

Git Doesn't Have to be Hard

Git Doesn't Have to be Hard

Peter S. Conrad

Copyright ©2021 by Peter S. Conrad

All rights reserved.

To Mel, who is never satisfied with “good enough.”

Contents

Preface.....	xv
How to Use This Book.....	xvii
Command Examples	xvii
What you Should Know	xvii
Wildcards.....	xviii
Git Doesn't Have to Be Hard	
Introduction: Why is Git So Hard?.....	3
Concepts	
Source Control	7
Centralized Source Control.....	7
How Git is Different.....	8
How Git Works.....	11
Working with Git	15
Basic Source Control with Git	15
Sharing Your Work.....	16
Branching.....	17
Branching Strategies.....	18
The GitHub Flow	18
Git Centralized Workflow	20
Branching Problems.....	20
How It All Fits Together	23

Graphical Git.....	23
Git on the Command Line.....	24
Git Online.....	24
Reviewing a Pull Request.....	25
Get Started	
Get Set Up with Git.....	29
Get Git	29
Set up a Repository.....	29
Set up a Repository with Bitbucket and Sourcetree.....	30
Set up a Repository with GitHub and GitHub Desktop.....	31
Result.....	31
Tutorials.....	33
Tutorial: Day to Day Work in a Git Client	33
Create a Working Branch.....	33
Commit Some Changes.....	34
Push and Create a Pull Request.....	35
Review a Pull Request	36
Approve a Pull Request	37
Merge a Pull Request.....	37
Tutorial: Working on the Command Line	38
Switch to a Working Branch	38
Commit Some Changes.....	39

Push and Create a Pull Request.....	39
Stay out of Trouble	41
Choose the Correct Branch.....	41
Pull Often.....	41
Think before Committing, Pushing, or Branching	41
Use a Simple Branching Strategy.....	41
Be Watchful	42
Write Good Commit Messages.....	43
Fun with Git	
Command Line Tips and Tricks	47
Authenticate Git on the Command Line.....	47
Cache Your Credentials.....	48
Linux	48
macOS.....	49
Windows.....	49
Examine the Past	51
Git Log.....	51
Git Reflog	55
Save Some Keystrokes.....	56
Compare Branches or Commits.....	57
Checkout Old Commits.....	59
Label Commits with Tags.....	60

Share Tags.....	60
Clean Up	62
Ignore Irrelevant Files.....	62
Delete Old Branches	62
Rebase	63
Move, Rename, or Delete Files	65
Git Move	65
Git Remove	65
Pack Up.....	67
Create a Git Wiki.....	69
Git Wiki Structure	69
Set Up a Wiki	70
Set Up a Wiki in Bitbucket and Sourcetree	70
Set Up a Wiki in GitHub and GitHub Desktop.....	70
Work with Content on the Host	71
Work with Content Locally	71
Clone a Wiki in Bitbucket and Sourcetree	71
Clone a Wiki in GitHub and GitHub Desktop	72
Clone a Wiki on the Command Line	72
Tutorial: Create Structured Wiki Content	73
Create Some Content Locally	73
Take a Look.....	73

Publish a Website	75
Bitbucket.....	75
GitHub Pages	76

Reference

Basic Git Operations	79
Pull	81
Pull in Sourcetree	81
Pull in GitHub Desktop.....	81
Pull on the Command Line	81
Stage and Commit	83
Stage and Commit in Sourcetree.....	83
Stage and Commit in GitHub Desktop.....	85
Stage and Commit on the Command Line.....	85
View Your Changes	87
View Your Changes in GitHub Desktop or Sourcetree	87
View Your Changes on the Command Line	88
Push	89
Push in Sourcetree.....	89
Push in GitHub Desktop.....	89
Push on the Command Line.....	89
Create a Branch	91
Create a Branch in Sourcetree.....	91

Create a Branch in GitHub Desktop.....	93
Create a Branch on the Command Line.....	94
Create a Pull Request	96
Create a Pull Request in Bitbucket and Sourcetree.....	96
Create a Pull Request in GitHub and GitHub Desktop.....	97
Create a Pull Request on the Command Line	99
Approve and Merge.....	100
Merge a Pull Request in Bitbucket	101
Merge a Pull Request in GitHub	101
Trouble	103
Edited in the Wrong Branch	103
Edited the Wrong File.....	104
Merge Conflict	105
Detached HEAD	106
Staged by Mistake	106
Committed by Mistake	107
Committed in the Wrong Branch	108
Lost Some Changes to History	108
Retrieve All Changes from an Old Commit.....	108
Grab an Old Version of a File.....	109
Pushed by Mistake	109
Revert a Bad Push.....	110

More trouble	111
Glossary	113

Preface

If you're a seasoned engineer working in a large team, this book is probably not for you. You probably eat commit hash for breakfast and wash it down with a cup of rebase.

On the other hand, if you're new to Git, this book will help you get started and stay out of trouble. You'll learn the basics of working with Git day to day, how to collaborate with others, and a few fun tips and tricks.

Git Doesn't Have to Be Hard is meant for people who are relatively new to Git. If you're a content developer collaborating with engineers, a tech writer working with docs-as-code, or a programmer who needs to track and share your work—this book is for you.

The first few chapters, in the "Concepts" section, explain how Git works, and why it's better than traditional centralized source control. Rather than starting with the inner architecture of Git, these chapters explain what problems Git is trying to solve, and what that means to you as a user. "Get Started" helps you get set up with an online Git host, teaches you how to get started, and tells you how to avoid the most common problems. "Fun with Git" gives you some tips and tricks to use once you're comfortable with Git. A reference chapter at the back of the book provides detailed instructions for the basic Git operations.

If you need to use Git but don't know how, *read on!*

How to Use This Book

This book is short enough to read all the way through, but you don't have to.

- To learn how Git works, read “Concepts” on page 5.
- To use Git right away, read “Get Started” on page 27.
- If you know a little about Git already, read “Fun with Git” on page 45.
- If you're using Git and you've gotten into some kind of trouble, see “Trouble” on page 103.

Command Examples

For command line instructions, examples that don't include output from a command also omit the command prompt:

```
git branch
```

To distinguish the output from the command you type, examples that include command output also show the prompt:

```
$ git branch
```

```
* main
```

Although the prompt shown in the examples is a dollar sign (\$), it can be different on different systems. On Windows, for example, it is often a greater-than sign (>).

TIP Don't type the prompt.

Options and parameters that you need to add are in {brackets}.

In the printed instructions, some commands spill over from one line to the next. When you use one of these commands, type it all on one line, keeping file paths together.

What you Should Know

It's helpful to have some familiarity with the command line, including:

- How to get to the command line
- Navigating and listing directories and files
- Specifying files with wildcards
- Running commands, including parameters and options

Wildcards

Wildcards are characters that let you specify groups of files or directories with similar names. There are two main wildcards:

- `*` - stands in for a group of characters
- `?` - stands in for a single character

If you want to specify a single file, you use its name. For example: `my-file.txt`.

If you want to specify all files that end in `.txt`, you can use the `*` wildcard: `*.txt`.

The `?` wildcard, lets you specify filenames where a specific character varies: `m?-file.txt` specifies all files that match the name, but with any character in the `?` spot. That is, `m0-file.txt`, `mx-file.txt`, and `mW-file.txt` would all match.

Any of the Git procedures (and the `.gitignore` file) can use wildcards to specify multiple files.

Examples

Add the `my-file.txt` file:

```
git add my-file.txt
```

Add all `.txt` files in the current directory:

```
git add *.txt
```

Add files named `file01`, `file02`, and so on, regardless of file extension:

```
git add file??.*
```


GIT DOESN'T HAVE TO BE HARD

Introduction: Why is Git So Hard?

I remember an engineer patiently explaining to me how Git works:

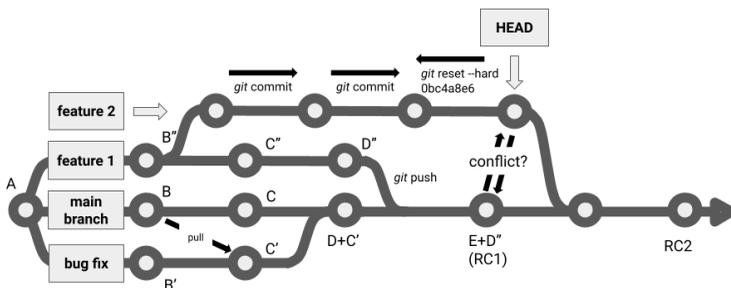
There's your working tree, your staging environment, your local and remote repos, and everything's really just a pointer, and so on, and so on...

I never understood it—so any time I got myself into trouble with Git, I never knew how to get myself out again. Now that I understand it better, I realize that there are three reasons people have trouble understanding Git.

First, Git is distributed rather than centralized. If you've ever used centralized source control, Git seems different from everything else you know. Concepts like checking in don't mean the same things in Git that they do in centralized source control.

Second, Git tracks changes rather than files. If you are used to circulating documents with version numbers, it's hard to make the switch to "unversioned" files.

Finally, Git is usually explained badly, in a hurry, by someone who knows too much about Git to think about the basics anymore. That's what happened to me. It took many years to undo the effects of the bad explanation. It's even worse if the explanation comes with a diagram.



Git is different from other source control tools, but it doesn't have to be hard to use. This book explains the important concepts behind Git, the basics of its use, and how to get out of a few common kinds of trouble.

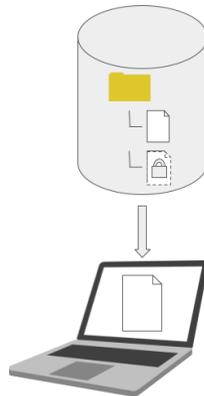
CONCEPTS

Source Control

Version control, or *source control*, is a system for keeping track of file versions and changes in a project. Most often, source control is used for tracking application codebases. With the rise of the docs-as-code movement, it has also become useful for documentation. It's now common to use source control systems for DITA, Markdown, reStructuredText, AsciiDoc, and other plain text markup languages. The goal of source control is to guarantee that files are stored securely and made available to the right people, that the contents of the files are known and verified, and that changes are tightly managed. With source control, you can see every change and who made it. If needed, you can roll back to previous versions of any file.

Centralized Source Control

Traditionally, source control has been centralized: a single central repository of files provides a single source of truth. Whatever is in the central repository is legitimate; everything else doesn't matter. To make changes to a file, you "check it out," like a library book. While you have a file checked out, you can edit it on your own computer (your "local machine"). While you're working on it, the repository's copy of the file is locked so that no one else can make changes to it.



When you're done editing the file and your work is tested and approved, you check the file back in. It is then available for other

people to check out. In this way, centralized source control ensures there's only one version of the truth.

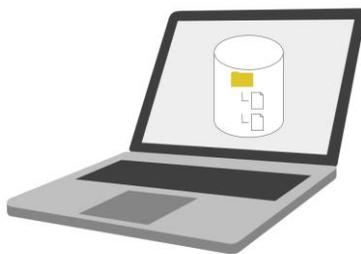
There are downsides to centralized source control:

- Only one person can work on any given file at a time, creating a bottleneck.
- Everything that's checked in becomes the truth, including bugs.
- Server traffic from a high volume of check-ins can cause lagging.
- The central repository is a single point of failure.

This last problem is a fatal weakness. If something happens to the repository, the whole project can be lost.

How Git is Different

The solution to all the aforementioned problems turns out to be decentralization. Because Git doesn't rely on the central repository as a single source of truth, there's neither a bottleneck nor a single point of failure. Branching is easier, making it possible to insulate people from each other's work and curb the spread of bugs. People can work in parallel without worrying about server traffic or locked files. With Git, every contributor has a complete source control system with a full copy of the repository.



This means thinking a little differently. When you use Git, you don't have to "check out" files from a central repository, because all the source control happens locally. You don't lock files, because you don't have to worry about someone else trying to work on files on your own computer. You don't get a central source of truth—but you don't need

one. You are creating and sharing the truth as you work. If two contributors disagree on the truth, there is a way to decide which version to keep.

In this way of thinking, file versions become far less interesting than *changes*. If two people work on the same file, it's possible to see who made which changes and when. It no longer makes sense to think about the version of a file, because changes by different people can overlap chronologically. Instead of keeping score on file versions, Git tracks changes. When you roll forward or backward in time, or if you switch branches, Git applies the changes to show you the correct "versions" of the files. In fact, changes can be lifted from one point in time and "replayed" elsewhere.

There is usually a central repository, but it exists only to keep everyone in sync. Each contributor's own local repository is the source of truth. If the central repository were destroyed, it could be restored in minutes by anyone who had downloaded the most recent changes.

To sum up:

- Git tracks changes, not files.
- Source control happens on your computer, not a server.
- You are the source of truth.

Once you get used to thinking about changes rather than files, everything else about Git becomes easier to understand. The next section gives a little more detail about how Git works.

How Git Works

Git doesn't care about files. Git cares about *changes*. As you add, edit, and delete files, Git tracks the changes you're making. From time to time, you create a *commit*, a snapshot that groups a bunch of changes together and gives you a point in time you can roll back to if needed.

With Git, all you need to do is:

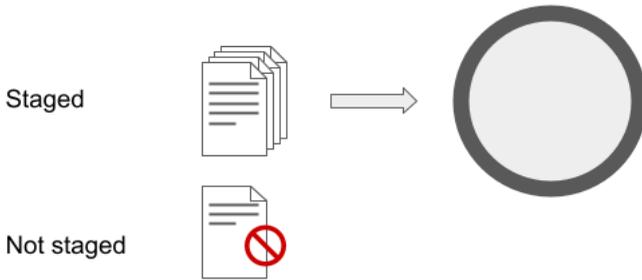
- Make changes.
- Tell Git which ones to track - this is called *staging*.
- Track them - this is called *committing*.

When you commit your work to Git, you aren't telling Git about new versions of files. You are telling Git about a group of changes you've made since the last commit. Editing or adding a file is a change—and so is deleting a file.

Git doesn't just grab every change, because it doesn't know which changes you intend to keep in source control. You might decide to commit at a certain point, but there's one file that's not really ready to commit yet. Or there might be a file that's in your directory by mistake, or some output or something that you don't need to track. Git gives you the opportunity to tell it what changes to track.

Committing is really two steps:

1. Stage: choose what to include in a group of changes.
2. Commit: save the changes in Git.



When people talk about a staging area, they mean the list of changes you are preparing to formalize in Git. The add command really means *add this change to the list*. Remember: you're tracking changes, not files. When you delete a file from Git, that creates a change that you must add!



As you go along, working and committing, you end up with a history of commits. You can roll back to any commit, if you need to.



When you roll backward or forward in time or switch branches, Git is just pointing to a particular commit that represents the way things were on that branch at that time. Git automatically applies all the appropriate changes to your working directory. The working directory becomes the correct view of that version of the files.

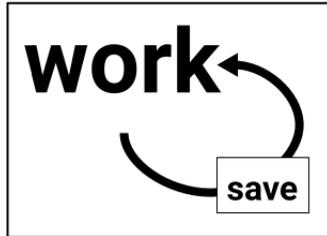
If you're collaborating with a larger team, it's useful to keep the main branch clean and always releasable while everyone's working. With Git, you can create working branches to keep people's work separate until it's ready to merge to the main branch. You can create working branches just for your own tasks, or a smaller team can have one or more long-standing working branches. In either case, the concept is the same. It's a series of commits independent from the main branch (or any other branch).



When it's time, those changes get merged back into the main branch (or whatever development branch your organization uses). Usually this happens on the central repository, also known as the *remote repo*, where people can have a chance to review it. When you work with Git, you frequently pull down the latest changes to your local repository (or repo), to make sure your work stays in sync with what everyone else is doing. When you're ready to share your work with others, you push your changes up to the remote repo so they can review your work. Once it's approved, you can merge it.

Working with Git

Let's start with the most familiar workflow: working on files and saving them, without using source control.

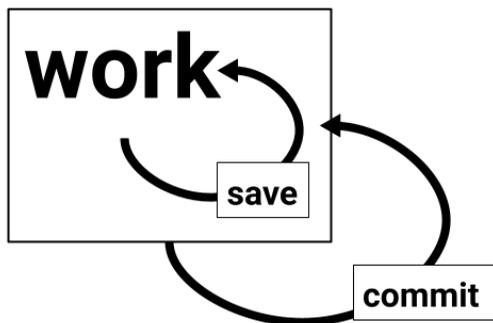


By adding a few steps to this basic way of working, Git lets you perform source control, create branches, and share your work with others.

Basic Source Control with Git

All the source control happens on your own computer. All you need to do is tell Git from time to time which changes you want tracked. Just as *save* writes your files to disk, *commit* is like "saving to Git." Before committing, of course, you have to tell Git which changes you want to save. This is called *staging* your changes.

When you stage and commit changes, you've done all you need for

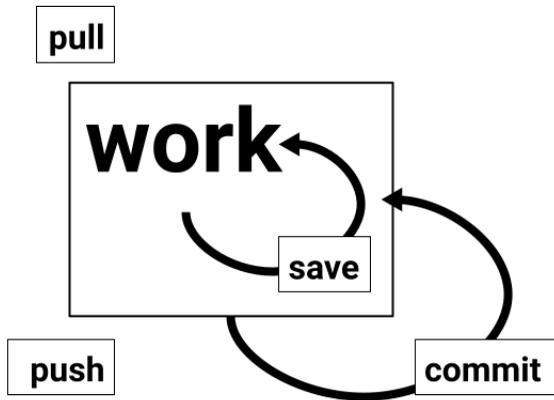


source control on your local computer. You can see the whole history of your commits, and can roll back to any of them to see earlier versions of your work.

Sharing Your Work

To share your changes with others, you *push* your changes to a central repository called the *remote*.

Everyone else who is working on the project *pulls* the latest changes from the remote to their local repo. Later, they push their own changes, and so on. That keeps everyone in sync, and also means that everyone's personal copy of the project (their local repo) is as much a source of truth as the central repo. You could blow away the central repo and life would just go on.



When you start working on a new task, you pull to make sure you're up to date. When you are ready to show your work to others, you push.

NOTE Pulling consists of two operations: *fetching* changes from the remote repository, and *merging* them with your local repository. When you pull, Git automatically fetches and merges changes from the remote repo. It's more common to pull than it is to fetch and merge separately.

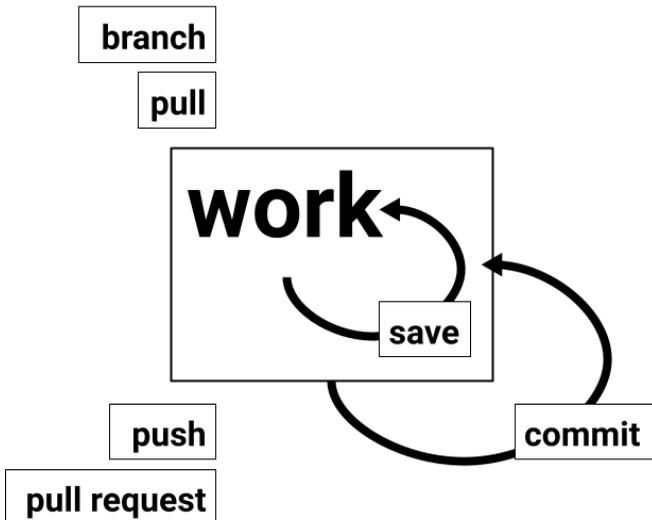
Branching

If there are more than a few people working on the same project, it's a good idea to separate people's work so they don't cause problems for each other. Git does this by *branching* work into different commit histories.

NOTE The main branch is usually called `main` or `master`. This book uses `main`, which is the newer name.



There are many branching strategies, from very simple to very complicated. Unless you have a reason to do something complex, you should keep your branching strategy simple. The *GitHub Flow* is one simple and effective branching strategy (see “The GitHub Flow” on page 18). In this workflow, instead of merging directly, you create a *pull request* so that others can review your work.



Git Centralized Workflow

If you are working with a small team (or by yourself), you can use an even simpler workflow called the Git Centralized Workflow. You should only use this branching strategy if:

- You don't need to separate people's work from each other.
- You don't need a review process.
- People almost never work on the same files at the same time.

The Git Centralized Workflow does not provide all of the protections that the GitHub Flow offers, but it is very easy for non-technical people or people who don't want to learn about Git. It is not well suited for larger projects, production source code, or tight collaboration within teams.

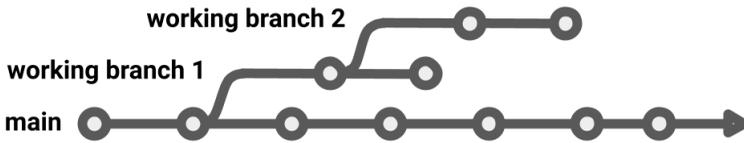
With a graphical Git client and a WYSIWYG Markdown editor, the centralized workflow can be an effective way for content creators, managers, and engineers to collaborate on non-production content such as specifications, planning documents, newsletters, internal documentation, and the like.



In this workflow, everyone works on the main branch. Before working, pull. While working, stage and commit. When you've completed a task, push. You don't need to create a pull request, and there is no working branch to merge. You just push your changes to the main branch and resolve any conflicts.

Branching Problems

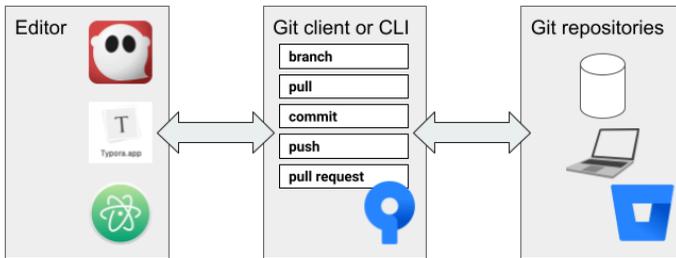
An overly complex branching strategy can cause problems. In this example, convoluted branching leads to shipping something that shouldn't have gone out.



The diagram above shows a branch (working branch 2) created from another branch (working branch 1). By itself, this is not a problem. As long as working branch 2 is merged back into working branch 1, all is well. But if you merge working branch 2 directly to the main branch, it will cause the commit it was based on—part of working branch 1—to become part of the main branch. If the main branch is then released, unapproved changes from working branch 1 can be included in that release.

How It All Fits Together

As you work with Git, you can choose whatever editor, branching strategy, and Git client make sense to you. You edit files, saving from the editor, then use your Git client or the command line to stage, commit, pull and push.



Here are a few examples that show what it might be like for several people to work together on some files using Git and other tools. For these examples, the repo is set up and has files in it. The people are continuing work on a project already in progress, using the GitHub Flow.

Graphical Git

One person has chosen to create some new work on the project using a graphical desktop client:

1. In the desktop Git client, pull and create a branch.
2. Work on files in the project.
3. Stage and commit, typing a short commit message.
4. Push the new branch and its changes to the remote repo.
5. Create a pull request online.

Git on the Command Line

A collaborator wants to work on the same files. That means pulling and checking out the same branch where the files are, editing the files, and pushing up to the remote repo. This particular collaborator prefers Git on the command line:

1. Use `git pull` to get the latest changes, including the branch the content author pushed.
2. Type `git checkout` and the branch name to switch to the new branch.
3. Work on files in the project.
4. Type `git commit` and write a commit message.
5. Pull again to make sure that there are no merge conflicts.
6. Push from the local working branch to the remote.
7. In the output of the `git push` command, find the URL to create a pull request online.

Git Online

A contributor wants to make small changes to a file without the overhead of pushing from their local machine:

1. Go to the repository on the Git host.
2. Navigate into the directory containing the file.
3. Click the filename.
4. Click the **Edit** button (it sometimes looks like a little pencil).
5. Commit the changes. If you don't need the changes reviewed, commit directly to the main branch. Otherwise, create a pull request.

Commit changes

Commit message	<input type="text" value="new-file.md edited online with Bitbucket"/>
	<input checked="" type="checkbox"/> Create a pull request for this change
Branch name	<input type="text" value="Peter-Conrad/newfilemd-edited-online-with-bitbucket-1602524184517"/>
Reviewers	<input type="text" value="Add reviewers..."/>

Reviewing a Pull Request

A reviewer needs to approve the work. You do this online, in the web interface provided by the Git host:

1. Go to the pull request.
2. Comment on any lines you'd like the author to change.
3. Once the work is acceptable, approve the pull request.
4. When enough reviewers have approved the work, the author can merge the pull request.

GET STARTED

Get Set Up with Git

Follow these steps to sign up with a Git host, install Git tools, and set up your first repository.

Get Git

Get a Git account on a host, then install a Git client.

1. Sign up with a Git host such as Bitbucket or GitHub. Look for instructions on the host's website:
 - <https://bitbucket.org/>
 - <https://github.com/>
2. Install a Git client such as Sourcetree or GitHub Desktop. Look for instructions on the client's website:
 - <https://www.sourcetreeapp.com/>
 - <https://desktop.github.com/>

Installing a Git client installs Git too. On the command line, you can check whether Git is installed on your computer by typing:

```
git --version
```

If Git wasn't installed for some reason, install Git on your computer using the instructions at <https://git-scm.com/>.

TIP You can use any Git client with any Git host, but some hosts and clients go together. For example:

- Bitbucket and Sourcetree
- GitHub and GitHub Desktop

Set up a Repository

A *repository* is where you keep your work. You'll need a *local repository* where you commit changes on your computer, and a *remote*

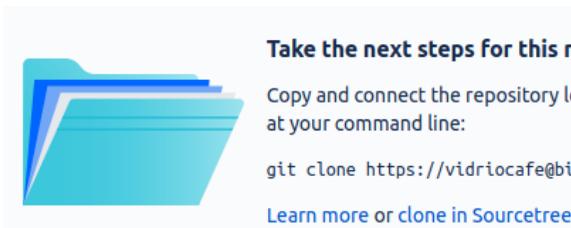
repository online where you collaborate with others. A straightforward way to create both is to set up a repository with an online host and then *clone* it (create a local copy). Your collaborators clone the repository to their own computers, so everyone can keep in sync by pushing and pulling changes.

Set up a Repository with Bitbucket and Sourcetree

1. Log on to Bitbucket.
2. Click the Create button (the + in the left sidebar):



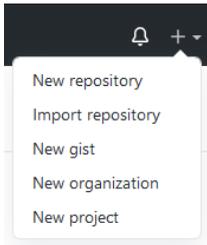
3. Under Create, select **Repository**.
4. Type a repository name, set it as a public repository, and click **Create repository**.
5. Click Clone in Sourcetree.



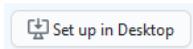
6. Choose a folder on your computer for the local copy of the repository, and click **Clone**.

Set up a Repository with GitHub and GitHub Desktop

1. Log on to GitHub.
2. Click the plus sign and select **New repository**:



3. Type a repository name, set it as a public repository, and click **Create repository**.
4. Click **Set up in Desktop** to open the repository in GitHub Desktop:



5. Choose a folder on your computer for the local copy of the repository and click **Clone**.

Result

It might not look like much has happened, but you now have:

- Git running on your computer
- A repository at an online Git host
- A local copy of the repository on your computer

Tutorials

These short tutorials show you how to get started working with Git. If you want more information about any of the steps, see “Basic Git Operations” on page 79.

TIP Before working on these tutorials, make sure you’ve completed all the steps in “Get Set Up with Git” on page 29.

Tutorial: Day to Day Work in a Git Client

This tutorial shows how to create a working branch, make changes to a few files, and commit. If you’re following the GitHub Flow, this is how you begin working on a new task.

TIP These steps are designed to work with either GitHub Desktop or Sourcetree, but you can use any Git client you choose. The names of the commands are similar but vary slightly in different Git clients.

Create a Working Branch

The first step is to create a working branch based on the main branch.

1. In your Git client, look at the tab at the top of the screen to see the name of the repository you created in the previous chapter.
2. Look under **Current branch** or **Branches** to make sure you’re on the main branch.
3. Select **Repository > Pull** to get the latest changes from the remote repository.

4. Make a new branch.

GitHub Desktop:

- Click Current branch, then New branch.

Sourcetree:

- Click **Branch**.

5. Type a descriptive name and click **Create Branch**.

- If you're using GitHub Desktop, click **Publish Branch**.

Commit Some Changes

Once you have a working branch, you can open your favorite editor and work on any files in the project.

TIP Before continuing, create a file and save it. For example, you could create a file called `hello.txt` with the contents "Hello, world!"

If you want to see how multiple files look in a pull request, create more than one file.

Stage and commit:

1. Look in your Git client to see the list of changed files.
2. Click a filename to see a *diff*, which shows the added and removed lines in that file.
3. If you're satisfied with all the changes, you're ready to commit.
4. Type a summary for the commit, then click **Commit**.
 - If you're using Sourcetree, click **Stage All** first.

Push and Create a Pull Request

When you're ready to share your work with others, push your branch and create a pull request.

1. Pull from the main branch.

GitHub Desktop:

- Click **Fetch origin**.
- If the button changes to **Pull origin**, click once more.

Sourcetree:

- Click **Pull**.

2. Push your branch.

GitHub Desktop:

- Click **Push Origin**.

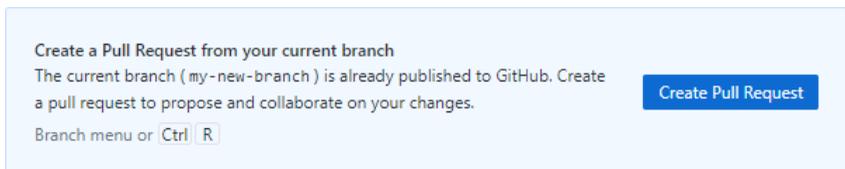
Sourcetree:

- Click the **Push** button.
- Make sure the local and remote branches are as expected, then click **Push**.

3. Create the pull request.

GitHub Desktop:

- After you push, the banner with the Push button changes to read "Create a pull request from your current branch." Click **Create Pull Request**.



Sourcetree:

- Click **Repository > Create pull request**.
 - In the dialog that appears, click **Create Pull Request on Web**.
4. The browser opens a page with a form for creating a pull request. Type a description.
 5. Add reviewers to the pull request. Since no one else has access to the repository yet, you'll just type your own username.
 - If you're using GitHub Desktop, you need to click the gear next to **Reviewers** to add reviewers.
 6. Click **Create pull request**.

Take some time to explore the pull request. It shows the description you typed, all the changes you made in each file, and the reviewers.

As people review the pull request, they can leave comments. When they are satisfied with your changes, they can approve the pull request.

Review a Pull Request

When someone submits a pull request for review, they're giving teammates the opportunity to see and comment on every change. Put on your reviewer hat and look at your pull request again.

1. Scroll down to see the lines that have changed in the files.
2. Move the mouse cursor over one of the lines in the file. Click the colored plus sign button that appears.

3. Type a comment and click **Save**.

You can continue adding comments in this way.

Approve a Pull Request

When the person who submitted the pull request (probably you, at the moment) has made the appropriate improvements, you can approve the pull request:

- Click **Approve**.

Merge a Pull Request

When enough reviewers have approved your pull request, you can merge:

- Click **Merge**.

Your team determines how many approvals are enough. If you are working by yourself, of course, you can approve your own pull request and then merge it—or just use the Centralized Git Workflow, which doesn't require pull requests.

If you have kept up with pulling the changes into your local branch and solving merge conflicts there, it will go smoothly. If not, you might need to pull, resolve, commit, and push once more before merging.

Tutorial: Working on the Command Line

Here are a few things to try on the command line.

Switch to a Working Branch

1. Make sure you're on the main branch:
`git checkout main`
2. Pull, to make sure you have the latest changes:
`git pull`
3. Switch to a working branch.
 - If you have completed "Tutorial: Day to Day Work in a Git Client" on page 33 and want to use your existing branch, switch to it by typing `git checkout {branch-name}`.
Example:
`git checkout my-working-branch`
 - Otherwise, create a new branch and switch to it with `git checkout -b {new-branch-name}`. Example:
`git checkout -b new-branch`
4. Type `git status` to make sure you're on the expected branch.

Commit Some Changes

Once you have a branch to work in, you can open your favorite editor and work on any files in the project.

TIP Before continuing, create a file and save it. If you've done "Tutorial: Day to Day Work in a Git Client" on page 33, just open one of your files and make changes.

Stage and commit:

1. Type `git status` to see your staged and unstaged changes.
2. Use `git diff` to see the lines that have changed:
 - `git diff` by itself shows the changes in all the files in the project.
 - `git diff {filename}` shows the changes in the specified file. You can use directories and wildcards to specify groups of files.
3. Type `git add .` to stage all the changes.
4. Type `git commit -m` along with a commit description. Example:

```
git commit -m "Make changes to my-file.txt"
```

Push and Create a Pull Request

When you're ready to share your work with others, push your branch and create a pull request.

1. Make sure you're on the correct branch and you have no uncommitted changes:

```
git status
```
2. Pull, to make sure you have the latest changes:

```
git pull
```

3. Type the `git push` command, specifying the remote (usually `origin`) and the branch. Example:

```
$ git push origin my-working-branch
```

```
Enumerating objects: 14, done.
Counting objects: 100% (14/14), done.
Delta compression using up to 4 threads
Compressing objects: 100% (10/10), done.
Writing objects: 100% (10/10), 4.39 KiB | 1.10
MiB/s, done.
Total 10 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), completed
with 1 local object. remote:
remote: Create a pull request for 'my-working-
branch' on GitHub by visiting: remote:
https://github.com/pconrad-fb/markdown/pull/new/my-
working-branch
remote:
To https://github.com/pconrad-fb/markdown.git
* [new branch] my-working-branch -> my-working-
branch
```

4. Take note of the URL in the next line after the `Create a pull request` line in the output. Copy and paste this URL into a browser.
5. Follow the instructions on the screen.

Once you've created a pull request, the steps for reviewing, approving, and merging are the same as the corresponding sections above.

Stay out of Trouble

There are several common ways to get into trouble with Git. Here are a few tips to help keep you working smoothly.

Choose the Correct Branch

When you're working on a project, it isn't always easy to tell which branch you're on just by looking at the files. To avoid making changes on the wrong branch, double-check before you start working. If you end up doing some work (or committing) on the wrong branch, there are ways to fix the problem, but it's better to avoid that mistake in the first place.

Pull Often

Before you start working, pull from the main branch (or whatever branch you plan to merge onto later). This helps keep your files in sync with the changes that others are pushing. If someone else's work conflicts with yours, you'll know more quickly, which can make the merge conflicts easier to solve.

Think before Committing, Pushing, or Branching

Before committing, make sure you're on the correct branch and looking at the right changes. Before pushing, make sure you're sharing the changes you mean to and that you're pointed at the correct branch on the appropriate remote repository. Before branching, make sure you're starting from the right branch—usually `main` or `master`, if you're using the GitHub Flow.

Use a Simple Branching Strategy

The more complicated the branching strategy, the more difficult it is to remember what goes where. Starting from the GitHub Flow is a good idea, unless your team already uses something else. If you always branch from `main` and merge to `main`, it's easy to know where your changes will end up.

Be Watchful

A Git client such as GitHub Desktop or Sourcetree keeps relevant information in front of you all the time. On the command line, you can use `git status` to see what branch you're on and any staged or unstaged changes. Keep an eye on this information as you work. It'll help you stay out of trouble, and make it easier to fix problems when they occur.

Write Good Commit Messages

Commits are the heart of Git, and your commit messages are the soul of documenting changes. Many people have written great advice about how to craft a good Git commit message.

If your changes are simple and obvious, you can get by with one line.

If you want to provide more information, write a proper commit message with a subject (the first line) and body (the other lines).

- In GitHub Desktop, the Summary (required) is the subject and the Description (optional) is the body.
- In Sourcetree or on the command line, the subject is the first line and the body is all the subsequent lines.

On the command line, using `git commit` without the `-m` option opens a text editor so you can type a proper commit message.

The subject line summarizes the change. Use an imperative sentence, 50 characters or less, with no punctuation at the end. The subject finishes the sentence *This commit will...*

After the subject, leave a blank line before the body (in GitHub Desktop, the separation of Summary and Description fields does this for you).

The body gives more context about the change. In the body, explain why the change is necessary, how it solves the problem, and what side effects it might have. Manually wrap body lines at 72 characters. You can use Markdown in the body for bullet points and other light formatting.

If you are tracking issues related to the commit, list the issue numbers at the bottom of the body.

Example

Add a short chapter about commit messages

Good commit message style was a missing feature from this manuscript. In the interest of providing complete information, I've added a guide to writing good commit messages. Most of this information came from the web, where almost all the guidelines seem to agree from one site to another.

This section might make readers think they aren't allowed to use a one-line commit message for a simple change, but that's less risky than not telling how to provide complete, proper documentation of changes.

This commit resolves issue #ED-678: No guidance on commit messages.

FUN WITH GIT

Command Line Tips and Tricks

If you prefer to work on the command line, these techniques can be helpful.

Authenticate Git on the Command Line

Git clients store browser-based credentials so they can authenticate you automatically. Git authentication on the command line is different.

The first time you use Git on the command line, it might ask you to complete authentication in your browser. If that happens, a browser window opens and Git helps you create a token to authenticate you every time you issue a command to the remote repository:

```
$ git push
```

```
info: please complete authentication in your browser...
```

Git might ask you for a username and password instead:

```
$ git push
```

```
Username for 'https://github.com':
```

If that happens, you'll need your Git host username and either a login password or a token, which works the same as a password. A token is a long string of random characters that is safer and more difficult to crack than most passwords.

Your Git host has instructions for creating an access token, if needed.

NOTE When you create the token on the Git host, copy it to the clipboard and paste it somewhere safe immediately. Once you dismiss the screen that shows your new token, there is no way to display it again.

Cache Your Credentials

If you don't want to type your username and token (or password) every time you run a Git command, you can cache your credentials by telling Git which *credential helper* to use.

The first time after you configure a credential helper, you'll need to supply your credentials once more. After that, you're authenticated automatically.

Linux

On Linux, there are three ways to cache or store your credentials.

The Credential Memory Cache

The cache credential helper keeps your credentials in memory. You configure it by typing:

```
git config --global credential.helper cache
```

For safety, your credentials are kept for only 15 minutes (900 seconds) by default. You can specify a different timeout with the `--timeout` parameter.

Example

Set the timeout to one hour (3600 seconds)

```
git config --global credential.helper 'cache --  
timeout=3600'
```

The Store Credential Helper

You can use the store credential helper to store your credentials in plain text in the `~/ .git-credentials` file.

WARNING This is a security risk. If anyone gains access to your computer, they can find and use your credentials to access your remote repo.

If you're sure you want to store your credentials in plain text, configure the store credential helper by typing:

```
git config --global credential.helper store
```

The Libsecret Credential Helper

The libsecret credential helper is more convenient than cache but takes a few steps to set up:

1. Install the libsecret source package. Example:

```
sudo apt-get install libsecret-1-0 libsecret-1-dev
```

2. Change to the directory where the source is installed:

```
cd /usr/share/doc/git/contrib/  
credential/libsecret
```

3. Build the code:

```
sudo make
```

4. Configure libsecret as the credential helper:

```
git config --global credential.helper  
/usr/share/doc/git/contrib/credential/  
libsecret/git-credential-libsecret
```

macOS

- On macOS, use `osxkeychain` to store your credentials in the OS X keychain:

```
git config --global credential.helper osxkeychain
```

Windows

- On Windows, use `wincred` to store your credentials in the Windows Credential Manager:

```
git config --global credential.helper wincred
```


Examine the Past

There are two main ways to see a history of commits:

- The `git log` command shows the current HEAD and the commit history leading up to it.
- The `git relog` command shows all commits HEAD has ever pointed to, without regard to branches.

If you want to see what you've done recently in the current branch, use `git log`. If you want to see everything that's ever happened in your local repo, use `git relog`.

NOTE Both commands show the *hash*, a number that uniquely identifies each commit.

Git Log

Git log provides a number of different ways to look at the history of the current branch. The simplest form of the command gives somewhat verbose output:

```
$ git log

commit 971ccbd2536265c9683659c69b502e9dcbd8791e
(HEAD -> master, origin/master, origin/HEAD)
Author: Peter Conrad <stymied@gmail.com>
Date: Tue May 18 20:30:55 2021 -0700
```

Fix problems with the structure

```
commit d22db10aca037d1d8ce4322ad3e5d20b803f9ce9
Author: Peter Conrad <stymied@gmail.com>
Date: Tue May 18 18:23:24 2021 -0700
```

Add content from the slide deck

```
commit 423e781d81d51f69c36599cfc3fe818c1c40444d
Author: Peter Conrad <stymied@gmail.com>
```

Date: Fri Apr 9 18:14:36 2021 -0700

Edit Concepts chapter

commit 363df41ce2c6de19bd2b654b87a62cc011dd7dd4

Author: Peter Conrad <stymied@gmail.com>

Date: Sun Dec 27 19:48:12 2020 -0800

Rewrite Basics chapter

commit 39d2dfd6410b9e66ff2e1e8a87cf0115f4d0316b

Author: Peter Conrad <stymied@gmail.com>

Date: Mon Dec 21 17:59:48 2020 -0800

Finalize troubleshooting section

To see a more compact version of the log, use --oneline like so:

```
$ git log --oneline
```

```
971ccbd (HEAD -> master, origin/master,
origin/HEAD) Messing with a lot of stuff to try to
get the structure right
d22db10 Add content from the slide deck
423e781 Edit Concepts chapter
363df41 Rewrite Basics chapter
39d2dfd Finalize troubleshooting section
ce2500c Revert "This commit is a mistake!"
d9fa1d2 This commit is a mistake!
d555f66 Commit some lovely changes
0856f2c Create initial structure for examples
1b122ee Fix up Git Concepts section
d62b067 Improve Troubleshooting section
99f68ba Fix errors in glossary
0086a00 Draft overview
75d01c9 Fix build errors
1c4d847 Make headings consistent
5db7b41 Fix screenshots
6c1c5bf Create overall structure
4667731 Write outline
```

NOTE If you prefer to see the full commit hash instead of just the first seven digits, you can use `--pretty=oneline` instead of `--oneline`.

View the Branch Structure

Along with the commits, Git can show you a diagram of branches and merges using the `--graph` and `--decorate` options:

```
$ git log --oneline --graph --decorate

* b786678 (HEAD -> working-on-some-stuff,
origin/working-on-some-stuff) Merge branch
'working-on-some-stuff' of
https://bitbucket.org/vidriocafe/bitbucket-test-
repo into working-on-some-stuff
|\
| * 69d234a Edit my-first-file.md
* | c911007 Edit Introduction
|/
* 062e771 Update my-first-file.md
* fcb9dda Merged in a-new-branch (pull request
#1)
|\
| * f57b781 Add my first file to the repo
|/
* cbdaa65 Initial commit
```

Compare Branches

To see commits that exist in one branch but not another, type the branch names separated by two dots. The syntax is:

```
git log {branch1}..{branch2}
```

This shows the changes that exist in `branch2` but not `branch1`.

You can use options like `--oneline` or `--pretty=oneline` to format the output.

Example

Show the commits that are in the `working-on-some-stuff` branch but not in `main`:

```
$ git log main..working-on-some-stuff --pretty=oneline
b7866782188fd66da1ccb784da721972349fdb54 (HEAD ->
working-on-some-stuff, origin/working-on-some-stuff)
Merge branch 'working-on-some-stuff' of
https://bitbucket.org/vidriocafe/bitbucket-test-repo
into working-on-some-stuff
c9110074618e7dab1c412a8f73d25487a4b0b0ae Edit Intro
69d234a992e860ddb9f0eaa246b4cc213fcc18a2 Edit my-first-
file.md
```

Show the Last Several Commits

You can use the two-dot syntax to show the last few commits. For example, `git log HEAD~3..HEAD` shows the last three commits on the current branch.

Another way to do the same thing is: `git log -n 3`.

Example

Show the last three commits:

```
$ git log -n 3 --pretty=oneline

b7866782188fd66da1ccb784da721972349fdb54 (HEAD ->
working-on-some-stuff, origin/working-on-some-stuff)
Merge branch 'working-on-some-stuff' of
https://bitbucket.org/vidriocafe/bitbucket-test-repo
into working-on-some-stuff
c9110074618e7dab1c412a8f73d25487a4b0b0ae Edit
Introduction
69d234a992e860ddb9f0eaa246b4cc213fcc18a2 Edit my-first-
file.md
```

Git Reflog

The `git reflog` command doesn't have the formatting options that `git log` has. It just shows an ordered list of all the commits that have happened on your local repository:

```
$ git reflog
```

```
971ccbd (HEAD -> master, origin/master, origin/HEAD)
HEAD@{0}: commit: Messing with a lot of stuff to try to
get the structure right
d22db10 HEAD@{1}: commit: Add content from the slide
deck
423e781 HEAD@{2}: commit: Edit Concepts chapter
363df41 HEAD@{3}: commit: Rewrite Basics chapter
39d2dfd HEAD@{4}: commit: Finalize troubleshooting
section
ce2500c HEAD@{5}: checkout: moving from
d555f66f942ee676a4401223884defe82a95fbb8 to master
d555f66 HEAD@{6}: reset: moving to HEAD~1
d9fa1d2 HEAD@{7}: checkout: moving from master to
d9fa1d2
ce2500c HEAD@{8}: checkout: moving from
d9fa1d294cb83cc751b3780a07dfce11f66fe2fa to master
d9fa1d2 HEAD@{9}: checkout: moving from master to
d9fa1d2
ce2500c HEAD@{10}: revert: Revert "This commit is a
mistake!"
d9fa1d2 HEAD@{11}: commit: This commit is a mistake!
d555f66 HEAD@{12}: commit: Commit some lovely changes
0856f2c HEAD@{13}: commit: Create initial structure for
examples
1b122ee HEAD@{14}: checkout: moving from test-branch to
master
1b122ee HEAD@{15}: checkout: moving from master to
test-branch
1b122ee HEAD@{16}: commit: Fix up Git Concepts
```

Save Some Keystrokes

If you find yourself typing lengthy Git commands often, you can use aliases to save time. The `git config` command lets you create a shortcut for the part of the command after the word `git`. To use the shortcut, type `git` and the alias.

Examples

Create the `last3` alias for a one-line listing of the most recent three commits:

```
git config --global alias.last3 "log -n 3 --oneline"
```

Use the `last3` alias:

```
$ git last3
```

```
b7866782188fd66da1ccb784da721972349fdb54 (HEAD ->
working-on-some-stuff, origin/working-on-some-stuff)
Merge branch 'working-on-some-stuff' of
https://bitbucket.org/vidriocafe/bitbucket-test-repo
into working-on-some-stuff
c9110074618e7dab1c412a8f73d25487a4b0b0ae Edit
Introduction
69d234a992e860ddb9f0eaa246b4cc213fcc18a2 Edit my-first-
file.md
```

Compare Branches or Commits

As you probably recall, `git diff` shows all the uncommitted changes since the last commit. To see the currently staged changes, you can use `git diff --cached` instead.

You can use two dot syntax to compare two commits or two branches. The syntax is:

```
git diff {since}..{until}
```

By specifying a file or directory name, you can limit `git diff` to show changes on a particular file or directory.

Examples

Show the difference between three commits ago and one commit ago:

```
git diff HEAD~3..HEAD~1
```

Use hashes to compare two commits directly:

```
$ git diff be753da..69d234a
```

```
diff --git a/my-first-file.md b/my-first-file.md
index 0de9f54..007371f 100644
```

```
--- a/my-first-file.md
```

```
+++ b/my-first-file.md
```

```
@@ -1,9 +1,9 @@
```

```
 # My First File
```

```
-This is the first file I am adding to my new
repository.
```

```
+This is the first file I added to my new repository.
```

```
 - It is Markdown
```

```
 - It is simple
```

```
-- It is new
```

```
-
```

```
-Later I will make changes to this file.
```

```
+-- It is not so new anymore.
```

```
+
```

+This is when I make changes to this file. Later has come and gone!

Show the changes in my-first-file.md between the master and working-on-some-stuff branches:

```
$ git diff master..working-on-some-stuff my-first-file.md
```

```
diff --git a/my-first-file.md b/my-first-file.md
index b708c3a..007371f 100644
--- a/my-first-file.md
+++ b/my-first-file.md
@@ -4,6 +4,6 @@ This is the first file I added to my
new repository.
```

- It is Markdown
- It is simple
- It has been around the block!
-
- + - It is not so new anymore.
- +

This is when I make changes to this file. Later has come and gone!

Checkout Old Commits

The `git checkout` command lets you go back to any commit by its hash. The syntax is:

```
git checkout {hash}
```

You can use either the long or short version of the commit number displayed by `git log`. Consider the following output:

```
$ git log --oneline --graph --decorate

*   b786678 (HEAD -> working-on-some-stuff,
origin/working-on-some-stuff) Merge branch 'working-on-
some-stuff' of
https://bitbucket.org/vidriocafe/bitbucket-test-repo
into working-on-some-stuff
|\
| * 69d234a Edit my-first-file.md
* | c911007 Edit Introduction
|/
* 062e771 Update my-first-file.md
```

If you wanted to go back in time to the state of your files when you edited `my-first-file.md`, you can checkout that commit with `git checkout 69d234a`.

NOTE If you checkout an old commit and want to create changes, you must create a branch first.

When you check out an old commit, your HEAD (the "you are here" pointer) no longer points to the tip of a branch. This is called a detached HEAD. If you make a commit without starting a new branch, the only way to get back to that commit later is to know its hash.

WARNING Don't commit with a detached HEAD.

Label Commits with Tags

Commit hashes are a reliable way to identify commits, but they are not easy to read. To label a commit so that you easily can find it later, you can add a commit tag. There are two kinds of tags, with different syntax:

- Lightweight tag:
`git tag {label}`
- Annotated tag:
`git tag -a {label} -m "{description}"`

A lightweight tag is just a label that points to a commit. An annotated tag is a Git database object that points to a commit, including a message and other information. Some people use annotated tags for commits they intend to push from, and lightweight tags for other commits.

You can use a tag name in any command that takes a commit hash.

Examples

Add an annotated tag to the most recent commit before pushing:

```
git tag -a release-1.0 -m "First release of the product"
```

Add a lightweight tag to a local commit for your own future reference:

```
git tag final-review
```

Show the changes between the commits tagged `final-review` and `release-1.0`:

```
git diff final-review..release-1.0
```

Share Tags

To share tags, you must push them explicitly.

Examples

Push a single tag:

```
git push origin my-tag
```

Push all tags:

```
git push origin --tags
```

Push annotated tags relevant to the commits being pushed:

```
git push origin --follow-tags
```

Clean Up

Keeping your repository clean helps avoid confusion. Here are three things you can do to keep things in order.

Ignore Irrelevant Files

If your working directory contains test output, compiled files, or other files you don't want included in source control, you can tell Git to ignore them. This keeps Git from tracking the files, preventing them from being included in commits or pushes.

The top level of your local repository working directory contains a file named `.gitignore` that shows which file types or filename patterns to ignore.

Here's a sample excerpt from `.gitignore` showing wildcards:

```
# Applications
*.app
*.exe
*.war

# Large media files
*.mp4
*.tiff
```

You can exclude a specific file or directory by name, or use wildcards (*) to exclude files and directories with similar names. The wildcards match any text. That is, `*.exe` means "any file that ends with `.exe`."

Delete Old Branches

After you're finished with a branch, you can use `git branch -d` to delete it. Deleting a branch doesn't delete the files or the changes you've made to them. It just means that branch is closed and you can't work on it anymore. Deleting a branch locally doesn't delete the remote (pushed) copy of the branch, but can flag the commit history of the local branch for deletion.

This command fails if the specified branch hasn't been merged to the current branch yet, including on the remote repo. If you are absolutely

sure you want to delete a branch without merging it—to abandon it, in effect—you can use a capital `-D` instead to force deletion.

Example

Delete the local branch `working-on-some-stuff`:

```
git branch -d working-on-some-stuff
```

Rebase

Rebasing means moving an entire branch by changing the commit that it's based on. It's a controversial practice. Some people feel that rebasing rewrites the true commit history, while others feel it cleans up history and makes it easier to read. Both opinions are true. Rebasing can be useful if you've made one or more commits in the wrong place and need to move them.

Rebasing can also be dangerous. If other people have based their work on some of your commits, rebasing those commits can make life painful for them.

WARNING Do not rebase commits outside your repository or commits that others might rely on.

If you are sure you want to rebase, here's how to do it.

1. Check out the branch you want to move:

```
git checkout the-working-branch
```
2. Use rebase to move it to the tip of another branch:

```
git rebase main
```
3. Use `git log` to verify that you did what you expected:

```
git log --oneline
```

You can also use an interactive rebase to choose which commits and changes to move. The command `git rebase -i {branch}` shows all the commits in the branch in a text editor, letting you move lines

around and change what happens to them. That level of surgery is too much for a book called *Git Doesn't Have to Be Hard*. You can find more information at <https://git-scm.com/book/> if you're curious.

NOTE If you lose track of a commit by rebasing carelessly, you can use `git reflog` to find the hash, then check it out. See “Checkout Old Commits” on page 59.

Move, Rename, or Delete Files

If you move, rename, or delete a file, Git needs to know whether to track that change. One way to do that is to move the file with the Finder or the File Explorer, then stage the change yourself. But there's an easier way. Git includes its own `mv` and `rm` commands to make it easier to manage files.

Git Move

You use `git mv` to move or rename files, groups of files, or directories. The syntax is:

```
git mv {from} {to}
```

Examples

Rename `bad-filename.txt` to `good-filename.txt`:

```
git mv bad-filename.txt good filename.txt
```

Move `good-filename.txt` into the directory `some-files`:

```
git mv good-filename.txt some-files
```

Git Remove

The `git rm` command removes files in one of two ways:

- `git rm` removes the file from Git tracking and deletes it.
- `git rm --cached` removes the file from Git tracking but doesn't delete it.

Either way, you can only use `git rm` on files that are unstaged, with no differences between the versions in your working directory and the tip of the current branch.

To remove a directory and all its contents, add the `-r` option.

NOTE Git tracks changes, not directories, so the `git rm -r` command won't work on an empty directory.

To safely see what `git rm` will delete before running the command for real, add the `--dry-run` option. This shows output of the files that Git would delete, but doesn't actually delete them.

Example

Un-track (but don't delete) everything in the `some-files` directory:

```
git rm -r --cached some-files
```

Pack Up

You can create an archive of your repository to share a copy with a reviewer who doesn't use Git. The simple syntax is:

```
git archive {branch or HEAD} --format={format} --output={path}
```

This creates an archive of all the files as they exist in the current HEAD, without any of the version control information.

You can also create a *bundle*, which includes the version control information, enabling the recipient to treat the bundle as if it were a remote repo. This is handy for air gapped environments or very large repositories that would be impractical to clone over a network. The simple syntax is:

```
git bundle create {path} --all
```

You can send the bundle file to a recipient, who can then clone it to create a local copy of your repo.

Examples

Create an archive of all the files in the working directory:

```
git archive HEAD --format zip --output archive-HEAD.zip
```

Create a bundle of the entire repository, including remote references:

```
$ git bundle create ../copy-of-test-repo.bundle --all
```

```
Enumerating objects: 45, done.
Counting objects: 100% (45/45), done.
Compressing objects: 100% (44/44), done.
Total 45 (delta 17), reused 0 (delta 0), pack-reused 0
```

Clone a repo from a bundle:

```
$ git clone copy-of-test-repo.bundle copy-of-repo -b master
```

```
Cloning into 'copy-of-repo'...
```

Receiving objects: 100% (45/45), 5.10 KiB | 1.70 MiB/s,
done.

Resolving deltas: 100% (17/17), done.

Create a Git Wiki

A Git repository comes with a wiki, where people can read and collaboratively edit documentation. You can create a wiki to document projects or code stored in the repository, or you can just use a repository for its wiki capability. Most of the time, you use Markdown to write content in a Git Wiki, but some hosts also support reStructuredText, AsciiDoc, or other lightweight markup.

NOTE A Git wiki is a second repository attached to your repository. You clone, pull, and push to the main repository and the wiki separately.

Git Wiki Structure

A Git wiki uses folders to organize files. The path to a file determines the URL where the content is displayed.

A simple directory structure might look like this:

```
Home.md
stuff/
  something.md
```

In that case, the URL to the content in something.md is: `/wiki/stuff/something`.

You set up your content URLs by placing the files in a corresponding directory structure in your wiki.

Set Up a Wiki

The easiest way to set up the wiki is by logging onto your Git host and adding it there.

Set Up a Wiki in Bitbucket and Sourcetree

1. In Bitbucket, in the **Repository settings**, find **Features**.
2. Click **Wiki**.
3. Select **Public wiki** and click **Save**.

Wiki

Create and edit a wiki from a web browser. [Bitbucket wikis](#) are DVCS repositories. You can clone the pages and edit them on your local system. For a private wiki, a user's access to the code repository also applies to the wiki. For a public wiki, anyone can view it even if the code repository is private.

Wiki No wiki
 Private wiki
Anyone with repository access can view the wiki. Only those with repository write access can edit the wiki.
 Public wiki
Anyone can view the wiki. Only those with repository write access can edit the wiki.

Editing Allow anyone with wiki access to edit the wiki
NOTE: Selecting this for a public wiki means anyone can edit the wiki.

Save

Set Up a Wiki in GitHub and GitHub Desktop

The GitHub documentation on wikis is helpful:
<https://docs.gitlab.com/ee/user/project/wiki/>

Here are the basic steps:

1. In GitHub, click the **Settings** button:



2. Scroll down to **Features** and select **Wikis**.

Features

<input checked="" type="checkbox"/> Wikis ✓
<input type="checkbox"/> Restrict editing to collaborators only <small>Public wikis will still be readable by everyone.</small>

Work with Content on the Host

If you just want to add a few pages to the wiki online, there's no more setup to do. Just go to your repository, click **Wiki**, and you'll see buttons for creating and editing pages.

To add a page in a new folder, make the folder part of the new filename. For example, creating a file called `morestuff/newpage.md` adds `newpage.md` in a folder called `morestuff`.

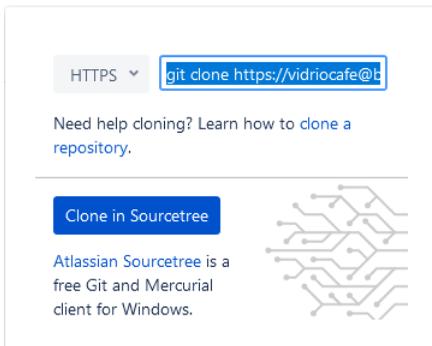
Work with Content Locally

There are advantages to working with wiki files locally, on your computer:

- It's much easier to add folders and move files around.
- You can work on it even when you're offline.
- You can use your editor of choice.
- Others can collaborate with you more safely.

Clone a Wiki in Bitbucket and Sourcetree

1. On Bitbucket, Click **Wiki**.
2. Click Clone wiki then Clone in Sourcetree.



3. Make sure the local path shows the directory where you want to clone the repository, and click **Clone**.

Clone a Wiki in GitHub and GitHub Desktop

1. On GitHub, click the **Wiki** button:



2. Copy the **Clone this wiki locally** URL.
3. In GitHub Desktop, click **File > Clone repository**.
4. Paste the URL, make sure the local path shows the directory where you want to clone the repository, and click **Clone**.

Clone a Wiki on the Command Line

1. Go to your online repository and click **Wiki**.
2. Copy the URL (or command and URL) for cloning the wiki repository.
3. On the command line, navigate to the directory where you want to clone the repository.
4. Use `git clone` and the URL to clone the repository. Example:

```
git clone https://my_name@bitbucket.org/  
my_name/markdown-stuff.git/wiki
```

Tutorial: Create Structured Wiki Content

Here's a quick tutorial that shows how to organize pages in the wiki.

Create Some Content Locally

1. Add a folder called `stuff`.
2. Using your favorite Markdown editor, make a file called `something.md` inside `stuff`, with the following contents:

```
# Something
```

```
Yes, there's *something* here! Now go  
[home](../Home).
```

3. Check that you now have a content structure that looks like this:

```
Home.md  
stuff/  
    something.md
```

4. Commit and push.

Take a Look

After you commit and push the changes, take a look at your online wiki on the host:

1. Go to your online repository and click **Wiki**.
2. View the page tree of the wiki. For example:
 - In Bitbucket, click the name of the wiki.
 - In GitHub, click **Pages**.

3. Navigate to the page you created.

Peter Conrad / Markdown Dreams / markdown

Wiki

[markdown](#) / [stuff](#) / [something](#)

Something

Yes, there's *something* here! Now go [home](#).

Updated 3 minutes ago

4. Try the home link.

Publish a Website

You can publish a static website directly on Bitbucket or GitHub. In either case, the website is a repository with a special name.

The structure of a static website is similar to Git Wiki structure. The directory path to each file determines its URL. There are two ways a static website's content is different from a wiki:

- The files are HTML, not Markdown.
- Each subdirectory should contain an `index.html` file.

You can add images and other embedded content to your website by uploading it to the repository and referring to it in HTML pages.

You can use JavaScript on your pages, but you can't do any server-side scripting such as PHP.

Publish a Website on Bitbucket

Bitbucket lets you create *workspaces* to organize your repositories. Each workspace can have its own website. After you create a workspace, you create a repository with the workspace name plus `bitbucket.io` to host the website.

1. On Bitbucket, click the Create button (the **+** in the left sidebar), then select **Workspace**.
2. Type a name and ID for the workspace, then click **Create**.
3. Click the **+** in the left sidebar again, then select **Repository**.
4. Enter `{workspace name}.bitbucket.io` for the **Repository Name**. For example, if the workspace is called `my-workspace`, name the repository `my-workspace.bitbucket.io`.
5. Enter a **Project** name.
6. Click Create repository.

7. Add an `index.html` file containing HTML. You can either edit and commit in Bitbucket directly, or edit, commit, and push from your local machine.
8. Browse to `{workspace name}.bitbucket.io` to see your first page.

Publish a Website on GitHub Pages

GitHub lets you create a single public website for your account, named with your username.

1. On GitHub, create a new public repository named `{username}.github.io`. For example, if your username is `pconrad`, name the repository `pconrad.github.io`.
2. Clone the repository to your local machine.
3. Working on your local machine, create an `index.html` file containing HTML.
4. Commit and push.
5. Browse to `{username}.github.io` to see your first page.

It can take some time for the website or any changes to appear.

REFERENCE

Basic Git Operations

The following sections describe how to perform the most common Git actions:

- Pull the latest changes from the remote repo.
- Create a branch.
- Stage and commit periodically as you work.
- Push your commits to the remote repo, sometimes creating a pull request.
- Approve and merge.

Each section shows steps for three environments:

- GitHub Desktop
- Sourcetree
- Git command line interface

Working with Git tools embedded in other applications is similar. Once you learn each procedure in your tool of choice, you'll be able to figure out how to do it elsewhere.

Pull

The Git pull command *fetches* content from the remote repository and automatically *merges* the changes with your local repository so that the current branch your local repository matches the latest version of the same branch on the remote (usually *origin*, where your copy of the project *originated* when you cloned it).

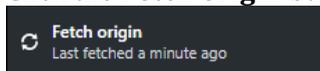
Pull in Sourcetree

1. Make sure you're on the right branch in the correct repository:
 - The bold text under **Branches** tells you the branch.
 - The tab at the top of the screen tells you the repository.
2. Select **Repository > Pull** or click the **Pull** button.

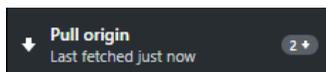


Pull in GitHub Desktop

1. Make sure you're on the right branch in the correct repository:
 - The bold text under **Current branch** tells you the branch.
 - The bold text under **Current repository** tells you the repository.
2. Select **Repository > Pull** or perform the following steps:
 - a. Click the **Fetch origin** button.



- b. Click the **Pull origin** button.



Pull on the Command Line

1. Make sure you're on the right branch:

```
$ git branch
* main
```

2. If necessary, switch branches with `git checkout` and the branch name. Example:

```
git checkout main
```

3. Check that your remote origin is set properly:

```
$ git remote -v
```

```
origin https://github.com/pconrad-fb/markdown.git
(fetch)
origin https://github.com/pconrad-fb/markdown.git
(push)
```

4. If necessary, set the origin with `git remote add origin` and the URL. Example:

```
git remote add origin https://github.com/pconrad-
fb/markdown.git
```

5. Type the `git pull` command.

Stage and Commit

Git knows when you make changes to your files. When you want to save those changes to Git, you must do two things:

- *Stage* them, which tells Git which changes you intend to keep.
- *Commit* them, which saves the changes.

First, you tell Git which changes to track. This is called *staging*. Since changes go with files, sometimes people think of it as staging the files themselves—but it’s really the changes that Git wants to know about. Remember, if you delete a file, that’s a change too.

Creating and saving a group of tracked changes in Git, or *committing* the changes, is a little like “saving changes to Git.” Git shows you the changes you’ve made so you can make sure you’re not accidentally tracking something irrelevant. When you commit, you type a little note describing the changes so that any collaborators (or you, in the future) know what you did.

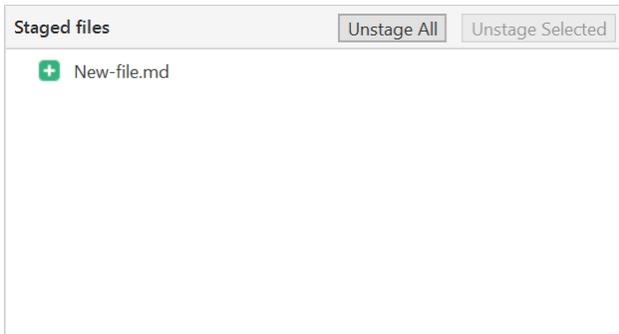
Stage and Commit in Sourcetree

In Sourcetree, you stage and commit your files in two operations.

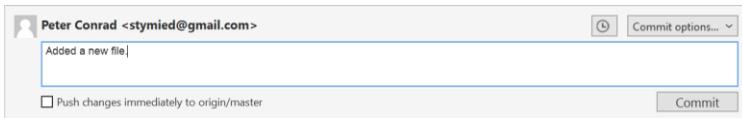
1. Make sure you’re on the right branch in the correct repository.
2. Look for the files you changed in the Unstaged files pane. Select the files you want to stage—in most cases, you can just click **Stage All**.



3. Make sure the correct files have moved to the Staged files pane.



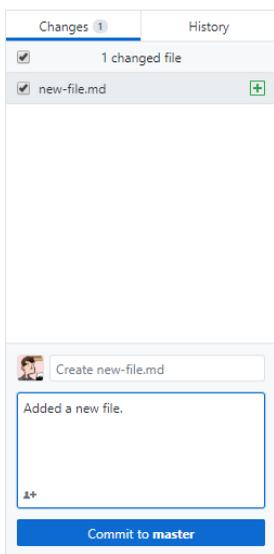
4. Type a short commit message and click **Commit**.



Stage and Commit in GitHub Desktop

In GitHub Desktop, you can stage and commit your files in one step.

1. Make sure you're on the right branch in the correct repository.
2. Look for the files you changed in the Changes tab. Unselect any files you don't want to change—most of the time, you can leave all the checkboxes checked.



3. Type a short commit message.
4. Make sure the **Commit** button refers to the correct branch ("Commit to my-working-branch," for example).
5. Click Commit to [branch].

Stage and Commit on the Command Line

1. Make sure you're on the right branch in the correct repository, and that your remote origin is set properly (see "Pull on the Command Line" on page 81).
2. Use `git status` to see what changes are not yet staged.

3. Stage any changes you plan to commit. In many cases, you can stage all the changes at once like this:

```
git add .
```

4. Commit the changes, adding a descriptive message:

```
git commit -m "Type your descriptive message here."
```

TIP If you are changing files but not adding or deleting any files, you can often stage and commit all in one line with `commit -am` like so:

```
git commit -am "Commit message"
```

You can also use filenames and wildcards with `git add` to stage changes in specific files or groups of files. For example, `git add directory-name/` adds everything in the `directory-name` directory, or `git add *.txt` adds all the text files in the current directory.

TIP To add more information about your changes, omit the `-m` option and write a longer commit message. See “Write Good Commit Messages” on page 43.

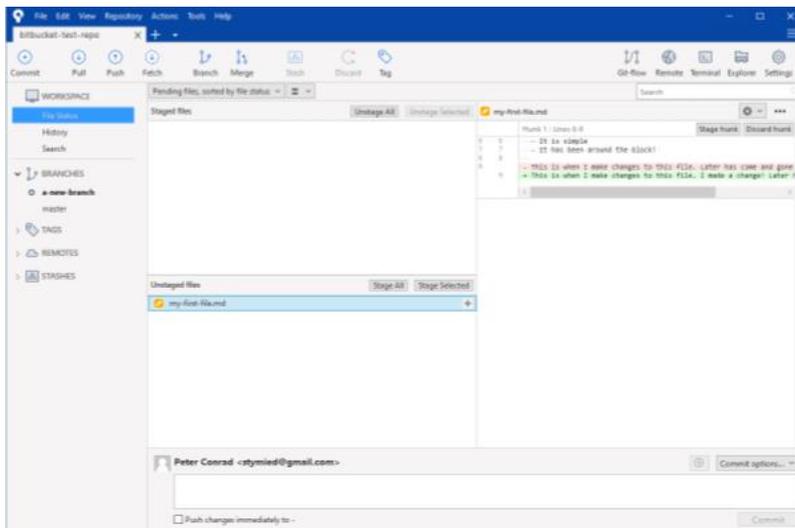
View Your Changes

You can view your unstaged changes by comparing different versions of files. The changes are called a *diff*, because they represent the difference between the current state of your files and the way they were in the last commit.

It's useful to see diffs as you work, to make sure you're making the changes you intend to.

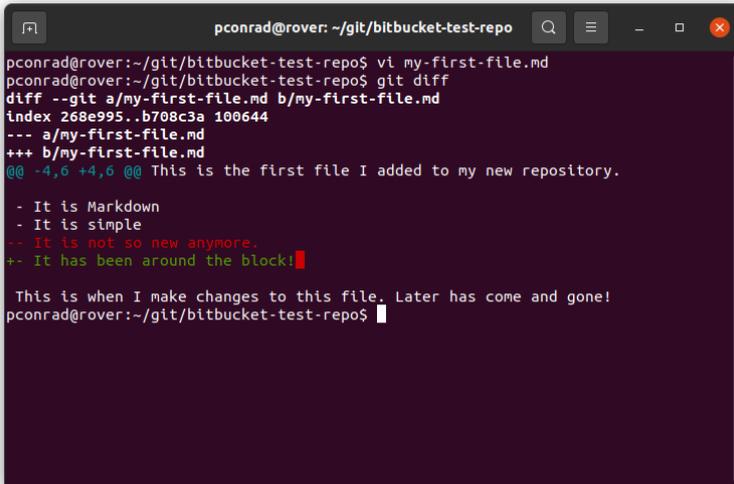
View Your Changes in GitHub Desktop or Sourcetree

- Click the filename to see the diff.



View Your Changes on the Command Line

- To see all the changes since the last commit: `git diff`
- To see the changes in a specific file: `git diff {filename}`



```
pconrad@rover: ~/git/bitbucket-test-repo
pconrad@rover:~/git/bitbucket-test-repo$ vi my-first-file.md
pconrad@rover:~/git/bitbucket-test-repo$ git diff
diff --git a/my-first-file.md b/my-first-file.md
index 268e995..b708c3a 100644
--- a/my-first-file.md
+++ b/my-first-file.md
@@ -4,6 +4,6 @@ This is the first file I added to my new repository.
- It is Markdown
- It is simple
-- It is not so new anymore.
+- It has been around the block!

This is when I make changes to this file. Later has come and gone!
pconrad@rover:~/git/bitbucket-test-repo$
```

You can use filenames, directory names, and wildcards to specify groups of files.

Push

If you want other people to be able to work on your files, put them in an online repository such as Bitbucket, GitHub, or GitLab. This is called a *push* to a *remote repository*. A remote repo is sometimes just called a *remote*.

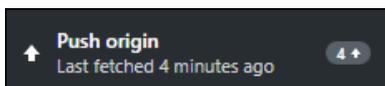
Push in Sourcetree

1. Make sure you're on the right branch in the correct repository.
2. Make sure you've committed all the changes you want to push.
3. Select **Repository > Push** or click the **Push** button.



Push in GitHub Desktop

1. Make sure you're on the right branch in the correct repository.
2. Make sure you've committed all the changes you want to push.
3. Select **Repository > Push** or click the **Push origin** button.



Push on the Command Line

1. Make sure you're on the right branch in the correct repository, and your remote origin is set properly (see “Pull on the Command Line” on page 81).
2. Make sure you've committed all the changes you want to push.
3. Type the `git push` command, specifying the remote (usually `origin`) and the branch. Example:

```
git push origin my-working-branch
```

If Git already knows what branch you're on and where your remote is, you can sometimes just type `git push` by itself.

Create a Branch

Git lets people work in separate work streams called *branches* so that they don't interfere with each other's work. A branch is just a series of commits (and a commit is a group of changes). You're always working in a branch, even if there's only one branch. When you have several branches to work in, Git remembers the state of everything in each branch so that when you switch between them everything is just how you expect it.

Creating a new branch is called *branching*, of course. The Git command for creating (or switching to) a branch is called, confusingly, *checkout*.

NOTE You can't switch branches with uncommitted changes. You always have to commit (or stash) before you switch to a new branch.

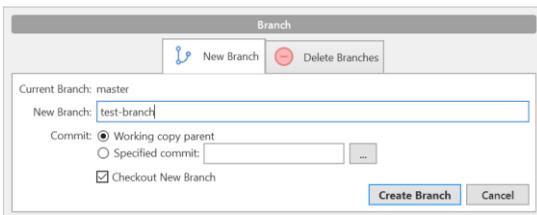
When you create a branch and push it to the remote repo, it becomes available to others. You can work on a remote branch someone else created by fetching (or pulling) and then checking out their branch.

Create a Branch in Sourcetree

1. Pull from `main` to make sure you have the latest changes.
2. Click the **Branch** button:



3. Type a descriptive name and click **Create Branch**.

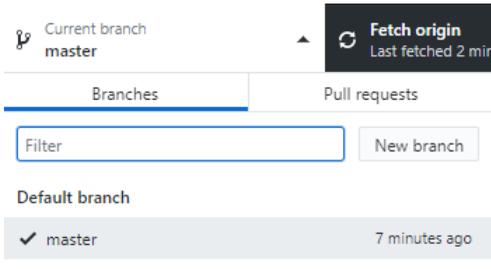


4. Look under **Branches** to see that you're on the new branch.

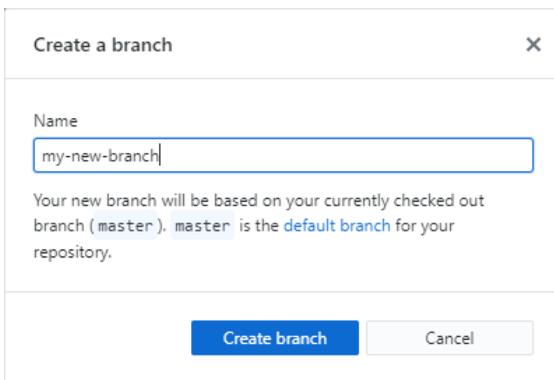
You can switch to a different branch by clicking it in the list of branches. To gain access to a remote branch, fetch or pull from the remote repository.

Create a Branch in GitHub Desktop

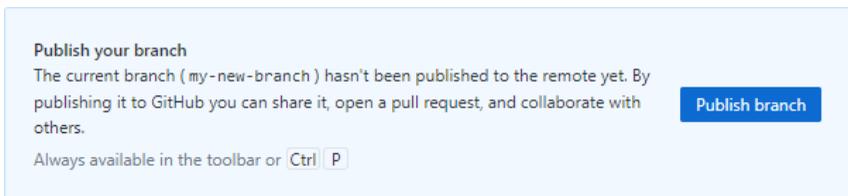
1. Pull from `main` to make sure you have the latest changes.
2. Click the **Current branch** tab, then click **New branch**:



3. Type a descriptive name and click **Create branch**:



4. Click **Publish branch**:



5. Look under **Branches** to see that you're on the new branch.

You can switch to a different branch by clicking it in the list of branches. To gain access to a remote branch, fetch or pull from the remote repository.

Create a Branch on the Command Line

1. Pull from main to make sure you have the latest changes:

```
$ git checkout main
```

```
Already on 'main'
```

```
Your branch is up to date with 'origin/main'.
```

```
$ git pull
```

```
Already up to date.
```

2. Create a new branch and switch to it with `git checkout -b`.
Example:

```
$ git checkout -b test-branch
```

```
Switched to a new branch
```

```
'test-branch'
```

You can switch to any existing branch by typing `git checkout {branch-name}`.

Example

```
$ git checkout another-branch
```

```
Switched to branch 'another-branch'
```

You can check out a remote branch using `git fetch` or `git pull` and then check it out like any other local branch:

```
$ git fetch
```

```
From https://github.com/pconrad-fb/the-lightweight-  
markup-book
```

```
* [new branch]      test-branch -> origin/test-branch
```

```
$ git checkout test-branch
```

```
Switched to a new branch 'test-branch'
```

```
Branch 'test-branch' set up to track remote branch  
'test-branch' from 'origin'.
```

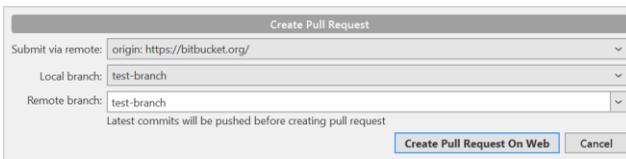
Create a Pull Request

A pull request lets others review your changes before they're merged into `main` or another important branch. A typical workflow looks like this:

1. Pull from the remote repository.
2. Create a local branch to work in.
3. Commit your changes.
4. Push your branch to the remote repository.
5. Create a pull request.
6. Once your work is approved, merge your branch into `main` (or, in some cases, another branch).

Create a Pull Request in Bitbucket and Sourcetree

1. Push to your branch.
2. Click Repository > Create pull request.
3. In the dialog that appears, click **Create Pull Request on Web**:



4. Type a description, add reviewers, and click **Create pull request**:

Create a pull request

vidriocafe/notes
Created 2019-11-29, updated 11 seconds ago

test-branch

vidriocafe/notes

master

Title ^{*} Added a new file

Description

Reviewers Start typing to search for a user

Close branch Close **test-branch** after the pull request is merged

Create pull request

Create a Pull Request in GitHub and GitHub Desktop

1. Push to your branch.
2. After you push, the banner with the Push button changes to read "Create a pull request from your current branch." Click **Create Pull Request**:

Create a Pull Request from your current branch

The current branch (my-new-branch) is already published to GitHub. Create a pull request to propose and collaborate on your changes.

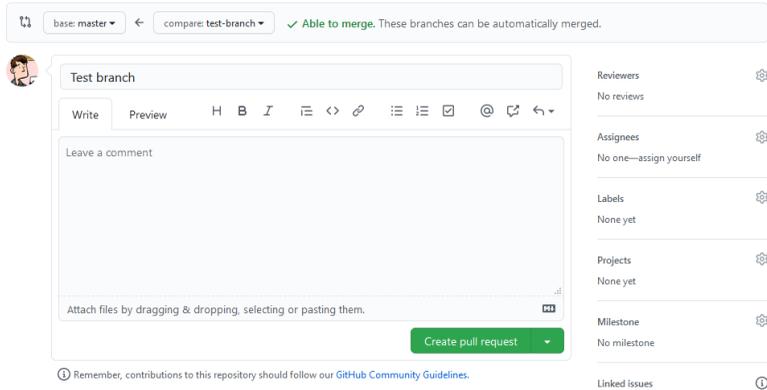
Branch menu or **Ctrl R**

Create Pull Request

3. The browser opens a page with a form for creating a pull request:

Open a pull request

Create a new pull request by comparing changes across two branches. If you need to, you can also [compare across forks](#).



base: master ← compare: test-branch ✓ Able to merge. These branches can be automatically merged.

Test branch

Write Preview H B I ☰ <> 📎 📧 📧 ↶

Leave a comment

Attach files by dragging & dropping, selecting or pasting them.

Create pull request

Remember, contributions to this repository should follow our [GitHub Community Guidelines](#).

Reviewers
No reviews

Assignees
No one—assign yourself

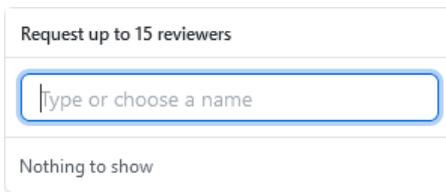
Labels
None yet

Projects
None yet

Milestone
No milestone

Linked issues

4. Click the gear next to **Reviewers** to add reviewers:



Request up to 15 reviewers

Type or choose a name

Nothing to show

5. Click Create pull request.

Create a Pull Request on the Command Line

1. Push to your branch. Example:

```
$ git push origin my-working-branch
```

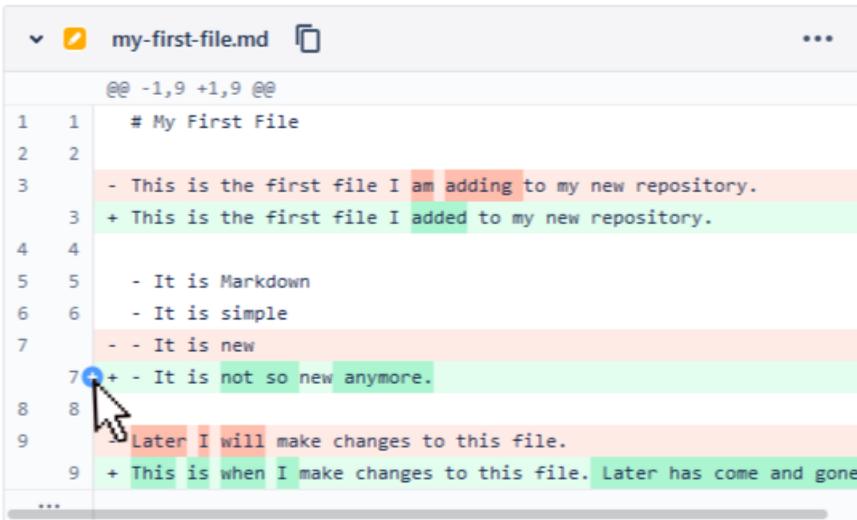
```
Enumerating objects: 14, done.  
Counting objects: 100% (14/14), done.  
Delta compression using up to 4 threads  
Compressing objects: 100% (10/10), done.  
Writing objects: 100% (10/10), 4.39 KiB | 1.10 MiB/s,  
done.  
Total 10 (delta 2), reused 0 (delta 0)  
remote: Resolving deltas: 100% (2/2), completed with 1  
local object. remote:  
remote: Create a pull request for 'my-working-branch'  
on GitHub by visiting: remote:  
https://github.com/pconrad-fb/markdown/pull/new/my-  
working-branch  
remote:  
To https://github.com/pconrad-fb/markdown.git  
* [new branch] my-working-branch -> my-working-branch
```

2. Take note of the URL in the next line after the Create a pull request line in the output. Copy and paste this URL into a browser.
3. Follow the instructions on the screen.

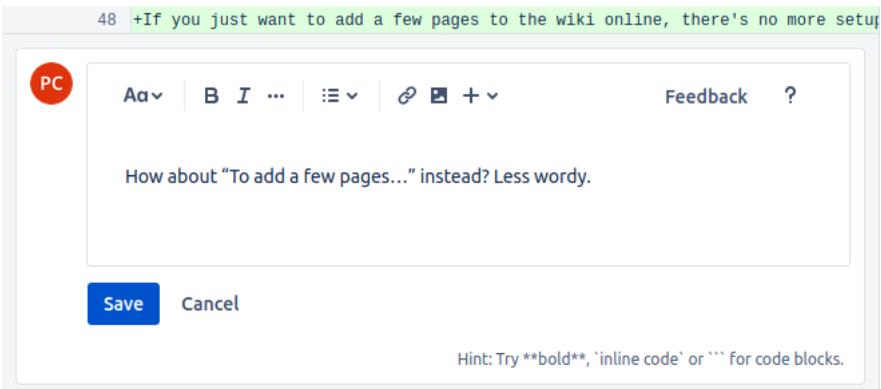
Approve and Merge

If there's more than one branch, there always comes a time to *merge*, which means to add the changes from one branch into another. Approving and merging a pull request happens in the web interface provided by the Git host.

Someone reviewing the pull request can add a comment by clicking the small plus sign next to a line in a file.



This opens a dialog for writing a comment on that line.



When reviewers are satisfied, each one can approve the pull request by clicking the **Approve** button. Once enough reviewers have approved, you can merge.

Git does its best to merge changes automatically. If there are edits to the same part of the same file on two or more branches, Git will ask you which edits take priority. This is called a merge conflict, and is usually not as bad as it sounds. See “Merge Conflict” on page 105 to learn how to fix merge conflicts.

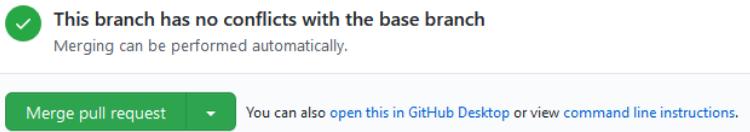
Merge a Pull Request in Bitbucket

- Click **Merge**:



Merge a Pull Request in GitHub

- Click **Merge**:



Trouble

From time to time, even the most seasoned Git users make a mistake. Here are some of the most common mistakes and how to fix them.

Edited in the Wrong Branch

You've edited a file in the wrong branch. What you'd like to be able to do is undo those changes, switch branches, then re-do them. Actually, it would be even better to lift those changes off of the wrong branch, laying them gently on top of the branch you meant to be in.

Fortunately, Git provides a command called `stash` that does exactly that.

If you've not only edited, but committed by mistake, un-commit the changes first (see "Committed by Mistake" on page 107) before proceeding with the following steps. If you've just committed in the wrong branch, see "Committed in the Wrong Branch" on page 108 instead.

1. Make sure you're in the right directory.
2. Stash your uncommitted changes:
`git stash`
3. Switch to the branch you wish you had been working in:
`git checkout the-right-branch`
Switched to branch 'the-right-branch'
4. Use `stash pop` to apply the changes there:
`git stash pop`

Edited the Wrong File

You opened a file to look at it, but then your cat walked across the keyboard.

You're not sure what was added or deleted. You just want to go back to the way things were at the last commit. For this, use `checkout`—it's not just for switching branches!

1. Make sure you're in the right directory.
2. Use `git status` to see what files were accidentally modified. For example:

```
$ git status
```

```
On branch main
```

```
Changes not staged for commit:
```

```
  (use "git add <file>..." to update what will be  
  committed)
```

```
  (use "git checkout -- <file>..." to discard  
  changes in working directory)
```

```
        modified: dont-change-this.md
```

3. Use `git checkout -- <file>` to undo the changes. For example:

```
git checkout -- dont-change-this.md
```

TIP The output of the `git status` command tells you how to use `git checkout` this way.

Merge Conflict

When two changes happen in the same place in the same file, Git can't merge without your help. You need to edit the file and decide which of the two changes to keep.

Most of the time, a merge conflict happens when you pull. Git tries to fetch the changes and merge them into the current branch, but can't resolve the overlapping changes. Git will tell you which file (or files) have a conflict.

When you open the file, the merge conflict looks like this:

```
<<<<<<< HEAD
Some content that was changed by one person
=====
Other content that someone else changed
>>>>>>> 9af9d3b
```

HEAD is usually a pointer to the latest commit in the branch you're on. The other label can be another branch name or a *hash*--a number representing another commit.

All you need to do is decide which version of the content you want to keep and then delete the merge conflict markers (<<<<<<<, =====, >>>>>>>).

After you've resolved all the changes in that way, just stage and commit again.

TIP If you have a merge conflict when trying to merge a pull request, it's easiest to pull and resolve the merge conflict locally, then push and try again.

Detached HEAD

HEAD is a pointer to the currently checked out commit in the current branch—like a YOU ARE HERE arrow. Usually, this is the latest commit in the branch. If you check out an older commit, you enter a detached HEAD state, meaning that the HEAD doesn't point to the latest commit, as Git normally expects. With a detached HEAD, anything you commit can only be reached later by knowing the commit's hash. This is tremendously inconvenient and doesn't give Git the opportunity to keep track of your work, since you aren't really on a branch.

- If you're just looking around at an older commit and don't plan to make changes there, don't worry about the detached HEAD. When you're ready to go back to working, use `git checkout HEAD` to go back to the latest commit in that branch, or `git checkout {branch}` to go to the latest commit in a different branch.
- If you've gone back to the older commit to do more work or fix something, use `git checkout -b {new-branch-name}` to start a new branch there.

WARNING Don't commit with a detached HEAD.

Staged by Mistake

You edited the right file the right way, but then you added it to the staging area too hastily. You don't want to undo your changes to the file, but you would like to remove it from the next commit. This is one of the uses of `reset`. You can also use `reset` to do more drastic rollbacks—you can undo entire commits if needed.

1. Make sure you're in the right directory.
2. Use `git status` to see what files were accidentally modified. For example:

```
$ git status
```

```
On branch main
Changes to be committed:
```

(use "git reset HEAD <file>..." to unstage)

```
renamed: README.md -> README
modified: dont-commit-this.md
```

3. Use `git reset` to remove the file from the next commit. For example:

```
git reset HEAD dont-commit-this.md
```

TIP The output of the `git status` command tells you how to use `reset` to unstage changes.

Committed by Mistake

If you've committed too hastily, you can un-commit your changes so long as you haven't also pushed them.

1. Make sure you're in the right directory.
2. Use `git status` to check what branch you're on. If necessary, change to the branch where you erroneously committed. For example:

```
$ git checkout the-wrong-branch
```

```
Switched to branch 'the-wrong-branch'
```

3. Use `git reset` to un-commit the changes:

```
git reset HEAD~ --soft
```

You can re-commit the changes later if you like.

If you want to move the changes to another branch first, you can use `git stash` (see "Edited in the Wrong Branch" on page 103).

Committed in the Wrong Branch

If you have committed—perhaps several times—on the wrong branch, you can rebase your changes to the correct branch.

1. Check out the first commit in the series of commits you want to move:

```
git checkout commit hash
```

2. Use rebase to move it to the tip of another branch:

```
git rebase new-branch
```

3. Use `git log` to verify that you did what you expected:

```
git log --oneline
```

Lost Some Changes to History

If some changes got lost, reverted, or overwritten by later commits, you can go back and find them again.

Retrieve All Changes from an Old Commit

If you want to retrieve changes so you can work on them again, go back to the old commit (for example, `d555f66`) and grab them:

1. Use `git checkout` to check out the old commit. Example:

```
git checkout d555f66
```

2. Do a soft reset, which uncommits all the changes:

```
git reset --soft HEAD~1
```

3. If you like, you can use `git status` to see what uncommitted changes you now have.

4. Use `git stash` to stash the changes.

5. Use `git checkout` to check out the branch again. If you have been working in the main branch, use `git checkout main`:

```
$ git checkout main
```

```
Previous HEAD position was d555f66 Committing some lovely changes  
Switched to branch 'main'  
Your branch is up to date with 'origin/main'.
```

6. Use `git stash pop` to apply the changes.

You can continue working on the changes, then commit and push when you're ready.

Grab an Old Version of a File

You can check out a version of a file from a specific commit:

- `git checkout commit file`

This restores your working copy of the file to the way it was in that commit. You can then stage and commit the changes.

Example

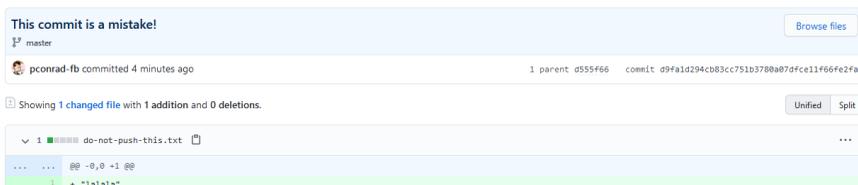
```
git checkout d555f66 my-file.txt
```

Pushed by Mistake

If you've pushed a commit erroneously, your mistake is now available to other people. Fortunately, there is a way to "undo" a push.

TIP Let other people know about the bad push so that they don't pull until you fix the repo.

It can be useful to log on to the git host to see what actually got pushed.



Revert a Bad Push

Here are the steps to revert a bad push:

1. Use `git log` to show a list of recent commits, and identify the one you pushed by mistake. The `--oneline` option makes the list easier to read. The list is sorted from newest to oldest. Example:

```
$ git log --oneline
```

```
d9fa1d2 (HEAD -> master, origin/master, origin/HEAD) This commit is a mistake!  
d555f66 Committing some lovely changes
```

2. Use `git revert` and the commit number (or commit *hash*) to create a new commit that un-does everything in the bad commit. Example:

```
git revert d9fa1d2
```

3. A screen appears where you can type a commit message, or just keep the default message. Saving and exiting are different depending on which editor Git is set up to use.
 - If you see a colon at the bottom of the screen, you're probably using `vi` or `Vim`. Type `wq` and press enter.
 - If you see a list of commands at the bottom of the screen, you're probably using `Nano`. Press `Control-o` then `Control-x`.

4. You can use `git status` to see that you have a new commit, ready to push:

```
$ git status
```

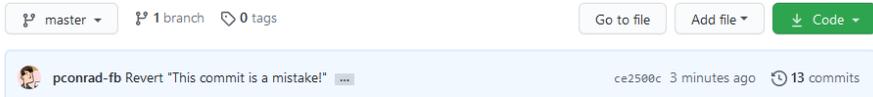
```
On branch main
```

```
Your branch is ahead of 'origin/main' by 1 commit.  
(use "git push" to publish your local commits)
```

5. Push the commit with the `git push` command:

```
git push
```

That takes care of fixing the repo. Anyone who pulls will now get the entire history, including both the mistake and the fix, and all will be right with the world.



This operation leaves things as they were before you made the changes that you mistakenly pushed. In other words, your files will look like they did after you committed "Committing some lovely changes," and all the work you did in "This commit is a mistake" has been undone.

More trouble

For more help and advice, check out Dangit, Git!?! (<https://dangitgit.com/>)

Glossary

admonition

A note, warning, or other call-out that draws attention to a block of content.

add

In Git, to *stage* changes for a *commit*.

AsciiDoc

A markup language invented by O'Reilly for articles, books, and other documentation.

authenticate

To verify identity for the purpose of access to a protected system or capability.

Bash

A Unix/Linux shell and language that lets users execute commands and programs.

branch

A series of commits representing changes to one or more files in a Git repository.

branching

Creating and using *branches*.

change

In Git, a modification, creation, or deletion of a file.

checkout

In Git, to switch to a different branch or restore a file. + In centralized source control, to lock a file, preventing others from editing it.

chmod

A shell command that changes file permissions.

client

Hardware or software that accesses a *service*. A web browser is a client to a webserver, and a Git client accesses a service provided by a Git host.

clone

In Git, to make a complete local copy of a *remote repository* so you can work with the files on your computer.

cloud

Someone else's computer.

commit

In Git, to save your changes to the *local repository*. + A group of changes saved together using the `commit` command.

commit hash

A unique number that identifies a commit.

commit message

A summary of the purpose of a commit, consisting of a subject line and an optional body, that helps others understand the purpose and scope of the changes.

credentials

Proof of identity, usually a username and a password or token, used for *authentication*.

CSS

Cascading Style Sheets, a style sheet language for defining the look and feel of a document written in HTML or another markup language.

DITA

Darwin Information Typing Architecture, an XML-based document markup language.

diff

A representation of the lines that have changed in a file.

div

A division or section in an HTML document, specified with a `<div>` tag.

docs-as-code

A content or documentation process that uses source control to manage documentation as if it were source code.

dynamic site

A site that is generated or modified at the time it is displayed. See *static site*.

Extensible Markup Language

See *XML*.

fenced

Delineated with a series of characters. For example, a *fenced code block* is marked with three backticks (```) at the top and bottom.

fetch

In Git, to download changes from the *remote repository*.

frontmatter

Metadata at the start of a file, often including information such as the title, author, and date.

FTP

File Transfer Protocol, a way of exchanging files between your computer and a server.

Git

A distributed source control system.

Git Centralized Workflow

A very simplified Git branching strategy characterized by no branching at all.

Git wiki

An additional repository, attached to a Git repository, for the purpose of displaying and managing content (often, content about the Git repository).

Git wiki structure

A content structure in which the display paths or URLs to content pages are defined by the directory paths of the files that make up the content.

GitHub Flow

A Git branching strategy, characterized by short-lived working branches that merge directly to the main branch.

hash

A unique, fixed-length string, output by an algorithm, used to index a piece of data. See *commit hash*.

HEAD

In Git, a pointer to the current commit.

host

A server, often on the web. A *Git host* provides access to Git repositories, a *web host* provides access to websites, and so on.

HTML

HyperText Markup Language, the standard markup language for creating web pages.

JavaScript

A programming language that enables the creation of interactive features on web pages.

JSON

JavaScript Object Notation, a format for storing and transporting data.

LaTeX

A document preparation system for high-quality typesetting.

Linux

A family of Unix-like operating systems first designed by Linus Torvalds in 1991.

local

On your own computer.

Markdown

A simple *markup language* originally designed as an easy way to write HTML pages.

markup language

A way of indicating display formatting and other information within a document.

merge

In Git, to combine two sets of changes into one branch.

merge conflict

In Git, a merge that cannot be completed automatically because the same parts of the files have been modified in both sets of changes.

metadata

Information about the content in a file, or about the file itself.

origin

In Git, a short name for the remote repository you push and pull from by default.

package manager

A tool for installing software.

PDF

Portable Document Format, a file format developed by Adobe in 1993 to present documents consistently across software, hardware, and operating systems.

permissions

Settings that specify what actions can be taken and by whom. For example, file permissions can specify who can read, write, or execute the file.

pull

In Git, to *fetch* and *merge* changes from a *remote* to your *local repository*.

pull request

In Git, a set of proposed changes to be approved and then merged into a branch.

push

In Git, to upload changes from your local *repository* to a *remote*.

Python

A popular programming language.

Python Markdown extensions

A set of additional features and syntax provided with the Python implementation of Markdown.

rebase

In Git, to move commits from one branch to another.

recursion

See recursion.

remote

A remote repository.

remote repository

A version of your project that is hosted on the network or online rather than on your computer. Also known as the *remote repo* or simply the *remote*.

repo

Repository.

repository

In Git, a collection of files and the entire history of all changes made to them.

reStructuredText

A markup language used primarily for documenting Python programs.

Samba

Open source software that runs on Unix or Linux to enable communication with Windows clients over a network.

script

A computer program that automates the execution of commands or tasks.

server

A computer or application that provides a service for other programs or devices, which in turn are called *clients*.

Sharepoint

A web-based collaboration platform that integrates with Microsoft Office and is often used to manage and store documents.

shell

A program that lets users type commands for the operating system to execute.

source control

A way of tracking and managing changes to code or other content.

stage

In Git, to specify which changes to save in the next *commit*.

staging area

In Git, the tree that stores all the changes that are to be included in the next *commit*.

stash

In Git, to record the current state of the working directory and revert the working directory to the previous *commit*.

static site

A site composed of HTML pages or other documents that are made available exactly as stored, as opposed to a *dynamic site* whose pages are rendered on the fly when they are requested. A static site often performs better and can be more secure, but lacks some of the capabilities of a *dynamic site*.

static site generator

A tool that builds a *static site*.

tag

A label.

token

A long string of random characters use in *authentication*, often in place of a password.

TOML

Tom's Obvious, Minimal Language, a text format for configuration files or metadata.

Unix

A family of operating systems designed at Bell Labs in the 1970s, that Linux is like.

unstage

In Git, to remove previously *staged* changes from the upcoming *commit*.

version control

See source control.

WebDAV

Web Distributed Authoring and Versioning, an HTTP extension that lets clients perform remote operations on content.

wiki

A structured content site, often edited and managed by the readers themselves, that collects information about a particular topic.

wildcard

A character that stands in for one or more characters to allow multiple possible matches to a single string.

working branch

In Git, a temporary branch created for working on a particular set of content or code changes.

working directory

The folder on your local computer where you store the content you are editing.

WYSIWYG

What You See Is What You Get, an editing experience that mimics the appearance of the document in its final form.

XML

Extensible Markup Language, a set of rules for encoding documents with *tags* that are both human-readable and machine-readable.

YAML

Yaml Ain't Markup Language, a text format for configuration files or metadata.

Thanks

I couldn't have put this book together without help from a few people:

Dennis Gentry, Alan Grover, Ben Hamilton, Meliana Handoko, Steve Lacy, and anyone I've ever bothered about Git.