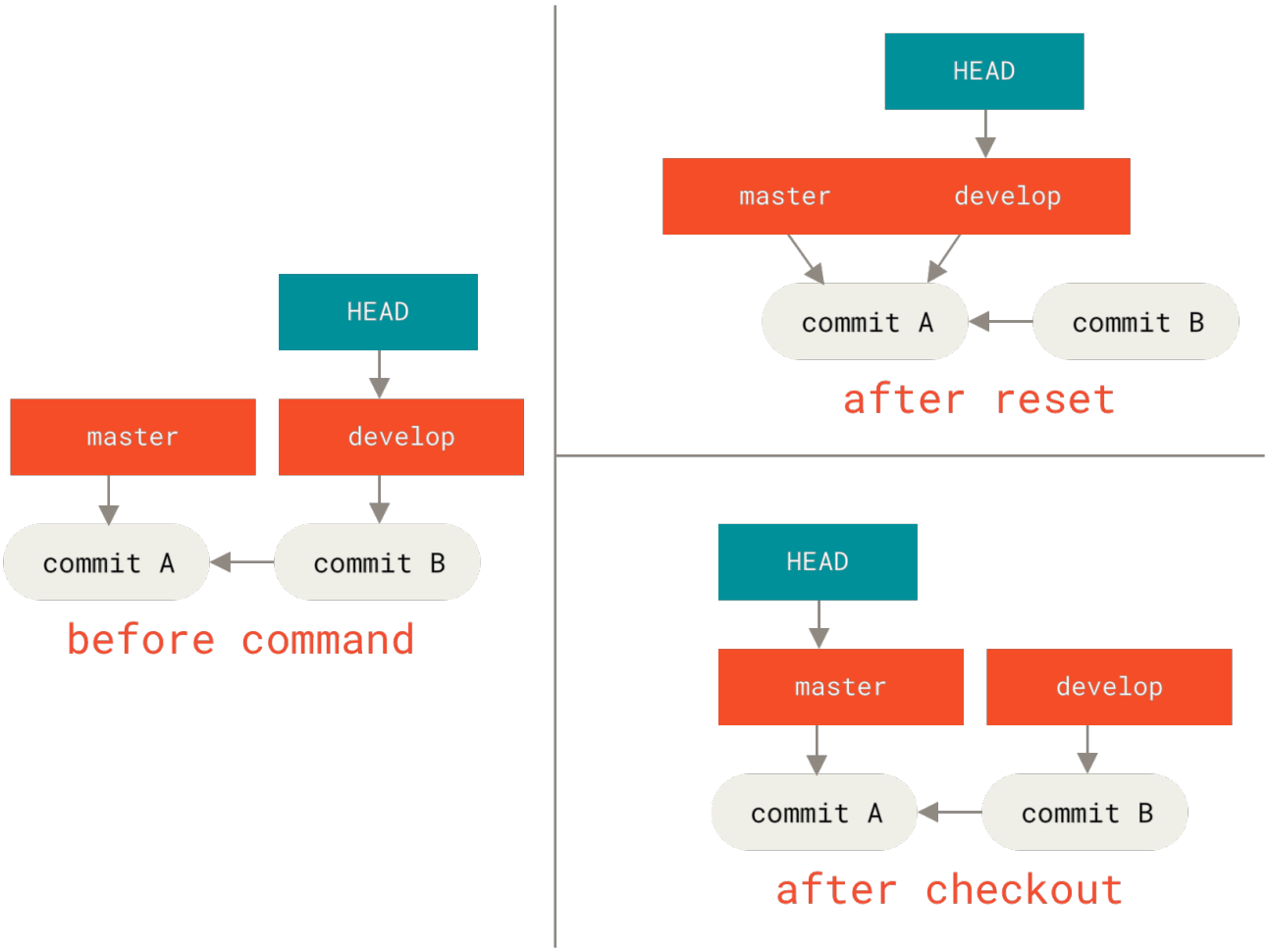
Checking out a branch is pretty similar to doing a **Reset**>*Delete Changes* to that branch, in that it updates all three trees for you to look like `[branch]`, but there are two important differences.

First, unlike **Reset**>*Delete Changes*, **Checkout** is working-directory safe; it will check to make sure it's not blowing away files that have changes to them. Actually, it's a bit smarter than that — it tries to do a trivial merge in the working directory, so all of the files you *haven't* changed will be updated. **Reset**>*Delete Changes*, on the other hand, will simply replace everything across the board without checking.

The second important difference is how **Checkout** updates HEAD. Whereas **Reset** will move the branch that HEAD points to, **Checkout** will move HEAD itself to point to another branch.

For instance, say we have `master` and `develop` branches which point at different commits, and we're currently on `develop` (so HEAD points to it). If we **Reset** to `master`, `develop` itself will now point to the same commit that `master` does. If we instead **Checkout** `master`, `develop` does not move, HEAD itself does. HEAD will now point to `master`.

So, in both cases we're moving HEAD to point to commit A, but *how* we do so is very different. **Reset** will move the branch HEAD points to, **checkout** moves HEAD itself.



Summary

Hopefully now you understand and feel more comfortable with the **reset** command, but are probably still a little confused about how exactly it differs from **checkout**.

Here's a cheat-sheet for which operations affect which trees. The "HEAD" column reads "REF" if that command moves the reference (branch) that HEAD points to, and "HEAD" if it moves HEAD itself. Pay especial attention to the *WD Safe?* column — if it says **NO**, take a second to think before using that command.

	HEAD	**Index**	**Workdir**	**WD Safe?**
Commit Level				
Reset>Keep Changes	REF	YES	NO	YES
Reset>Delete Changes	REF	YES	YES	**NO**
Checkout	HEAD	YES	YES	YES

Debugging with Git

In addition to being primarily for version control, Git also provides a couple commands to help you debug your source code projects. Because Git is designed to handle nearly any type of content, these tools are pretty generic, but they can often help you hunt for a bug or culprit when things go wrong.

File Annotation

If you track down a bug in your code and want to know when it was introduced and why, file annotation is often your best tool. It shows you what commit was the last to modify each line of any file. So if you see that a method in your code is buggy, you can annotate the file with **Blame** to determine which commit was responsible for the introduction of that line. You can right-click and **Blame** a file in working folder using the solution explorer, or a file change by a commit in Team Explorer.

The following shows the commits introducing lines of the recording changes section of this guide.

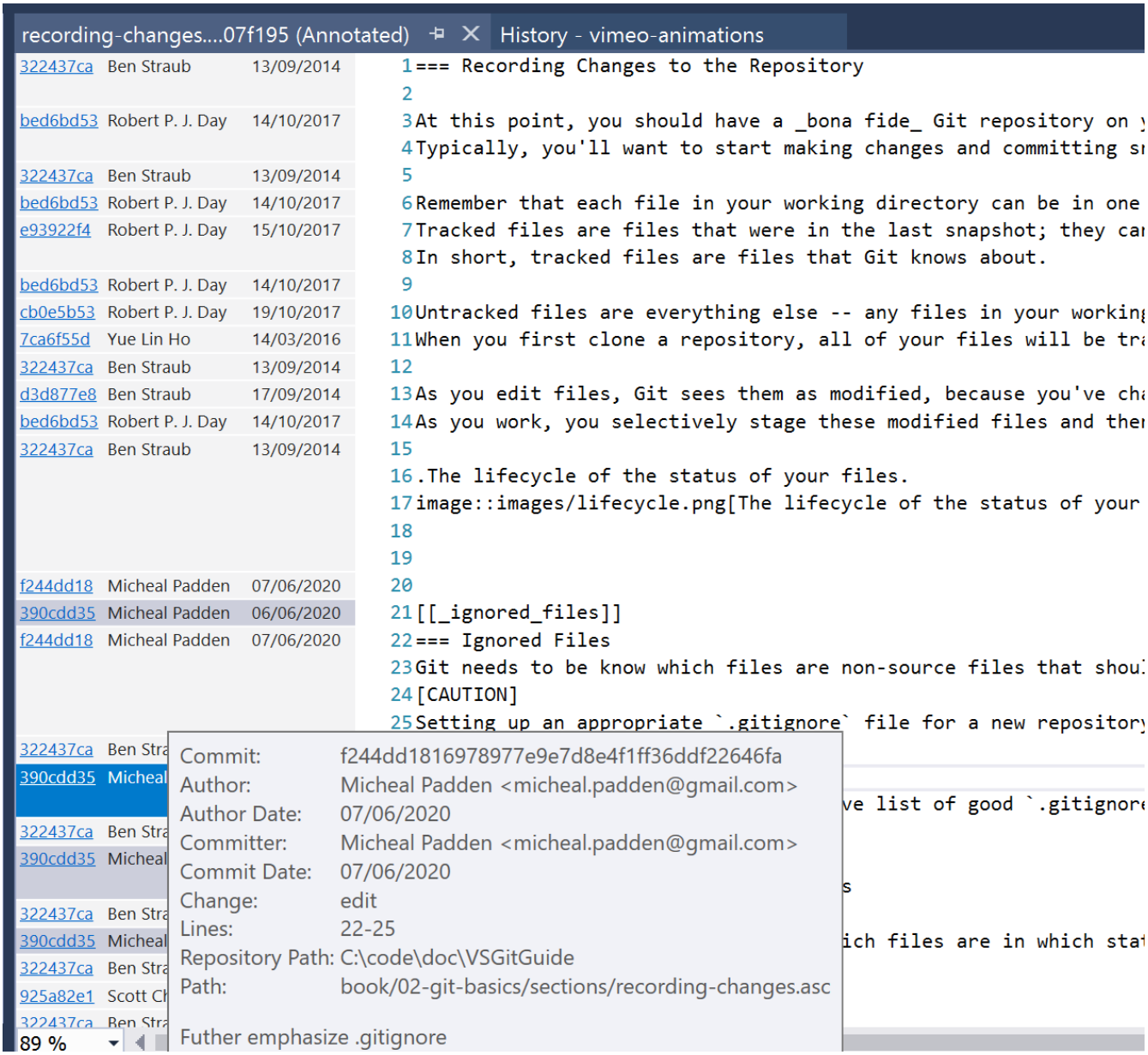


Figure 66. Blame.

Notice that the first field is the partial SHA of the commit that last modified that line. The next two fields are values extracted from that commit — the author name and the authored date of that commit — so you can easily see who modified that line and when. After that come the line number and the content of the file. Hovering over the commit shows a tip with further commit details including the commit message.

Another cool thing about Git is that it doesn't track file renames explicitly. It records the snapshots and then tries to figure out what was renamed implicitly, after the fact. In the example below, the highlighted tip shows that the relevant section of SwitchingToGitInVisualStudio.asc originally came from progit.asc.

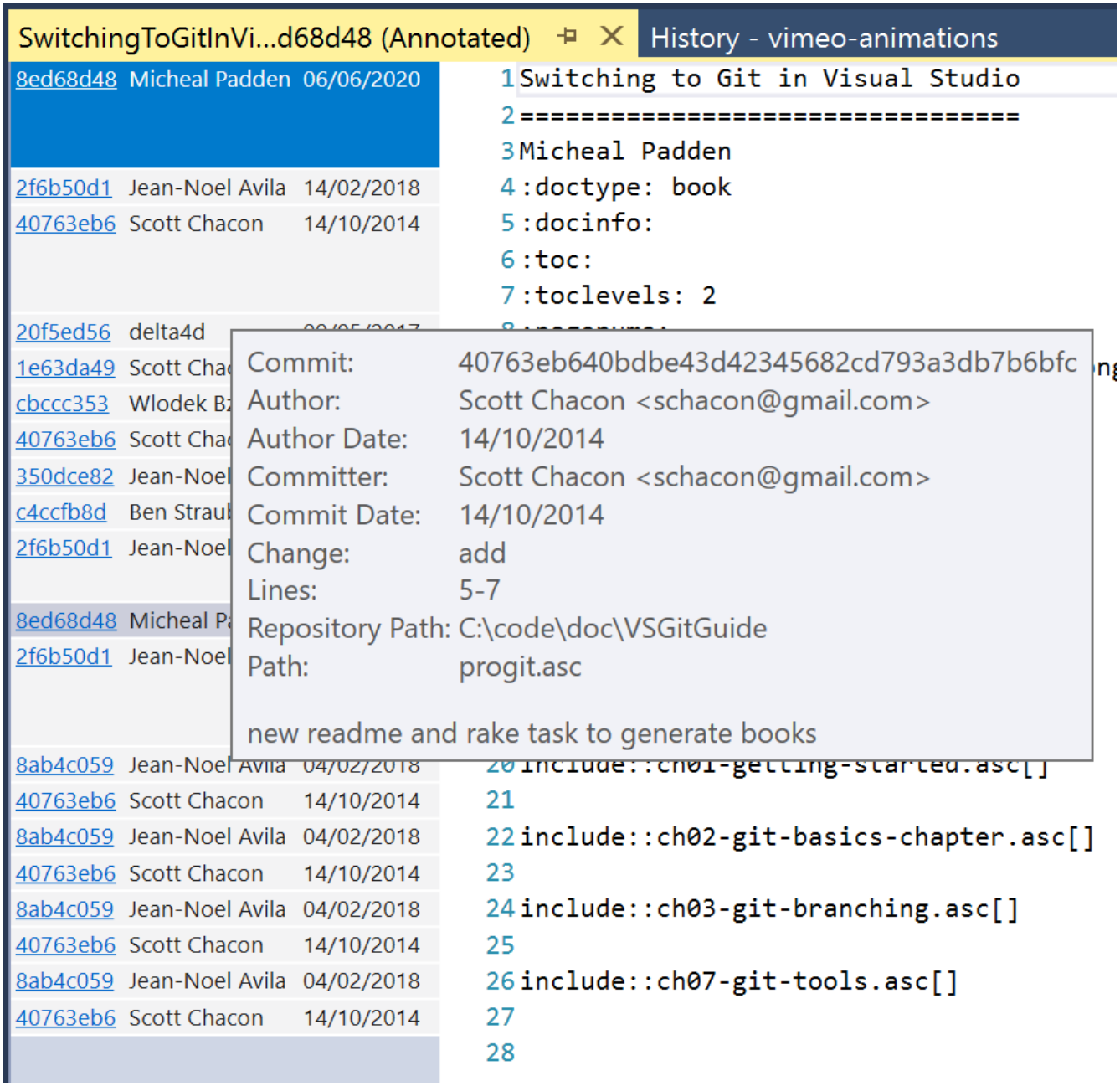


Figure 67. Blame Across Rename.

Summary

You've seen a couple of advanced tools that allow you to squash commits and find which commit changes came from. At this point, you should be able to do most of the things in Git that you'll need day to day and feel comfortable doing so.

Further Reading

There are some tasks that you will need to perform from time to time. For these you will need to step outside Visual Studio and use a dedicate Git GUI and/or the command line. The Visual Studio Git Extensions plugin provides access to tools that give you more control over your commits and staging area. The command line provides powerful merging and conflict resolution and debugging tools, commit signing, and project dependency management with subprojects. You can read about these in Chapter 7 of Pro Git, from the book on which this Guide is based.