Beej’s Guide to Git

Brian “Beej Jorgensen” Hall

v0.0.7, Copyright © July 13, 2024

Contents

Chapter 1

Foreword

Hello again, everyone! In my role as as industry professional-turned-college instructor, I definitely see my fair share of students struggling with Git.

And who can blame 'em? It's a seemingly-overcomplicated system with lots of pitfalls and merge conflicts and detached heads and remotes and cherrypicks and rebases and an endless array of other commands that do who-knows-what.

Which leads us directly to the goal: let's make sense of all this. We'll start off easy (allegedly) with commands mixed in with some theory of operation. And we'll see that understanding what Git does under the hood is critical to using it correctly.

And I *promise* there's definitely a chance that after you get through some of this guide, you might actually start to appreciate Git and like using it.

I've been using it for years (I'm using it for the source code for this guide right now) and I can certainly vouch for it becoming easier over time, and then, even, second nature.

But first, some boilerplate!

1.1 Audience

The initial draft of this guide was put online for the university students where I worked (or maybe still work, depending on when you're reading this) as an instructor. So it's pretty natural to assume that's the audience I had in mind.

But I'm also hoping that there are enough other folks out there who might get something of use from the guide, as well, and I've written it in a more general sense with all you non-college students in mind.

This guide assumes that you have basic POSIX shell (i.e. Bash, Zsh, etc.) usage skills, i.e.:

- You know basic commands like `cd`, `ls`, `mkdir`, `cp`, etc.
- You can install more software.

It also assumes you're in a Unix-like environment, e.g. Linux, BSD, Unix, macOS, WSL, etc. with a POSIX shell. The farther you are away from that (e.g. PowerShell, Commodore 64), the more manual translation you'll have to do.

Windows is naturally the sticking point, there. Luckily Git for Windows comes with a Bash shell variant called Git Bash. You can also install WSL[1] to get a Linux environment running on your Windows box. I wholeheartedly recommend this for hacker types, since Unix-like systems are hacker-awesome, and additionally I recommend you all become hacker types.

[1] https://learn.microsoft.com/en-us/windows/wsl/

1.2 Official Homepage

This official location of this document is (currently) https://beej.us/guide/bggit/[2].

1.3 Email Policy

I'm generally available to help out with email questions so feel free to write in, but I can't guarantee a response. I lead a pretty busy life and there are times when I just can't answer a question you have. When that's the case, I usually just delete the message. It's nothing personal; I just won't ever have the time to give the detailed answer you require.

As a rule, the more complex the question, the less likely I am to respond. If you can narrow down your question before mailing it and be sure to include any pertinent information (like platform, compiler, error messages you're getting, and anything else you think might help me troubleshoot), you're much more likely to get a response.

If you don't get a response, hack on it some more, try to find the answer, and if it's still elusive, then write me again with the information you've found and hopefully it will be enough for me to help out.

Now that I've badgered you about how to write and not write me, I'd just like to let you know that I *fully* appreciate all the praise the guide has received over the years. It's a real morale boost, and it gladdens me to hear that it is being used for good! `:-)` Thank you!

1.4 Mirroring

You are more than welcome to mirror this site, whether publicly or privately. If you publicly mirror the site and want me to link to it from the main page, drop me a line at `beej@beej.us`.

1.5 Note for Translators

If you want to translate the guide into another language, write me at `beej@beej.us` and I'll link to your translation from the main page. Feel free to add your name and contact info to the translation.

Please note the license restrictions in the Copyright and Distribution section, below.

1.6 Copyright and Distribution

[2] https://beej.us/guide/bggit/

1.7 Dedication

The hardest things about writing these guides are:

- Learning the material in enough detail to be able to explain it
- Figuring out the best way to explain it clearly, a seemingly-endless iterative process
- Putting myself out there as a so-called *authority*, when really I'm just a regular human trying to make sense of it all, just like everyone else
- Keeping at it when so many other things draw my attention

A lot of people have helped me through this process, and I want to acknowledge those who have made this book possible.

- Everyone on the Internet who decided to help share their knowledge in one form or another. The free sharing of instructive information is what makes the Internet the great place that it is.
- Everyone who submitted corrections and pull-requests on everything from misleading instructions to typos.

Thank you! ♥

Chapter 2

Git Basics

Welcome to the *Beej's Guide to Git*!

This guide has two goals, in no particular order:

1. Help you get some familiarity with Git syntax on the command line.
2. Help you get a mental model that describes how Git stores its information.

I feel the second of these is very important for becoming even remotely adept at using Git, which is why I spend so much time talking about it. Yes, you can get by with a cheat-sheet of common Git commands, but if you want to fearlessly use the tool to its full effectiveness, you gotta learn the internals!

2.1 What is Git?

Git is a *source code control system*, also known as a *version control system*.

Clear? OK, not really? Let's dive in a bit more, then!

Git's main job is to keep a log of *snapshots* of the current state of all your source code in a particular directory tree.

The idea is that you'll make some changes (to implement a feature, for example), and then you'll *commit* those changes to the *source code repo* (repository) once the feature is ready. This saves the changes to the repo and allows other collaborators to see them.

And if you ever change something you didn't want to, or your want to see how things were implemented in the past, you can always *check out* a previous commit and take a look.

Git keeps a history of all the commits you've ever made. Assuming nothing criminal is happening, this should be a great relief to you; if you accidentally delete a bunch of code, you can look at a previous commit and get it all back.

But that's not all! As we'll see, Git also works well as a remote backup mechanism, and works wonderfully when cooperating with a team on the same codebase.

Definitions:

- **Source Code Control System**/**Version Control System**: Software that manages changes to a software project potentially consisting of thousands of source files edited by potentially hundreds of developers. Git is a source code control system. There are many others.
- **Commit**: An explicit moment in time where a snapshot of the contents of all the source files are recorded in the source code control system. Very, very typically the code is in a working state when the commit is made; in other words, the commit represents in some ways a seal of approval that the repo in this state is in working order even if the changes aren't complete.

  Example commits might be:

  - "Added feature *X* to the codebase."

  - "Fixed bug *Y*."
  - "Merged other contributor's changes into the codebase."
  - "Partially completed the Spanish translation."

- **Repo**/**Source Code Repository**: This is where a particular software project is stored in the source code control system. Typically each project has its own repo. For example, you might "create a Git repo" to hold a new project you're working on.

  Sometimes repos are local to your computer, and sometimes they're stored on other, remote computers.

- **Check out**: To look at a particular commit (or branch—more later).

2.2 What is GitHub?

GitHub[1] is **not** Git.

2.3 What is GitHub?

Oh, more?

GitHub is a website that provides a front end to a lot of Git features, and some additional GitHub-specific features, as well.

It also provides remote storage for your repo, which acts as a backup.

Takeaway: GitHub is a web-based front-end to Git (specifically one that works on the copy of your repo at GitHub—stay tuned for more on that later).

> **Information:** GitLab[a] is a competitor to GitHub. Gitea[b] is an open-source competitor that allows you to basically run a GitHub-like front-end on your own server. None of this information is immediately important.
>
> [a] https://gitlab.com
> [b] https://docs.gitea.com/

Regardless of whatever repos you have on GitHub, you'll also have copies (known as *clones*) of those repos on your local system for you to work on. Periodically, in a common workflow, you'll sync your copy of the repo with GitHub.

> **Note:** Even though you might be commonly using GitHub, there's no law that says you have to. You can just create and destroy repos on your local system all you want, even if you're not connected to the Internet. See Appendix: Making a Playground for more information once you're more comfortable with the basics.

2.4 The Most Basic Git Workflow

There's a super-common workflow that you'll use repeatedly:

1. *Clone* a *remote* repo. The remote repo is commonly on GitHub, but not necessarily.
2. Make some local changes.
3. Add those changes to the *stage*.
4. *Commit* those changes.
5. *Push* your changes back to the remote repo.
6. Go back to Step 2.

[1] https://github.com/

This is not the only workflow; there are others that are also not uncommon.

Definitions:

- **Clone** (verb): to make a copy of a remote repo locally.
- **Clone** (noun): a local copy of a remote repo.
- **Remote**: In Git, a clone of a repo in another location.
- **Stage**: In Git, a place you add copies of files to in preparation for a commit. The commit will include all the files that you've placed on the stage. It will not include files you haven't placed on the stage, even if you've modified those files.

2.5 What is Cloning?

First, some backstory.

Git is what's known as a *distributed* version control system. This means that, unlike many version control systems, there's no one central authority for the data. (Though commonly Git users treat a site like GitHub in this regard, loosely.)

Instead, Git has *clones* of repos. These are complete, standalone copies of the entire commit history of that repo. Any clone can be recreated from any other. None of them are more powerful than any others.

Looking back at The Most Basic Git Workflow, above, we see that Step 1 is to clone an existing repo.

If you're doing this from GitHub, it means you're making a local copy of an entire, existing GitHub repo.

Making a clone is a one time-process, typically (though you can make as many as you want).

Definitions:

- **Distributed Version Control System**: A VCS in which there is no central authority of the data, and multiple clones of a repo exist.

  This means after you clone a repo, there are two: one that is remote, and one that is local to your computer.

  These clones are completely separate and changes you make to your local repo will not be reflected in the remote clone. Unless, that is, you explicitly make them interact.

2.6 How Do Clones Interact?

After you make a clone, there are two major operations you typically use:

- **Push**: This takes your local commits and uploads them to the remote repo.
- **Pull**: This takes the remote commits and downloads them to your local repo.

Behind the scenes, there's a process going on called a *merge*, but we'll talk more about that later.

Until you push, your local changes aren't visible on the remote repo.

Until you pull, the changes on the remote repo aren't visible on your local repo.

2.7 Actual Git Usage

Let's put all this into play. This section assumes you have the command line Git tools installed. It also generally assumes you're running a Unix shell like Bash or Zsh.

> Linux/BSD/Unix and Mac users will already have these shells. Recommendation for Windows users is to install and run Ubuntu with WSL[a] to get a virtual Linux installation.
>
> [a] https://learn.microsoft.com/en-us/windows/wsl/

For this example, we'll assume we have a GitHub repo already in existence that we're going to clone.

Recall the process in The Most Basic Git Workflow, above:

1. *Clone* a *remote* repo. The remote repo is commonly on GitHub, but not necessarily.
2. Make some local changes.
3. Add those changes to the *stage*.
4. *Commit* those changes.
5. *Push* your changes back to the remote repo.
6. Go back to Step 2.

2.7.1 Step 0: One-time Setup

"Wait! You didn't say there was a Step 0!"

Yes, one time, before you start using Git, you should tell it what your name and email address are. These will be attached to the commits you make to the repo.

You can change them any time in the future, and you can even set them on a per-repo basis. But for now, let's set them globally so Git doesn't complain when you make a commit.

You just have to do this once then never again (unless you want to).

Type both of these on the command line, filling in the appropriate information.

> **NOTE**: in this guide, things you type at the shell prompt are indicated by a prefaced `$`. Don't type the `$`; just type what follows it. Your actual shell prompt might be `%` or `$` or something else, but here we use the `$` to indicate it.

```
$ git config --global user.name "Your Name"
$ git config --global user.email "your-email@example.com"
```

If you need to change them in the future, just run those commands again.

2.7.2 Step 1: Clone an Existing Repo

Let's clone a repo! Here's an example one you can actually use. Don't worry–you can't mess anything up on the remote repo even though (and because) you don't own it.

> Like we said before, this isn't the only workflow. Sometimes people make a local repo first, add some commits, then create a remote repo and push those commits. But for this example, we'll assume the remote repo exists first, though this isn't a requirement.

Switch in a subdirectory where you want the clone created. This command will create a new subdirectory out of there that will hold all the repo files.

(In the example, anything that begins with `$` represents the shell prompt indicating this is input, not output. Don't type the `$`; just type in the part after it.)

```
$ git clone https://github.com/beejjorgensen/git-example-repo.git
```

You should see some output similar to this:

```
Cloning into 'git-example-repo'...
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 4 (delta 0), reused 4 (delta 0), pack-reused 0
```

```
Receiving objects: 100% (4/4), done.
```

Congratulations! You have a clone of the repo. Let's have a peek:

```
$ cd git-example-repo
$ ls -la
```

And we see a number of files:

```
total 16
drwxr-xr-x   5 user  user  160 Jan 26 11:50 .
drwxr-xr-x  14 user  user  448 Jan 26 11:50 ..
drwxr-xr-x  12 user  user  384 Jan 26 11:50 .git
-rw-r--r--   1 user  user   65 Jan 26 11:50 README.md
-rwxr-xr-x   1 user  user   47 Jan 26 11:50 hello.py
```

There are two files in this repo: `README.md` and `hello.py`.

> The directory `.git` has special meaning; it's the directory where Git keeps all its metadata and commits. You can look in there, but you don't have to. If you do look, don't change anything. The only thing that makes a directory a Git repo is the presence of a valid `.git` directory within it.

Let's ask Git what it things the current status of the local repo is:

```
$ git status
```

Gives us:

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

nothing to commit, working tree clean
```

There's a lot of information here, surprisingly.

We haven't talked about branching yet, but this is letting us know we're on branch `main`. That's fine for now.

It also tells us this branch is up to date with a branch called `origin/main`. A branch in Git is just a reference to a certain commit that's been made, like a PostIt note attached to that commit. (Recall that a commit is a snapshot of the code repo at some time.)

We don't want to get caught up in the intricacies of branching right now, but bear with me for a couple paragraphs.

`origin` is an alias for the remote repository that we originally cloned from, so `origin/main` is "branch `main` on the repo you originally cloned from".

There is one important thing to notice here: there are two `main` branches. There's the `main` branch on your local repo, and there's a corresponding `main` branch on the remote (`origin`) repo.

Remember how clones are separate? That is, changes you make on one clone aren't automatically visible on the other? This is an indication of that. You can make changes your your local `main` branch, and these won't affect the remotes `origin/main` branch. (At least, not until you push those changes!)

Lastly, it mentions we're up-to-date with the latest version of `origin/main` (that we know of), and that there's nothing to commit because there are no local changes. We're not sure what that means yet, but it all sounds like vaguely good news.

2.7.3 Step 2: Make Some Local Changes

Let's edit a file and make some changes to it.

> Again, don't worry about messing up the remote repo—you don't have permissions to do that. Your safety is completely assured from a Git perspective.

If you're using VS Code, you can run it in the current directory like so:

```
$ code .
```

Otherwise, open the code in your favorite editor, which, admit it, is Vim[2].

Let's change `hello.py`:

It was:

```
#!/usr/bin/env python

print("Hello, world!")
```

but let's add a line so it reads:

```
#!/usr/bin/env python

print("Hello, world!")
print("Hello, again!")
```

And save that file.

Let's ask Git what the status is now.

```
$ git status

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

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   hello.py

no changes added to commit (use "git add" and/or "git commit -a")
```

This is telling us a couple important things.

First, Git has detected that we modified a file, namely `hello.py`, which we did.

But it also says there are `no changes added to commit` (i.e. "there is nothing to make a commit with"). What does that mean?

It means we haven't added anything to the *stage* yet. Recall that the stage is where we can place items that we wish to include in the next commit. Let's try that.

2.7.4 Step 3: Add Changes to the Stage

The Git status message, above, is trying to help us out. It says:

[2] https://www.vim.org/

```
no changes added to commit (use "git add" and/or "git commit -a")
```

It's suggesting that `git add` will add things to the stage—and it will.

Now we, the developers, know that we modified `hello.py`, and that we'd like to make a commit that reflects the changes to that file. So we need to first add it to the stage so that we can make a commit.

Let's do it:

```
$ git add hello.py
$ git status

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

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   hello.py
```

Now it's changed from saying "Changes not staged for commit" to saying "Changes to be committed", so we have successfully copied `hello.py` to the stage!

> There's also a helpful message there about how to *unstage* the file. Let's say you accidentally added it to the stage and you changed your mind and wanted to not include it in the commit after all. You can run
>
> ```
> $ git restore --staged hello.py
> ```
>
> and that will change it back to the "Changes not staged for commit" state.

2.7.5 Step 4: Commit those Changes

Now that we have something copied to the stage, we can make a commit. Recall that a commit is just a snapshot of the state of the repo given the modified files on the stage. Modified files not on the stage will not be included in the snapshot. Unmodified files are automatically included in the snapshot.

In short, the commit snapshot will contain all the unmodified files Git currently tracks plus the modified files that are on the stage.

Let's do it:

```
$ git commit -m "I added another print line"

[main 0e1ad42] I added another print line
 1 file changed, 1 insertion(+)
```

> The `-m` switch allows you to specify a commit message. If you don't use `-m`, you'll be popped into an editor, which will probably be Nano or Vim, to edit the commit message. If you're not familiar with those, see Getting Out of Editors for help.

And that's good news! Let's check the status:

```
$ git status

On branch main
```

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

nothing to commit, working tree clean
```

"Nothing to commit, working tree clean" means we have no local changes to our branch.

But look! We're "ahead of 'origin/main' by 1 commit"! This means our local commit history on the `main` branch has one commit that the remote commit history on its `main` branch does not have.

Which makes sense—the remote repo is a clone and so it's independent of our local repo unless we specifically try to sync them up. It doesn't magically know that we've made changes to our local repo.

And Git is helpfully telling us to run `git push` if we want to update the remote repo so that it also has our changes.

So let's try to do that. Let's push our local changes to the remote repo.

2.7.6 Step 5: Push Your Changes to the Remote Repo

Let's push our local changes to the remote repo:

```
$ git push
```

And that produces:

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

Uh oh—trouble brewing. Let's try entering our credentials:

```
Username for 'https://github.com': my_username
Password for 'https://beejjorgensen@github.com': [my_password]
remote: Support for password authentication was removed on August
        13, 2021.
remote: Please see https://docs.github.com/en/get-started/getting-
        started-with-git/about-remote-repositories#cloning-with-
        https-urls for information on currently recommended modes
        of authentication.
fatal: Authentication failed for 'https://github.com/beejjorgensen/
       git-example-repo.git/'
```

Well, that's all kinds of not-working. Largely this is because you don't have permission to write to that repo since you're not the owner. And, notably, support for authenticating with a password seems to have been removed in 2021 which, last I checked, was in the past.

So what do we do? Firstly, we should be the owner of the GitHub repo that we've cloned and that'll solve some of the permission problems. Secondly, we'd better find another way to authenticate ourselves to GitHub that's not plain password.

Let's try that in the next section.

Chapter 3

Using GitHub

Here we'll make a new GitHub account and see how authentication works. This involves some one-time setup.

If you already have a GitHub account, you can skip that section.

If you already have authentication set up with GitHub CLI or with SSH keys, you can skip that section, as well.

3.1 Making a GitHub Account

Head on over to GitHub[1] and click `Sign Up`. Follow those instructions.

Eventually you'll end up on your home screen dashboard.

3.2 Creating a New Repo on GitHub

This will make a repository on GitHub that you own. It does not make a local repository—you'll have to clone the repo for that, something we'll do later.

In GitHub, there's a blue `New` button on the left of the dashboard.

Also, there's a `+` pulldown on the upper right center that has a "New Repository" option. Click one of those.

On the subsequent page:

1. Enter a "Repository name", which can be anything as long as you don't already have a repo by that name. Let's use `test-repo` for this example.
2. Check the "Add a README file" checkbox.

   (In the future, you might already have a local repo you're going to push to this new repo. If that's the case, do **not** check this box or it'll prevent the push from happening.)
3. Click `Create repository` at the bottom.

And there you have it.

3.3 Authentication

Before we get to cloning, let's talk authentication. In the previous part of the intro, we say that username/password logins were disabled, so we have to do something different.

There are a couple options:

[1] https://github.com/

- Use a tool called GitHub CLI
- Use SSH keys

GitHub CLI is likely easier. SSH keys are geektacular.

If you already have authentication working with GitHub, skip these sections.

3.3.1 GitHub CLI

This is a command line interface to GitHub. It does a number of things, but one of them is providing an authentication helper so you can do things like actually push to a remote repo.

Visit the GitHub CLI page[2] and follow the installation instructions. If you're using WSL, Linux, or another Unix variant, see their installation instructions[3] for other platforms.

Once you have it installed, you should be able to run 'gh –version' and see some version information, e.g.:

```
$ gh --version
gh version 2.42.1 (2024-01-15)
https://github.com/cli/cli/releases/tag/v2.42.1
```

Then you'll want to run the following two commands:

```
$ gh auth setup-git
$ gh auth login
```

The first is one-time only.

The second command will take you through the login process. You'll have to do this again if you log out.

When choosing the authentication type between SSH and HTTPS, you can choose HTTPS or SSH. Remember your choice when you go to clone a repo later.

3.3.2 SSH Keys

This is more involved, but has slightly more geek cred and doesn't require you to install GitHub CLI.

If you already have an SSH keypair, you can skip the key generation step. You'd know you had one if you ran `ls ~/.ssh` and you saw a file like `id_rsa.pub` or `id_ed25519.pub`.

To make a new keypair, run the following command:

```
$ ssh-keygen -t ed25519 -C youremail@example.com
```

(The `-C` sets a "comment" in the key. It can be anything, but an email address is common.)

This results in a lot of prompts, but you can just hit ENTER for all of them.

> Best practice is to use a password to access this key, otherwise anyone with access to the private key can impersonate you and access your GitHub account, and any other account you have set up to use that key. But it's a pain to type the password every time you want to use the key (which is any time you do anything with GitHub from the command line), so people use a *key agent* which remembers the password for a while.
>
> If you don't have a password on your key, you're relying on the fact that no one can get a copy of the private portion of your key that's stored on your computer. If you're confident that your computer is secure, then you don't need a password on the key.
>
> Setting up the key agent is outside the scope of this document, and the author in unsure of how it even works in WSL. GitHub has documentation on the matter[a].

[2] https://cli.github.com/
[3] https://github.com/cli/cli#installation

For this demo, we'll just leave the password blank. All of this can be redone with a new key with a password if you choose to do that later.

[a] https://docs.github.com/en/authentication/connecting-to-github-with-ssh/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent

Anyway, just hitting ENTER for all the prompts gets you something like this:

```
Generating public/private ed25519 key pair.
Enter file in which to save the key (/home/user/.ssh/id_ed25519):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in id_ed25519
Your public key has been saved in id_ed25519.pub
The key fingerprint is:
SHA256:/lrT43BQBRPJpUXxpTBFInhdtZSQjQwxU4USwt5c0Lw user@locahost
The key's randomart image is:
+--[ED25519 256]--+
|        .o.X^%^=+|
|        ..oo*^.=o|
|        ..o = o..|
|         . + E   |
|        S .      |
|       .   o     |
|        . + +    |
|         o = .   |
|        ... .    |
+----[SHA256]-----+
```

Important Note: If you chose any file name other than the default for your key, you'll have to do some additional configuration to get it to work with GitHub[a].

[a] https://www.baeldung.com/linux/ssh-private-key-git-command

Unimportant Note: What's that randomart thing with all the weird characters? It's a visual representation of that key. There are ways to configure SSH so that you see the randomart every time you log in, say. And the idea is that if one day you see it looks different, something could be amiss security-wise. I doubt most people every look at it again once it's been generated, though.

Now if you type `ls ~/.ssh` you should see something like this:

```
id_ed25519    id_ed25519.pub
```

The first file is your *private key*. This is never to be shared with anyone. You have no reason to even copy it.

The second file is your *public key*. This can be freely shared with anyone, and we're going to share it with GitHub in a second so that you can log in with it.

If you have trouble in the following sections, try running these two commands:

```
$ chmod 700 ~/.ssh
$ chmod 600 ~/.ssh/*
```

You only have to do that once, but SSH can be a bit picky if the file permissions on those files aren't locked down.

Now in order to make this work, you have to tell GitHub what your public key is.

First, get a copy of your public key in the clipboard. ***Be sure you're getting the file with the `.pub` extension!***

```
$ cat ~/.ssh/id_ed25519.pub
```

You should see something like this:

```
ssh-ed25519 AAAAC3N[a bunch of letters]V+znpoO youremail@example.com
```

Copy the entire thing into the clipboard so you can paste it later.

Now go to GitHub, and click on your icon in the upper right.

Choose "Settings".

Then on the left, choose "SSH and GPG keys".

Click "New SSH Key".

For the title, enter something identifying, like, "My laptop key".

Key type is "Authentication Key".

Then paste your key into the "Key" field.

And click "Add SSH key".

We'll be using SSH to clone URLs later. Remember that.

3.4 Make a Local Clone of the Repo

We need to figure out the URL to the repo so we can clone it.

If you click on your icon in the upper right, then "My Repositories", you should see a page with all your repos. At this point, it might just be your `test-repo` repo. Click on the name.

And you should then be on the repo page. You can browse the files here, among other things, but really we want to get the clone URL.

Click the big blue "Code" button.

What you do next depends on if you're using GitHub CLI or SSH keys.

3.4.1 Cloning from GitHub with GitHub CLI

You have two options.

- **Option 1**: Earlier when we authenticated with `gh auth login` I said to remember if you chose HTTPS or SSH. Depending on which you chose, you should choose that tab on this window.

  Copy the URL.

  Go to the command line and run `git clone [URL]` where `[URL]` is what you copied. So it'll be this for HTTPS:

```
$ git clone https://github.com/user/test-repo.git
```

or this for SSH:

```
$ git clone git@github.com:user/test-repo.git
```

- **Option 2**: Choose the "GitHub CLI" tab. Run the command as they have it, which will be something like:

```
$ gh repo clone user/test-repo
```

3.4.2 Cloning from GitHub with SSH Keys

If you set up an SSH key earlier, you can use this method.

After hitting the blue "Code" button, make sure the "SSH" tab is selected.

Copy that URL.

Go to the command line and run `git clone [URL]` where `[URL]` is what you copied. So it'll be something like this:

```
$ git clone git@github.com:user/test-repo.git
```

3.5 Make Changes and Push!

Now that you've cloned the repo, you should be able to `cd` into that directory, edit a file, `git add` it to the stage, then `git commit -m message` to make a commit…

And then `git push` to push it back to the clone on GitHub!

And after that if you go to the repo page on GitHub and hit reload, you should be able to see your changes there!

And now we're back to that standard common flow:

1. *Clone* a *remote* repo.
2. Make some local changes.
3. Add those changes to the *stage*.
4. *Commit* those changes.
5. *Push* your changes back to the remote repo.
6. Go back to Step 2.

3.6 Collaboration on GitHub

There are two main techniques for this:

1. Fork/pull request
2. Add a collaborator

We'll talk about the first one in the future.

For now, the easiest way to add collaborators is to just add them to your repo.

On the repo page on GitHub, choose "Settings", then "Collaborators" on the left.

After authenticating, you can click "Add people". Enter the username of the person you want to collaborate with.

They'll have to accept the invitation from their GitHub inbox, but then they'll have access to the repo.

Be sure to only do this with people you trust!

Chapter 4

The Git Log and `HEAD`

When we make commits to a Git repo, it tracks each of those commits in a log that you can visit. Let's take a look at that now.

4.1 An Example Log

You can get the commit log by typing `git log`.

Let's say I'm in a repo with a single commit, where I've just added a file with the commit message "Added".

```
$ git log
```

produces:

```
commit 5a02fede3007edf55d18e2f9ee3e57979535e8f2 (HEAD -> main)
Author: User Name <user@example.com>
Date:   Thu Feb 1 09:24:52 2024 -0800

    Added
```

If I make another commit, we get a longer log:

```
commit 5e8cb52cb813a371a11f75050ac2d7b9e15e4751 (HEAD -> main)
Author: User Name <user@example.com>
Date:   Thu Feb 1 12:36:13 2024 -0800

    More output

commit 5a02fede3007edf55d18e2f9ee3e57979535e8f2
Author: User Name <user@example.com>
Date:   Thu Feb 1 09:24:52 2024 -0800

    Added
```

Notice that the most recent commit entry is at the top of the output.

4.2 What's in the log?

There are a few things to notice in the log:

- The commit comment

- The date
- The user who made the commit

Also we have those huge hex[1] numbers after the word `commit`.

This is the *commit ID* or *commit hash*. This is universally unique number that you can use to identify a particular commit.

Normally you don't need to know this, but it can be useful for going back in time or keeping track of commits in multideveloper projects.

We also see a bit at the top that says `(HEAD -> main)`. What's that about?

4.3 The `HEAD` Reference

We've seen that each commit has a unique and unwieldy identifier like this:

`5a02fede3007edf55d18e2f9ee3e57979535e8f2`

Luckily, there are a few ways to refer to commits with more human symbolic names.

`HEAD` is one of these references. It indicates which branch or commit you're looking at right now in your project subdirectory. Remember how we said you could go look at previous commits? The way you do that is by moving `HEAD` to them.

> We haven't talked about branches yet, but the `HEAD` normally refers to a branch. By default, it's the `main` branch. But since we're getting ahead of ourselves, I'm going to just keep saying that `HEAD` refers to a commit, even though it usually does it indirectly via a branch.
>
> So this is a bit of a lie, but I hope you forgive me.

Some terminology: the files in your git subdirectory you're looking at right now is referred to as your *working tree*. The working tree is the files as they appear at the commit pointed to by `HEAD`, plus any uncommitted changes you might have made.

So if you switch `HEAD` to another commit, the files in your working tree will be updated to reflect that.

Okay, then, how do we know which commit `HEAD` is referring to? Well, it's right the at the top of the log:

```
commit 5e8cb52cb813a371a11f75050ac2d7b9e15e4751 (HEAD -> main)
Author: User Name <user@example.com>
Date:   Thu Feb 1 12:36:13 2024 -0800

    More output
```

We see `HEAD` right there on the first line, indicating that `HEAD` is referring to commit with ID:

`5e8cb52cb813a371a11f75050ac2d7b9e15e4751`

Again that's a bit of a lie, though. The `HEAD -> main` means that `HEAD` is actually referring to the `main` branch, and that `main` is referring to the commit. `HEAD` is therefore indirectly referring to the commit. More on that later.

4.4 Going Back In Time and Detached `HEAD`

Here's my full Git log:

```
commit 5e8cb52cb813a371a11f75050ac2d7b9e15e4751 (HEAD -> main)
Author: User Name <user@example.com>
```

[1] https://en.wikipedia.org/wiki/Hexadecimal

```
Date:   Thu Feb 1 12:36:13 2024 -0800

    More output

commit 5a02fede3007edf55d18e2f9ee3e57979535e8f2
Author: User Name <user@example.com>
Date:   Thu Feb 1 09:24:52 2024 -0800

    Added
```

If I look at the files, I'll see the changes indicated by the "More output" commit. But let's say I want to go back in time to the previous commit and see what the files looked like then. How would I do that?

> Maybe there were some changes that existed back in an earlier commit that had been since removed, and you wanted to look at them, for example.

I can use the `git checkout` command to make that happen.

Let's checkout the first commit, the one with ID `5a02fede3007edf55d18e2f9ee3e57979535e8f2`.

Now, I could say:

```
$ git checkout 5a02fede3007edf55d18e2f9ee3e57979535e8f2
```

and that would work, but the rule is that you must specific at least 4 unique digits of the ID, so I could have also done this:

```
$ git checkout 5a02
```

for the same result.

And that result is:

```
Note: switching to '5a02'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by switching back to a branch.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -c with the switch command. Example:

  git switch -c <new-branch-name>

Or undo this operation with:

  git switch -

Turn off this advice by setting config variable advice.detachedHead to false

HEAD is now at 5a02fed Added
```

Looks sort of scary, but look—Git is telling us how to undo the operation if we want, and so there's really nothing to fear.

Let's take a look around with `git log`:

```
commit 5a02fede3007edf55d18e2f9ee3e57979535e8f2 (HEAD)
Author: Brian "Beej Jorgensen" Hall <beej@beej.us>
Date:   Thu Feb 1 09:24:52 2024 -0800

    Added
```

That's all! Just one commit?! Where's the second commit I made? Is it gone forever?!

No. Everything is fine.

When you have `HEAD` at a certain commit, you're looking at the world as it looked at that snapshot in time. Future commits haven't "happened" yet from this perspective. They are still out there, but you'll have to change back to them by name.

Also, do you see anything different about that first line that reads `(HEAD)`? That's right: no `main` to be seen.

That's because the `main` branch is still looking at the latest commit, the one with the "More output" comment. So we don't see it from this perspective.

> Remember earlier when I said it was a bit of a lie to say that `HEAD` points to a commit? Well, detached head state is the case where it actually **does**. Detached head state is just what happens when `HEAD` is pointing to a commit instead of a branch. To reattach it, you have to change it to point to a branch again.

Let's get back to the `main` branch. There are three options:

1. `git switch -`, just like the helpful message says.
2. `git switch main`
3. `git checkout main`

Git replies:

```
Previous HEAD position was 5a02fed Added
Switched to branch 'main'
```

And now if we `git log`, we see all our changes again:

```
commit 5e8cb52cb813a371a11f75050ac2d7b9e15e4751 (HEAD -> main)
Author: User Name <user@example.com>
Date:   Thu Feb 1 12:36:13 2024 -0800

    More output

commit 5a02fede3007edf55d18e2f9ee3e57979535e8f2
Author: User Name <user@example.com>
Date:   Thu Feb 1 09:24:52 2024 -0800

    Added
```

and our working tree will be updated to show the files as they are in the `main` commit.

4.5 The New Command: `git switch`

In ye olden days, `git checkout` did a lot of things, and it still does. Because it does so much, the maintainers of Git have been trying to break some of that functionality into a new command, `git switch`.

We could redo the previous section by using just `git switch` instead of `git checkout`. Let's try:

```
$ git switch 5a02
```

and it says:

```
fatal: a branch is expected, got commit '5a02'
hint: If you want to detach HEAD at the commit, try again with the
      --detach option.
```

Hmmm! `git switch` is warning us that we're about to go into detached head state, and is that what we really want? It's not a crime or anything to do so, but it's just letting us know that we're not going to be on a branch any longer.

So we can override, just like it suggests:

```
$ git switch --detach 5a02
HEAD is now at 5a02fed Added
```

All right! No big message about being detached, but we don't need it because we know it's detached since we specified.

And like before, we can get back to the `main` branch with either:

1. `git switch -`, switch to the previous state
2. `git switch main`

Easy.

Chapter 5

Branches and Fast-Forward Merges

5.1 What is a Branch?

Normally you think of writing code as a linear sequence of changes. You start with an empty file, add some things, test them, add some more, test some more, and eventually the code is complete.

Figure 5.1: A simple commit graph.

In Git we might think of this as a sequence of commits. Let's look at a graph (Figure 5.1) where I've numbered commits 1-5. There, `(1)` was the first commit we made on the repo, `(2)` is some changes we made on top of `(1)`, and `(3)` is some changes we made on top of `(2)`, etc.

Git always keeps track of the parent commit for any particular commit, e.g. it knows the parent commit of `(3)` is `(2)` in the above graph. In this graph, the parent relationship is indicated by an arrow. "The parent of commit 3 is commit 2", etc. It's a little confusing because clearly commit 3 came *after* commit 2 in terms of time, but the arrow points to the parent, which is the opposite of the nodes' temporal relationship.

A *branch* is like a name tag stuck on one **specific** commit. You can move the name tag around with various Git operations.

The default branch is called `main`.

> The default branch used to be called `master`, and still is called that in some older repos.

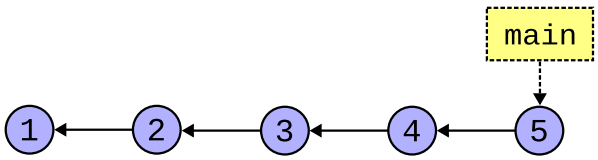

Figure 5.2: The main branch on a commit.

So to make it a little more complete, we can show that branch in Figure 5.2. There's our `main` branch attached to the commit labeled `(5)`.

> It's tempting to think of the whole sequence of commits as "the branch", but this author recommends against it. Better to keep in mind that the branch is just a name tag for a single commit, and that we can move that name tag around.

But Git offers something more powerful, allowing you (or collaborators) to pursue multiple branches simultaneously.

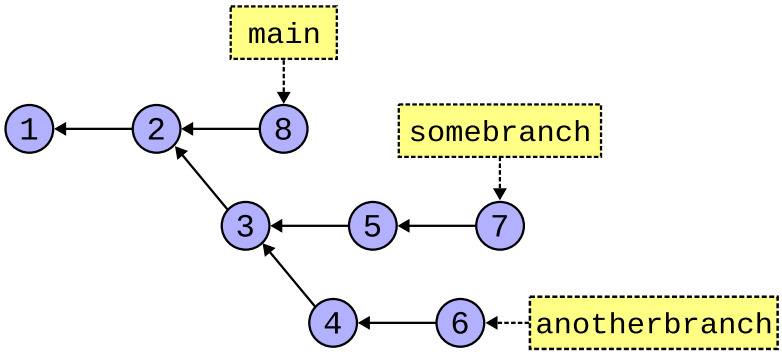

Figure 5.3: Lots of branches.

So there might be multiple collaborators working on the project at the same time.

And then, when you're ready, you can *merge* those branches back together. In this diagram we've merged commit 6 and 7 into a new commit, commit 9. In Figure 5.4, commit 9 contains the changes of both commits 7 and 6.

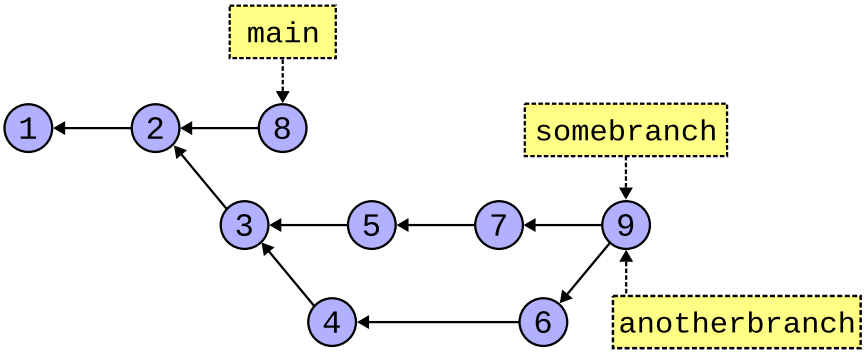

Figure 5.4: After merging `somebranch` and `anotherbranch`.

In that case, `somebranch` and `anotherbranch` both point to the same commit. There's no problem with this.

And then we can keep merging if we want, until all the branches are pointing at the same commit (Figure 5.5).

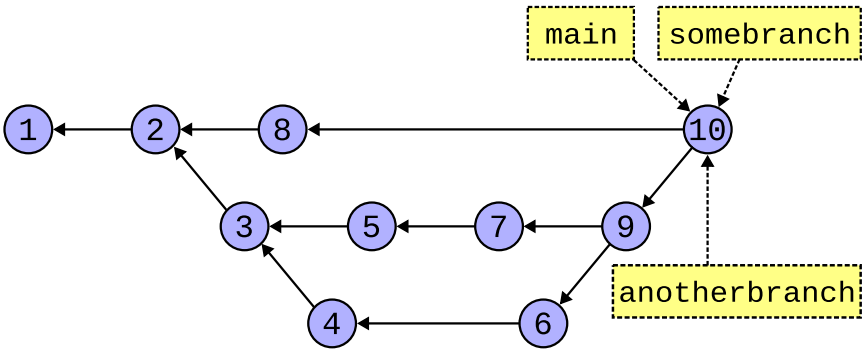

Figure 5.5: After merging all branches.

And maybe after all this we decide to delete `somebranch` and `anotherbranch`; we can do this safely because they're fully merged, and can do this without affecting `main` or any commits (Figure 5.6).

This chapter is all about getting good with branching and partially good with merging.

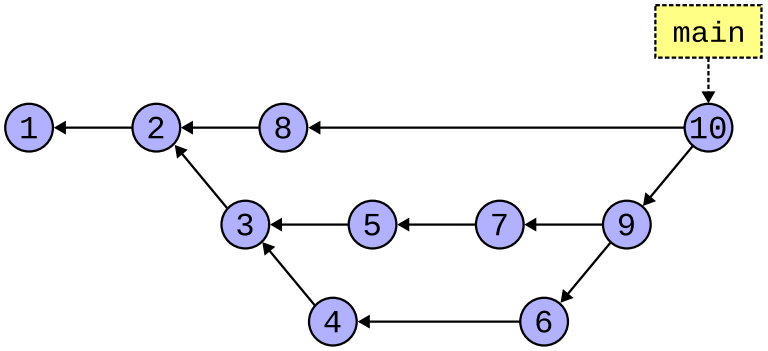

Figure 5.6: After deleting merged branches.

5.2 A Quick Note about `git pull`

When you do a pull, it actually does two things: (a) *fetch* all the changes from the remote repo and (b) *merge* those changes.

If two or more people are committing to the same branch, eventually `git pull` is going to have to merge. And it turns out there are a few ways it can do this.

For now, we're going to tell `git pull` to always classically merge divergent branches, and you can do that with this one-time command:

```
$ git config --global pull.rebase false
```

If you don't do that, Git will pop up an error message complaining about it the first time it has to merge on a pull. And you'll have to do it then.

When we talk about rebasing later, this will make more sense.

5.3 `HEAD` and Branches

We said earlier that `HEAD` refers to a specific commit, namely the commit you're looking at right now in your working tree.

And we also said that was a bit of a lie.

In normal usage, `HEAD` points to a branch, not to a commit. In detached head state, `HEAD` points to a commit.

It's like Figure 5.7 when `HEAD` is pointing to a branch as per normal.

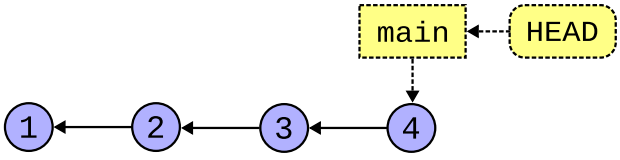

Figure 5.7: `HEAD` pointing to a branch.

But if we check out an earlier commit that doesn't have a branch, we end up in detached head state, and it looks like Figure 5.8.

So far, we've been making commits on the `main` branch without really even thinking about branching. Recalling that the `main` branch is just a label for a specific commit, how does the `main` branch know to "follow" our `HEAD` from commit to commit?

It does it like this: the branch the `HEAD` points to follows the current commit. That is, when you make a commit, the branch `HEAD` points to moves along to that next commit.

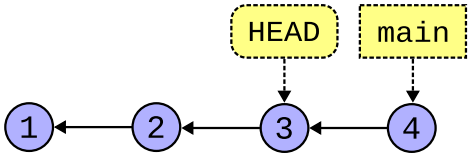

Figure 5.8: `HEAD` pointing to a commit.

If we were here back at Figure 5.7, when `HEAD` was pointing to the `main` branch, we could make one more commit and get us to Figure 5.9.

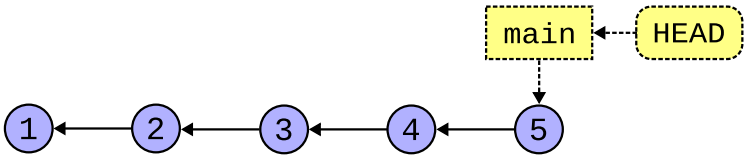

Figure 5.9: `HEAD` moving with a branch.

Contrast that to detached head state, back in Figure 5.8. If we were there, a new commit would get us to Figure 5.10, leaving `main` alone.

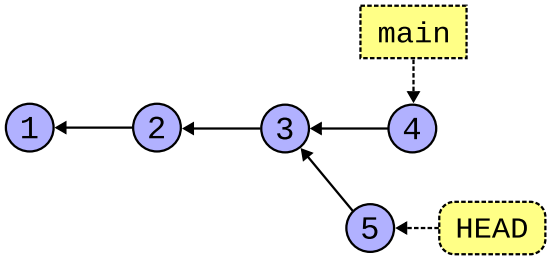

Figure 5.10: A commit with detached `HEAD`.

At this point, there's nothing stopping you from creating a new branch at the same commit as `HEAD`, if you want to do that. Or maybe you are just messing around and decide to switch back to `main` later, abandoning the commits you've made in detached `HEAD` state.

Now that we have the abstract theory stuff laid out, let's talk specifics.

5.4 Creating a Branch

When you make the first commit to a new repo, the `main` branch is automatically created for you at that commit.

But what about new branches we want to make?

> Why make a branch? A common case is that you want to work on your own commits without impacting the work of others. (In this case you're really just putting off the work until you merge your branch with theirs, but it's a good workflow.)
>
> Another case is that you want to mess around with some changes but you're not sure if they'll work. If they end up not working, you can just delete the branch. If they do work, you can merge your changes back into the non-messing-around branch.

The most common way to make new branches is this:

1. Switch to the commit or branch from which you want to make the new branch.
2. Make the new branch there and switch `HEAD` to point to the new branch.

Let's try it. Let's branch off `main`.

You might already have `main` checked out (i.e. `HEAD` points to `main`), but let's do it again to be safe, and then we'll create a branch with `git switch`:

```
$ git switch main
$ git switch -c newbranch
```

Normally you can just switch to another branch (i.e. have `HEAD` point to that branch) with `git switch branchname`. But if the branch doesn't exist, you use the `-c` switch to create the branch before switching to it.

> **ProTip**: make sure all your local changes are committed before switching branches! If you `git status` it should say "working tree clean" before you switch. Later we'll learn about another option with `git stash`.

So after checking out `main`, we have Figure 5.11.

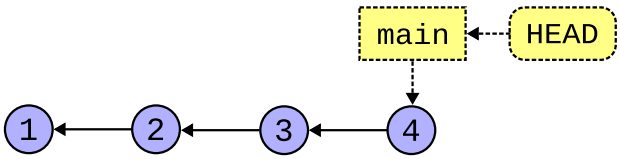

Figure 5.11: `HEAD` pointing to `main`.

And then with `git switch -c newbranch`, we create and switch to `newbranch`, and that gets us to Figure 5.12.

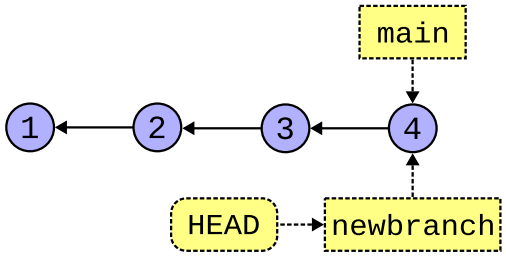

Figure 5.12: `HEAD` pointing to `newbranch`.

That's not super exciting, since we're still looking at the same commit, but let's see what happens when we make some new commits on this new branch.

> **Important note**: the branches we're making here exist only on your local clone; they're not automagically propagated back to wherever you cloned the repo from.
>
> The upshot is that if you accidentally (or deliberately) delete your local repo, when you `git clone` again, all your local branches will be gone (along with any commits that aren't part of `main` or any other branches pushed to the server).
>
> There is a way to set up that connection where your local branches are uploaded when you push, called *remote-tracking branches*. `main` is an example of a remote-tracking branch, which is why `git push` from `main` works while `git push` from `newbranch` gives an error. But we'll talk about all this later.

5.5 Make Some Commits on a Branch

This is not really that different than what we were doing with our commits before. Before we made a branch, we had `HEAD` pointing to `main`, and we were making commits on `main`.

Now we have `HEAD` pointing to `newbranch` and our commits will go there, instead.

Right after creating `newbranch`, we had the situation in Figure 5.12. Now let's edit something in the working tree and make a new commit. With that, we'll have the scenario in Figure 5.13.

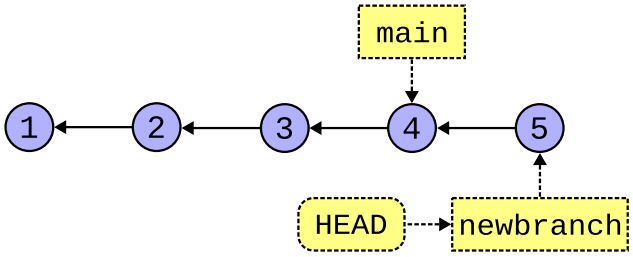

Figure 5.13: Adding a new commit to `newbranch`.

Right? Let's make another commit and get to Figure 5.14.

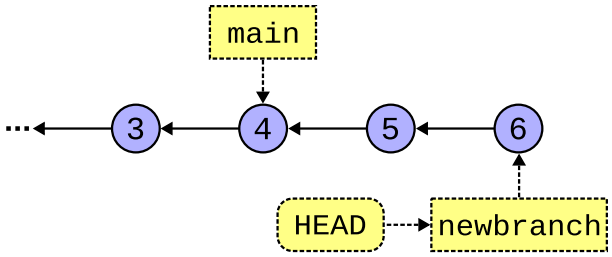

Figure 5.14: Adding another commit to `newbranch`.

We can see that `newbranch` and `main` are pointing at different commits.

> If we wanted to see the state of the repo from `main`'s perspective, what would we have to do? We'd have to `git switch main` to look at that branch.

Now for another question. Let's say we've decided that we're happy with the changes on `newbranch`, and we want to merge them into the code in the `main` branch. How would we do that?

5.6 Merging: Fast-Forward

Bringing two branches back into sync is called *merging*.

The branch you're on is the branch you're bringing other changes *into*. That is, if you're on Branch A, and you tell give to Merge Branch B, Branch B's changes will be applied onto Branch A. (Branch B remains unchanged in this scenario.)

But in this section we're going to be talking about a specific kind of merge: the *fast-forward*. This occurs when the branch you're merging from is a direct ancestor of the branch you're merging into.

Let's say we have `newbranch` checked out, like from the previous example in Figure 5.14.

I decide I want to merge `main`'s changes into `newbranch`, so (again, having `newbranch` checked out):

```
$ git merge main
Already up to date.
```

Nothing happened? What's that mean? Well, if we look at the commit graph, above, all of `main`'s changes are already in `newbranch`, since `newbranch` is a direct ancestor.

Git is saying, "Hey, you already have all the commits up to `main` in your branch, so there's nothing for me to do."

But let's reverse it. Let's check out `main` and then merge `newbranch` into it.

```
$ git switch main
```

Now we've moved `HEAD` to track `main`, as shown in Figure 5.15.

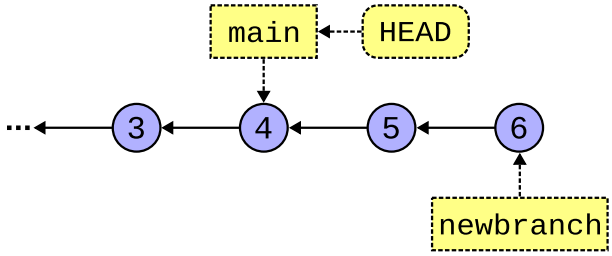

Figure 5.15: Checking out `main` again.

And `newbranch` is **not** a direct ancestor of `main` (it's a descendant). So `newbranch`'s changes are **not** yet in `main`.

So let's merge them in and see what happens (your output may vary depending on what files are included in the merge):

```
$ git merge newbranch
Updating 087a53d..cef68a8
Fast-forward
 foo.py | 4 +++-
 1 file changed, 3 insertions(+), 1 deletion(-)
```

And now we're at Figure 5.16.

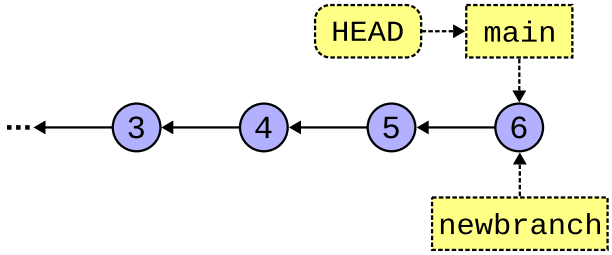

Figure 5.16: After merging `newbranch` into `main`.

Wait a second—didn't we say to merge `newbranch` into `main`, like take those changes and fold them into the `main` branch? Why did `main` move, then?

We did! But let's stop and think about how this can happen in the special case where the branch you're merging *into* is a direct ancestor of the branch you're merging *from*.

It used to be that `main` didn't have commits `(5)` or `(6)` in the graph, above. But `newbranch` has already done the work of adding `(5)` and `(6)`!

The easiest way to get those commits "into" `main` is to simply *fast-forward* `main` up to `newbranch`'s commit!

Again, this only works when the branch you're merging into is a direct ancestor of the branch you're merging from.

That said, you certainly can merge branches that are not directly related like that, e.g. branches that share a common ancestor but have both since diverged.

Git will automatically fast-forward if it can. Otherwise it does a "real" merge. And while fast-forward merges can never lead to *merge conflicts*, regular merges certainly can.

But that's another story.

5.7 Deleting a Branch

If you're done merging your branch, it's easy to delete it. **Importantly this doesn't delete any commits; it just deletes the branch "label" so you can't use it any longer**. You can still use all the commits.

Let's say we've finished the work on our `topic1` branch and we want to merge it into `main`. No problem:

```
$ git commit -m "finished with topic1"   # on topic1 branch
$ git switch main
$ get merge topic1                       # merge topic1 into main
```

At this point, assuming a completed merge, we can delete the `topic` branch:

```
$ git branch -d topic1
Deleted branch topic1 (was 3be2ad2).
```

Done!

But what if you were working on a branch and wanted to abandon it before you merge it into something? For that, we have the more imperative Capital `D` option, which means, "I *really* mean it. Delete this unmerged branch!"

```
$ git branch -D topic1
```

Use lowercase `-d` unless you have reason to do otherwise. It'll at least tell you if you're about to lose your reference to your unmerged commits!

Chapter 6

Merging and Conflicts

We've seen how a fast-forward merge can bring to branches into sync with no possibility of conflict.

But what if we can't fast-forward because two branches are not direct ancestors? In other words, what if the branches have *diverged*?

6.1 An Example of Divergent Branches

Let's look at a commit graph where things are still OK to fast-forward in Figure 6.1.

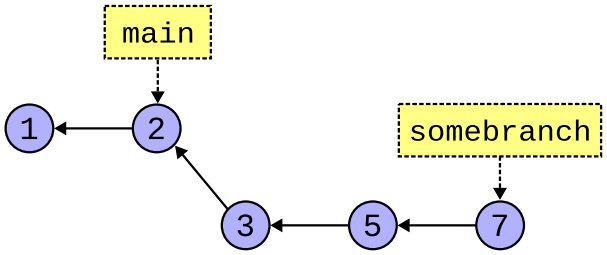

Figure 6.1: A direct ancestor branch.

Yes, I've bent the graph a bit there, but we can merge `somebranch` into `main` as a fast-forward because `main` is a direct ancestor and `somebranch` is therefore a direct descendant.

But what if, **before** we merged, someone made another commit on the `main` branch? And now it looks like it does in Figure 6.2.

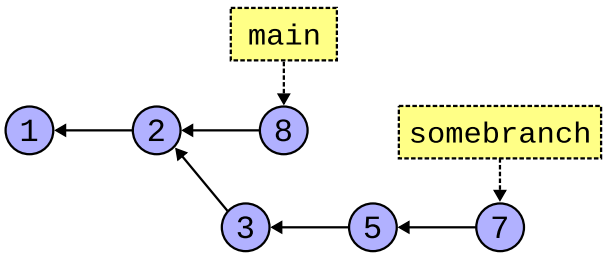

Figure 6.2: Not a direct ancestor branch.

There's a common ancestor at commit `(2)`, but there's no direct line of descent. `main` and `somebranch` have diverged.

Is all hope lost? How can we merge?

6.2 Merging Divergent Branches

Turns out you do it the exact same way as always.

1. Check out the branch you want to merge *into*.
2. `git merge` the branch you want to merge *from*.

In our Figure 6.2 example above, let's say we've done this:

```
$ git switch main
$ git merge somebranch    # into main
```

> The `#` is a shell comment delimiter. You can paste that in if you want, but it does nothing.

The difference here is that Git can't simply fast-forward. It has to somehow, magically, bring together the changes from commit `(7)` **and** commit `(8)` even if they're radically different than one other.

This means that after we bring those two commits together, the code will look like it's never looked before, a combination of two sets of changes.

And because it looks like it hasn't before, we need *another commit* (another snapshot of the working tree) to represent the joining of both sets of changes.

We call this the *merge commit*, and Git will automatically make it for you. (When this happens, you'll see an editor pop up with some text in it. This text is the commit message. Edit it (or just accept it as-is) and save the file and exit the editor. See Getting Out of Editors if you need help with this.

So after our merge, we end up with Figure 6.3.

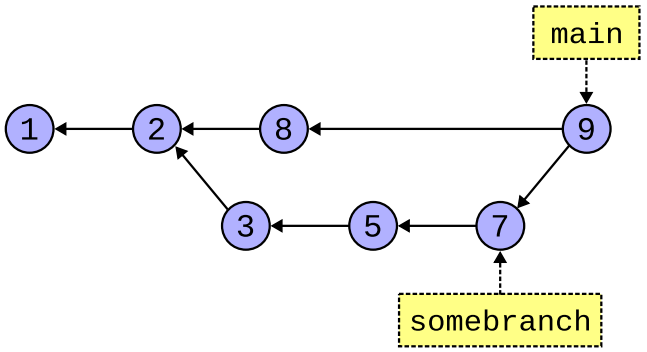

Figure 6.3: Creating a merge commit.

Commit labeled `(9)` is the merge commit. It contains both the changes from `(8)` and `(7)`. And has the commit message you saved in the editor.

And we see `main` has been updated to point to it. And that `somebranch` is unaffected.

Importantly, we see that commit `(9)` has **two parents**, the commits that were merged together to make it.

And look! If we want, we can now fast-forward `somebranch` to `main` because it's now a direct ancestor!

In this example, Git was able to determine how to do the merge automatically. But there are some cases where it cannot, and this results in a *merge conflict* that requires manual intervention. By you.

6.3 Merge Conflicts

If two branches have changes that are "far apart" from one another, Git can figure it out. If I edit line 20 of a file in one branch, and you edit line 3490 of the same file in another, Git can bring both edits in automatically.

But let's say I edit line 20 in one commit, and you edit line 20 (the same line) in another commit.

Which one is “right”? Git has no idea because it’s just dumb software and doesn’t know our business needs.

So it asks us, during the merge, to fix it. After we fix it, Git can complete the merge.

> **There’s an important point here.** When you’re merging, if a conflict occurs, *you’re still merging*. Git is is the “merge” state, waiting for more merge-specific commands.
>
> You can resolve the conflict then commit the changes to complete the merge. Or you can back out of the merge making as if you’d never started it in the first place.
>
> The important point is that you’re aware Git is in a special state and you have to either complete (or abort) the merge to get back to normal before you continue to use it.

Let’s have an example where both `main` and `newbranch` have added a line to end of file, i.e. they both added line 4. Git doesn’t know which one is correct, so there’s a conflict.

```
$ git merge newbranch
Auto-merging foo.py
CONFLICT (content): Merge conflict in foo.py
Automatic merge failed; fix conflicts and then commit the result.
```

Now if I look at my status, I see we’re in merge state, as noted by `You have unmerged paths`. We’re in the middle of merge; we have to either go out the front or back out the back to get back to normal.

```
$ git status
On branch main
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)
    both modified:   foo.py

no changes added to commit (use "git add" and/or "git commit -a")
```

It’s also hinting that I can do one of two things:

1. Fix conflicts and run “git commit”.
2. Use “git merge –abort” to abort the merge.

The second just rolls back the merge making it as if I hadn’t run `git merge` in the first place.

So let’s focus on the first. What are these conflicts and how do I resolve them?

6.4 What a Conflict Looks Like

My error message above is telling me that `foo.py` has unmerged paths. So look at what’s happened with that file.

Before I started any of this, the file `foo.py` only had this in it on branch `main`:

```
print("Commit 1")
```

And I added a line so it looked like this:

```
print("Commit 1")
print("Commit 4")
```

And committed it.

But what I didn't realize was that my teammate had also made another commit on `newbranch` that added different lines to the bottom of the file.

So when I went to merge `newbranch` into `main`, I got this conflict. Git doesn't know which additional lines are correct.

Here's where the fun begins. Let's edit `foo.py` here in the middle of the merge and see what it looks like:

```
print("Commit 1")
<<<<<<< HEAD
print("Commit 4")
=======
print("Commit 2")
print("Commit 3")
>>>>>>> newbranch
```

What the giblets is all that? Git has totally screwed with the contents of my file!

Yes, it has! But not for no reason; let's examine what's in there.

We have three delimiters: `<<<<<<`, `======`, and `>>>>>>`.

Everything from the top delimiter to the middle one is what's in `HEAD` (the branch you're on and merging *into*).

Everything from the middle delimiter to the bottom one is what's in `newbranch` (the branch you're merging *from*).

So Git has "helpfully" given us the information we need to make a semi-informed decision about what to do.

And here's exactly the steps we must follow:

1. Edit the conflicting file(s), remove all those extra lines, and **make the file(s) Right**.
2. Do a `git add` to add the file(s).
3. Do a `git commit` to finalize the merge.

Now, when I say "make the file *Right*", what does that mean? It means that I need to have a chat with my teammate and figure out what this code is supposed to do. We clearly have different ideas, and only one of them is right.

So we have a chat and hash it out. We finally decide the file should look like this:

```
print("Commit 1")
print("Commit 4")
print("Commit 3")
```

And then I (since I'm the one doing the merge), edit `foo.py` and remove all the merge delimiters and everything else, and make it look exactly like we agreed upon. I make it look *Right*.

Then I add the file to the stage and make a merge commit. (Here we're manually making the merge commit, unlike above where Git was able to automatically make it.)

```
$ git add foo.py
$ git status
On branch main
All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)

Changes to be committed:
```

```
      modified:   foo.py
```

Notice that `git status` is telling me we're still in the merging state, but I've resolved the conflicts. It tells me to `git commit` to finish the merge.

So let's do that, making the merge commit:

```
$ git commit -m "Merged with newbranch"
[main 668b506] Merged with newbranch
```

And that's it! Let's check status just to be sure:

```
$ git status
On branch main
nothing to commit, working tree clean
```

Success!

Just to wrap up, let's take a look at the log at this point:

```
$ git log
commit 668b5065aa803fa496951b70159474e164d4d3d2 (HEAD -> main)
Merge: e4b69af 81d6f58
Author: User Name <user@example.com>
Date:   Sun Feb 4 13:18:09 2024 -0800

    Merged with newbranch

commit e4b69af05724dc4ef37594e06d0fd323ca1b8578
Author: User Name <user@example.com>
Date:   Sun Feb 4 13:16:32 2024 -0800

    Commmit 4

commit 81d6f58b5982d39a1d92af06b812777dbb452879 (newbranch)
Author: User Name <user@example.com>
Date:   Sun Feb 4 13:16:32 2024 -0800

    Commmit 3

commit 3ab961073374ec26734c933503a8aa988c94185b
Author: User Name <user@example.com>
Date:   Sun Feb 4 13:16:32 2024 -0800

    Commmit 1
```

We see a few things. One is that our merge commit is pointed to by `main` (and `HEAD`). And looking down a couple commits, we see our now-direct ancestor, `newbranch` back on Commit 3.

We also see a `Merge:` line on that top commit. It lists the UUIDs for the two commits that it came from (the first 7 digits, anyway).

6.5 Why Merge Conflicts Happen

Generally, it's because you haven't coordinated with your team about who is responsible for which pieces of code. Generally two people shouldn't be editing the same lines of code in the same file at once.

That said, there are absolutely cases where it does happen and is expected. The key is to communicate with your team when resolving the conflict if you don't know what is *Right*.

6.6 Merging with IDEs or other Merge Tools

IDEs like VS Code might have a special merge mode where you can choose one set of changes or another, or both. Likely "both" is what you want, but make an informed decision on the matter.

Also, even when selecting "both", it could be that the editor puts them in the wrong order. It's up to you to make sure the file is *Right* before making the final commit to complete the merge.

You can do this by, after the tool has been used to merge, opening the file again in a new window and making sure it's as you want it, and editing it to be if it's not.

6.7 Merge Big Ideas

DON'T PANIC! If you have a merge conflict, you can totally work it out. They're a common occurrence, and the more of them you do, the better at them you get.

Nothing to worry about. Everything is in Git's commit history, so even if you botch it, you can always get things back the way they were.

Chapter 7

Using Subdirectories with Git

This is a shorter chapter, but we want to talk about Git's behavior when it comes to working in subdirectories and some gotchas that you probably don't want to get wrapped up in.

7.1 Repos and Subdirectories

When you run a `git` command, Git looks for a special directory called `.git` ("dot git") in the current directory. As we've already mentioned, this is the directory, created when you create the repo, that holds the metadata about the repo.

But what if you're in a subdirectory in your project, and there's no `.git` directory there?

Git starts by looking in the current directory for `.git`. If it can't find it there, it looks in the parent directory. And if it's not there, it looks in the grandparent, etc., all the way back to the root directory.

7.1.1 What about Subprojects?

One common student question is, "Should I make one single repo for CS101 with subdirectories for each project? Or should I make a different repo for every project?"

Firstly, see if your instructor has a requirement or preference, but other than that, it doesn't technically matter which approach you use.

In real life, bigger repos (much bigger than you'll typically be using for a class) take a lot longer to clone due to their size.

What happens if you initialize a new Git repo *inside* an existing repo? It's not great. Don't do this.

For mixing and matching different repos in the same hierarchy, Git has the concept of submodules[1], but that's out of scope for this guide, and rarely used in school.

7.2 Accidentally Making a Repo in your Home Directory

Git won't stop you from making a repo there, i.e. a repo that contains everything in all your directories.

But that's probably not what you wanted to do.

How does one make this mistake? Usually it's with `git init .` in your home directory. You can also make this error by launching VS Code from your home directory and telling it to "Initialize Repository" in that location.

This is particularly insidious because if you're in a subdirectory that you *think* is a standalone repo, you might have been misled since Git searches parent folder for the `.git` directory and it could be finding the spurious one you accidentally made in your home directory.

[1] https://git-scm.com/book/en/v2/Git-Tools-Submodules

We recommend against one big repo from your home directory. You should have separate subdirectories for each of your repos.

If you accidentally create a repo where you didn't want to, changing a Git repo to a regular subdirectory is as simple as removing the `.git` directory. Be careful that you're removing the correct one when you do this!

> One hack you can do to prevent Git from creating a repo in your home directory is to preemptively put an unwriteable `.git` directory there.
>
> ```
> $ mkdir ~/.git       # Make the .git directory
> $ chmod 000 ~/.git   # Take away all permissions
> ```
>
> This way when Git tries to make its metadata folder there, it'll be stopped because you don't have write permission to that `.git` directory.

7.3 Empty Subdirectories in Repos

Turns out Git doesn't support this. It only tracks files, so if you want a subdirectory represented in your repo, you must have at least one file in it.

A common thing to do is add an empty file called `.gitkeep` ("dot git keep") to the subdirectory, then add it to the repo. This will cause Git to recreate the subdirectory when it clones or merges the `.gitkeep` directory.

> The file `.gitkeep` isn't special in any way, other than convention. The file could be called anything. For example, if you know you'll need to eventually put a `.gitignore` in that directory, you might just use that instead. Or a `README`.

Chapter 8

Ignoring Files with `.gitignore`

What if you have files in your subdirectory you don't want Git to pay any attention to? Like maybe you have some temporary files you don't want to see in the repo. Or maybe you have an executable you built from a C project and you don't want that checked in because your incredibly strict instructor won't grade your project if the repo contains any build products? For example.

That's what this part of the guide is all about.

8.1 Adding a `.gitignore` File

In any directory of a project, you can add a `.gitignore` ("dot gitignore") file.

This is a simple textfile that contains a list of file names to ignore.

Let's say I have a C project that builds and executable called "doom". I wouldn't want to check that into my source repo because it's not source, and it's just a big binary that takes a bunch of disk.

But when I get the status, it's annoying to see Git complaining about it:

```
$ git status
On branch main
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    doom

nothing added to commit but untracked files present (use "git add" to track)
```

So I edit a `.gitignore` file in that directory and add this one line to it:

```
doom
```

Now I run status again:

```
$ git status
On branch main
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    .gitignore

nothing added to commit but untracked files present (use "git add" to track)
```

What? Same thing? Not quite! Read the fine print!

It used to be complaining that `doom` was untracked, but now it's not complaining. So the `.gitignore` worked. Woo hoo!

But Git has found another untracked file in the brand new `.gitignore`. So we should add that to the repo.

Always put your `.gitignore` files in the repo unless you have a compelling reason not to. This way they'll exist in all your clones, which is handy.

```
$ git add .gitignore
$ git commit -m Added
[main 07582ad] Added
 1 file changed, 1 insertion(+)
 create mode 100644 .gitignore
```

Now we get the status:

```
$ git status
On branch main
nothing to commit, working tree clean
```

and we're all clear.

8.2 Can I Specify Subdirectories in `.gitignore`?

Yes!

You can be as specific or as non-specific as you like with file matches.

Here's a `.gitignore` looking for a very specific file:

```
subdir/subdir2/foo.txt
```

That will match anywhere in the project. If you want to only match a specific file from the project root, you can prepend a slash:

```
/subdir/subdir2/foo.txt
```

Note that means `subdir` in the root of the *project*, not the root directory of your entire filesystem.

8.3 Where do I Put the `.gitignore`?

You can add `.gitignore` files to any subdirectories of your project. But how they behave depends on where they are.

The rule is this: *each `.gitignore` file applies to all the subdirectories below it.*

So if you put a `.gitignore` in your project's root directory that has `foo.txt` in it, every single `foo.txt` in every subdirectory of your project will be ignored.

Use the highest-level `.gitignore` file to block things you know you don't want **anywhere** in your project.

If you add additional `.gitignore` files to subdirectories, those only apply to that subdirectory and below.

The idea is that you start with the most broadly applicable set of ignored files in your project root, and then get more specific in the subdirectories.

For simple projects, you're fine just having one `.gitignore` in the project root directory.

8.4 Wildcards

Do I have to individually list all the files I don't want in the `.gitignore`? What a pain!

Luckily Git supports *wildcards* in ignored file naming.

For example, if we wanted to block all the files that ended with a `.tmp` or `.swp` (Vim's temp file name) extension, we could use the `*` ("splat") wildcard for that. Let's make a `.gitignore` that blocks those:

```
*.tmp
*.swp
```

And now any files ending with `.tmp` or `.swp` will be ignored.

Turns out that Vim has two kinds of swap files, `.swp` and `.swo`. So could we add them like this?

```
*.tmp
*.swo
*.swp
```

Sure! That works, but there's a shorter way where you can tell Git to match any character in a bracketed set. This is equivalent to the above:

```
*.tmp
*.sw[op]
```

You can read that last line as, "Match file names that begins with any sequence of characters, followed by `.sw`, followed by either `o` or `p`."

8.5 Negated `.gitignore` Rules

What if your root `.gitignore` is ignoring `*.tmp` files for the entire project. No problem.

But then later in development you have some deeply nested subdirectory that has a file `needed.tmp` that you really need to get into Git.

Bad news, though, since `*.tmp` is ignored at the root level across all subdirectories in the project! Can we fix it?

Yes! You can add a new `.gitignore` to the subdirectory with `needed.tmp` in it, with these contents:

```
!needed.tmp
```

This tells Git, "Hey, if you were ignoring `needed.tmp` because of some higher-up ignore rule, please stop ignoring it."

So while `needed.tmp` was being ignored because of the root level ignore file, this more-specific file overrides that.

If you needed to allow all `.tmp` files in this subdirectory, you could use wildcards:

```
!*.tmp
```

And that would make it so all `.tmp` files in this subdirectory were not ignored

8.6 How To Ignore All Files Except a Few?

You can use the negated rules for that.

Here's a `.gitignore` that ignores everything except files called `*.c` or `Makefile`:

```
*
!*.c
!Makefile
```

The first line ignores everything. The next two lines negate that rule for those specific files.

8.7 Getting Premade `.gitignore` Files

Here's a repo[1] with a whole bunch.

But you can also roll your own as needed. Use `git status` often to see if any files are there you want to ignore.

When you create a new repo on GitHub, it also gives you the option to choose a prepopulated `.gitignore`. **Warning!** Only do this if you're not planning to push an already-existing repo into this newly-made GitHub repo. If you plan to do this, GitHub's `,gitignore` will get in the way.

[1] https://github.com/github/gitignore

Chapter 9

Remotes

A *remote* is just a name for a remote server you can push and pull from.

We identify these by a URL; with GitHub, this is a URL we copied when we went to clone the repo initially.

It's possible to use this URL to identify the server in our Git usage, but it's unwieldy to type. So we give the remote server URLs nicknames that we just tend to call "remotes".

A remote we've already seen a bunch of is `origin`. This is the nickname for the remote repo you cloned from, and it gets set automatically by Git when you clone.

9.1 Remote and Branch Notation

Before we begin, note that Git uses slash notation to refer to a specific branch on a specific remote: `remotename/branchname`.

For example, this refers to the `main` branch on the remote named `origin`:

```
origin/main
```

And this refers to the branch named `feature3490` on a remote named `nitfol`:

```
nitfol/feature3490
```

We'll talk more about this in the Remote Tracking Branches chapter.

9.2 Getting a List of Remotes

You can run `git remote` with the `-v` option in any repo directory to see what remotes you have for that repo:

```
$ git remote -v
origin    https://github.com/beejjorgensen/git-example-repo.git (fetch)
origin    https://github.com/beejjorgensen/git-example-repo.git (push)
```

We see that we're using the same URL for the remote named `origin` for both push (part of which is `fetch`) and pull. Having the same URL for both is super common.

And that URL is the exact same one we copied from GitHub when cloning the repo in the first place.

9.3 Renaming a Remote

Remember that a remote name is just an alias for some URL that you cloned the repo from.

Let's say that you are all set up with your SSH keys to use GitHub for both push and pull, but you accidentally cloned the repo using the HTTPS URL. In that case, you'll see the following remote:

```
$ git remote -v
origin    https://github.com/beejjorgensen/git-example-repo.git (fetch)
origin    https://github.com/beejjorgensen/git-example-repo.git (push)
```

And then you try to push, and GitHub tells you that you can't push to an HTTPS remote… dang it!

You meant to copy the SSH URL when you cloned, which for me looks like:

```
git@github.com:beejjorgensen/git-example-repo.git
```

Luckily it's not the end of the world. We can just change what the alias points to.

(The example below is split into two lines so that it's not too wide for the screen, but it can be on a single line.)

```
$ git remote set-url origin \
                     git@github.com:beejjorgensen/git-example-repo.git
```

And now when we look at our remotes, we see:

```
$ git remote -v
origin    git@github.com:beejjorgensen/git-example-repo.git (fetch)
origin    git@github.com:beejjorgensen/git-example-repo.git (push)
```

And now we can push! (Assuming we have our SSH keys set up.)

9.4 Adding a Remote

There's nothing stopping you from adding another remote.

A common example is if you *forked* a GitHub Project (more on that later). A fork is a GitHub construct that enables you to easily clone someone else's public repo into your own account, and gives you a handy way to share changes you make with the original repo.

Let's say I forked the Linux source repo. When I clone my fork, I'll see these remotes:

```
origin    git@github.com:beejjorgensen/linux.git (fetch)
origin    git@github.com:beejjorgensen/linux.git (push)
```

I don't have access to the real Linux source code, but I can fork it and get my own copy of the repo.

Now, if Linus Torvalds makes changes to his repo, I won't automatically see them. So I'd like some way to get his changes and merge them in with my repo.

I need some way to refer to his repo, so I'm going to add a remote called `reallinux` that points to it:

```
$ git remote add reallinux https://github.com/torvalds/linux.git (fetch)
```

Now my remotes look like this:

```
origin    git@github.com:beejjorgensen/linux.git (fetch)
origin    git@github.com:beejjorgensen/linux.git (push)
```

```
reallinux    https://github.com/torvalds/linux.git (fetch)
reallinux    https://github.com/torvalds/linux.git (push)
```

> Normally when setting up a remote the refers to the source of a forked repo on GitHub, people tend to call that remote `upstream`, whereas I've clearly called it `reallinux`.
>
> I did this because when we subsequently talk about remote tracking branches, we're going to use "upstream" to mean something else, and I don't want the two to be confusing.
>
> Just remember IRL when you set up a remote to point to the forked-from repo, it's relatively customary to call that remote `upstream`.

Now I can run this to get all the changes from Linus's repo:

```
$ git fetch reallinux
```

And I can merge it into my branch (the Linux repo uses `master` for the `main` branch):

```
$ git switch master           # My local master
$ git merge reallinux/master   # Note the slash notation!
```

That will merge the `master` branch from the `reallinux` into my local master, once we've dealt with any conflicts.

At this point if I did a `git log`, I'd see that the latest commit would indicate that my `HEAD` was attached to my `master` branch, and it was pointing to the same commit as the `reallinux/master`:

```
(HEAD -> master, reallinux/master)
```

This is expected, since I just merged `reallinux/master` into my `master`, so they definitely should be pointing to the same commit.

But looking farther down, I'd see the `master` branch on my origin lagging behind a few commits:

```
(origin/master, origin/HEAD)
```

> You might or might not have `origin/HEAD` depending on how you made your repo.

At this point I'd do a `git push` to get them all on the same commit, so the top commit would show:

```
(HEAD -> master, reallinux/master, origin/master, origin/HEAD)
```

And now we're all happily pointing to the same commit.

It's interesting that my local `master` can be out of sync from the `master` on `origin`, right?

We'll look at this in the Remote Tracking Branches chapter.

Chapter 10

Remote Tracking Branches

We've seen how to create local branches that you do work on and then merge back into the `main` branch, then `git push` it up to a remote server.

This part of the guide is going to try to clarify what's actually going on behind the scenes, as well as give us a way to push our local branches to a remote for safe keeping.

10.1 Branches on Remotes

First, some refresher!

Recall that the remote repo you cloned yours from is a complete copy of your repo. The remote repo has a `main` branch, and therefore your clone also has a `main` branch.

That's right! When you make a GitHub repo and then clone it, there are **two** `main` branches!

How do we differentiate them?

Well, on your local clone, we just refer to branches by their plain name. When we say `main` or `topic2`, we mean the local branch by that name on our repo.

If we want to talk about a branch on a remote, we have to give the remote name along with the branch using that slash notation we've already seen:

```
main            # main branch on your local repo
origin/main     # main branch on the remote named origin
upstream/main   # main branch on the remote named upstream
zork/mailbox    # mailbox branch on the remote named zork
mailbox         # mailbox branch on your local repo
```

Importantly, not only do the words `origin/main` refer to the `main` branch on `origin` in casual conversation, but *you actually have a branch on your local repo called* `origin/main`.

This is called a *remote-tracking branch*. It's your local copy of the `main` branch on the remote. You can't move your local `origin/main` branch directly; Git does it for you as a matter of course when you interact with the remote (e.g. when you pull).

We're going to call the `main` branch on our local machines the *local branch*, and we'll call the one on `origin` the *upstream branch*.

10.2 Pushing to a Remote

Fun Fact: when you push or pull, you technically specify the remote and the branch you want to use. This is me saying, "Push the branch I'm on right now (presumably `main`) and merge it into `main` on `origin`.

```
$ git push origin main
```

"But wait! I haven't been doing that!"

It turns out there's an option you can set to make it happen automatically. Let's say you're on the `main` branch and then run this:

```
$ git push --set-upstream origin main
$ git push -u origin main              # same thing, shorthand
```

This will do a couple things: 1) it'll push changes on your local `main` to the remote server, and 2) it'll remember that the remote branch `origin/main` is tracking your local `main` branch.

And then, from then on, from the `main` branch, you can just:

```
$ git push
```

and it'll automatically push to `origin/main` thanks to your earlier usage of `--set-upstream`.

And `git pull` has the same option, as well, though you only need to do it once with either push or pull.

"But wait! I've never used `--set-upstream`, either!"

That's because by default when you clone a repo, Git automagically sets up a local branch to track the `main` branch on the remote.

> **Bonus Info**: Depending on how you made your repo, you might also have a reference to `origin/HEAD`. It might be weird to think that there's a `HEAD` ref on a remote server that you can see, but in this case it's just referring to the branch that you'll be checking out by default when you clone the repo.

"OK, so what you're telling me is that I can just `git push` and `git pull` like always and just ignore everything you wrote in this section?"

Well… yes. Ish. No. We're going to make use of this to push other branches to the remote!

10.3 Making a Branch and Pushing to Remote

I'm going to make a new local branch `topic99`:

```
$ git switch -c topic99
Switched to a new branch 'topic99'
```

And make some changes:

```
$ vim README.md        # Create and edit a README
$ git add README.md
$ git commit -m "Some important additions"
```

In our log, we can see where all the branches are:

```
commit 79ddba75b144bad89e1cbd862e5f3b3409f6c498 (HEAD -> topic99)
Author: User Name <user@example.com>
Date:   Fri Feb 16 16:44:50 2024 -0800

    Some important additions
```

```
commit 3be2ad2c31b627b431af8c8e592c01f4b989d621 (origin/main, main)
Author: User Name <user@example.com>
Date:   Fri Feb 16 16:14:13 2024 -0800

    Initial checkin
```

`HEAD` refers to `topic99`, and that's one commit ahead of `main` (local) and `main` (upstream on the `origin` remote), as far as we know. And we know this because it's one commit ahead of our remote-tracking branch `origin/main`.

Now let's push!

```
$ git push
fatal: The current branch topic99 has no upstream branch.
To push the current branch and set the remote as upstream, use

    git push --set-upstream origin topic99

To have this happen automatically for branches without a tracking
upstream, see 'push.autoSetupRemote' in 'git help config'.
```

Ouch. The short of all this is that we said "push", and Git said, "To what? You haven't associated this branch with anything on the remote!"

And we haven't. There's no `origin/topic99` remote-tracking branch, and certainly no `topic99` branch on that remote. Yet.

The fix is easy enough—Git already told us what to do.

```
$ git push --set-upstream origin topic99
```

And that will do it.

At this point, assuming you've pushed to GitHub, you could go to your GitHub page for the project, and near the top left you should see something that looks like Figure 10.1.

Figure 10.1: Two branches on GitHub

If you pull down that `main` button, you'll see `topic99` there as well. You can select either branch and view it in the GitHub interface.

Chapter 11

File States

We've talked about this quite a bit in passing already.

If you create a new file, you have to `git add` it to the stage before you commit.

If you modify a file, you have to `git add` it to the stage before you commit.

If you add a file `foo.txt` to the stage, you can remove it from the stage before you commit with `git restore --staged foo.txt`.

So clearly files can exist in a variety of "states" and we can move them around between those states.

To figure out what state a file is in and get a hint on how to "undo" it from that state, `git status` is your best friend (except in the case of renaming, but more on that mess soon).

11.1 What States Can Files in Git Be In?

There are four of them: **Untracked**, **Unmodified**, **Modified**, and **Staged**.

- **Untracked**: Git does not know anything about this file (e.g. you just created it in the repo). Git will ignore it, but you'll see it in the status.

  You can make Git aware of this file by moving it to Staged State with `git add`.

  Or you can simply remove the file if you don't want it to exist, or you can add it to your `.gitignore` if you want to leave it in place but still have Git ignore it.

- **Unmodified**: Git knows about this file and it's in the repo. But you haven't made any changes to it since it was last committed.

  You can move this file to Modified State by making changes to the file (and saving).

  You can remove this file with `git rm`, which changes the removed file to the Staged State. (Wait—removing the file puts it on the stage? Yes! More on that later.)

- **Modified**: Git knows about this file and knows that you've changed it. It's ready for you to stage those changes or to undo them.

  You can change the file to Staged State with `git add`.

  You can change the file to Unmodified State (throwing away your changes) with `git restore`.

- **Staged**: The file is ready to be included in the next commit.

  You can change to Unmodified State by making a commit with `git commit`.

  You can remove the file from the stage and back to Modified State with `git restore --staged`.

A file typically goes through this process to be added to a repo:

1. The user creates a new file and saves it. This file is **Untracked**.

2. The user adds the file with `git add`. The file is now **Staged**.
3. The user commits the file with `git commit`. The file is now **Unmodified** and is part of the repo and ready to go.

After it's in the repo, the typical file life cycle only differs by the first step:

1. The user changes the file and saves it. The file is now **Modified**.
2. The user adds the file with `git add`. The file is now **Staged**.
3. The user commits the file with `git commit`. The file is now **Unmodified** and is part of the repo and ready to go.

Keep in mind that often a commit is a bundle of different changes to different files. All those files would be added to the stage before the single commit.

Here's a partial list of ways to change state:

- **Untracked** → `git add foo.txt` → **Staged** (as "new file")
- **Modified** → `git add foo.txt` → **Staged**
- **Modified** → `git restore foo.txt` → **Unmodified**
- **Unmodified** → `edit foo.txt` → **Modified** (with your favorite editor)
- **Staged** → `git commit` → **Unmodified**
- **Staged** → `git restore --staged` → **Modified**

Again, `git status` will often give you advice of how to undo a state change.

11.2 Renaming Files

You can use the OS rename command to rename files, but if they're in a Git repo, it's better to `git mv` them so that Git has total awareness.

Git is confusingly unhelpful with this.

Let's rename `foo.txt` to `bar.txt` and get a status:

```
$ git mv foo.txt bar.txt
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    renamed:    foo.txt -> bar.txt
```

So it knows the file is renamed, and the file has been moved to the stage. Like so:

- **Unmodified** → `git mv foo.txt bar.txt` → **Staged** (as "renamed")

And if we look, we see the file has actually been renamed in the directory to `bar.txt`, as well.

If we make a commit at this point, the file will be renamed in the repo. Done.

But what if we want to undo the rename?

Git suggests `git restore --staged` to the rescue… But which file name to use, the old one or new one? And then what? It turns out that while you *can* use `git restore` to undo this by following it with multiple other commands, you should, in this case, ignore Git's advice.

Just remember this part: **the easiest way to undo a Staged rename is to just do the reverse rename**.

Let's say we renamed and got here:

```
$ git mv foo.txt bar.txt    # Rename foo.txt to bar.txt
$ git status

On branch main
```

```
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    renamed:    foo.txt -> bar.txt
```

This easiest way to revert this change is to do this:

```
$ git mv bar.txt foo.txt    # Rename it back to foo.txt
$ git status

On branch main
nothing to commit, working tree clean
```

And there you go.

In summary, the way to rename a file is:

- **Unmodified** → `git mv foo.txt bar.txt` → **Staged**
- **Staged** → `git commit` → **Unmodified**

And the way to back out of a Staged rename is to rename them back the way they were:

- **Staged** → `git mv bar.txt foo.txt` → **Unmodified**

11.3 Removing Files

You can use the OS remove command to remove files, but if they're in a git repo, it's better to `git rm` them so that Git has total awareness.

And what happens might seem a little strange.

Let's say we have a file `foo.txt` that has already been committed. But we decide to remove it.

```
$ git rm foo.txt
rm 'foo.txt'         # This is Git's output
```

This actually removes the file—if you look in the directory, it's gone.

But let's check the status:

```
$ git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    deleted:    foo.txt
```

So the now-deleted file is in Staged State, as it were.

If we do a commit here, the file is deleted. Done.

But what if we want to undo the staging of the now-deleted file? There's a hint for how to get it back with `git restore --staged`, as per usual.

Let's try it:

```
$ git restore --staged foo.txt
$ git status
On branch main
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    deleted:    foo.txt
```

```
no changes added to commit (use "git add" and/or "git commit -a")
```

Hmmm. “Changes not staged for commit” are files in Modified State. This means that `foo.txt` has been “modified”, which is, in this context, a friendlier way of saying “deleted”.

So we’ve backed up from Staged State to Modified State. But looking around, the file is still gone! I want my file back!

We want to move it back to Unmodified State, which Git once again hints how to do in the status: `git restore`. Let’s try:

```
$ git restore foo.txt
$ git status
On branch main
nothing to commit, working tree clean
```

Git’s telling us there are no Modified files here. Let’s look and see:

```
$ ls foo.txt
foo.txt
```

There it is, back safe and sound.

So the process for deleting a committed file is a variant of what we’ve already seen:

- **Unmodified** → `git rm foo.txt` → **Staged**
- **Staged** → `git commit` → The file is now gone

And you can undo a deleted file (as long as the delete hasn’t yet been committed) in the same way you can undo any other file states:

- **Staged** → `git restore --staged foo.txt` → **Modified**
- **Modified** → `git restore foo.txt` → **Unmodified**

Later we’ll talk about ways to recover a deleted file from an earlier commit. But one way you already know: check out the earlier commit where the file exists, copy the file into a new Untracked file, checkout the branch where the file will be restored to, rename the Untracked file to the name of the restored file, then add it and commit.

> **Note**: Just because you remove a file and push your changes doesn’t mean the file is permanently gone. It’s still in the repo, part of whatever commits it was previously seen with.
>
> If you accidentally commit something that should be secret, you should consider that secret compromised and change it where it is used. It will be visible to anyone who clones the repo and sees that commit.
>
> There are ways around this if you haven’t yet pushed, but that’s beyond the scope of this guide.

11.4 Unmodified to Untracked

A variation of `git rm` tells Git to remove the file from the repo but leave it intact in the working tree. Maybe you want to keep the file around but don’t want Git to track it any longer.

To make this happen, you use the `--cached` switch.

Here’s an example where we remove the file `foo.txt` from the repo but keep it around in our working tree:

```
$ ls

foo.txt

$ git rm --cached foo.txt

rm 'foo.txt'

$ git status

On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    deleted:    foo.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    foo.txt

$ ls
foo.txt
```

There you see in the `status` output that Git has staged the file for deletion, but it's also mentioning that the file exists and is untracked. And a subsequent `ls` shows that the file still exists.

At this point, you can commit and the file would then be in Untracked state.

Chapter 12

Collaboration across Branches

Let's say you're on a team of coders and you all have access to the same GitHub repo. (One person on the team owns the repo, and they've added you all as collaborators[1].)

> I'm going to use the term *collaborator* to mean "someone to whom you have granted write access to your repo".

How can you all structure your work so that you're minimizing conflicts?

There are a number of ways to do this.

- Everyone is a collaborator on the repo, and:
  - Everyone uses the same branch, probably `main`, or:
  - Everyone uses their own remote tracking branch and periodically merges with the main branch, or:
  - Everyone uses their own remote tracking branch and periodically merges with a development branch, which itself is periodically merged into `main` for each official release.
- Or everyone has their own repo (and are not collaborators on the same repo), and:
  - Everyone uses *pull requests* or other synchronization methods to get their repos merged into the other devs'.

We'll look at the first few ways in this chapter, but we'll save pull requests for later.

There's no one-size-fits-all approach to teamwork with Git, and the methods outlined below can be mixed and matched with local topic branches, or people having multiple remote tracking branches, or whatever. Often management will have an approach they want to use for collaboration which might be one of the ones in this section, or maybe it's a variant, or maybe its something completely different.

In any case, the best strategy for you, the learner, is to just be familiar with the tools (branching, merging, conflict resolution, pushing, pulling, remote tracking branches) and use them for effect where it makes the most sense.

And when you're first starting out, your intuition about "where it makes the most sense" might not be dead-on, but it probably won't be lethal and you'll figure it out in the school of hard knocks.

> "Oh great. Another f—ing learning experience."
>
> —Actual quote from my mother

[1] https://docs.github.com/en/enterprise-server@3.9/account-and-profile/setting-up-and-managing-your-personal-account-on-github/managing-access-to-your-personal-repositories/inviting-collaborators-to-a-personal-repository

12.1 Communication and Delegation

Git can't save you from poor communication. The only way to minimize conflicts in a shared project to communicate with your team and clearly assign different tasks to people in a non-conflicting way.

Two people shouldn't generally be editing the same part of the same file, or even any part of the same file. That's more of a guideline than a rule, but if you follow it, you will never have a merge conflict.

As we've seen, it's not the end of the world if there is a merge conflict, but life sure is easier if they're just avoided.

Takeaway: without good communication and a good distribution of work on your team, you're doomed. Make a plan where no one is stepping on toes, and stick to it.

12.2 Approach: Everyone Uses One Branch

This is really easy. Everyone has push access to the repo and does all their work on the `main` branch.

Benefits:

- Super simple to set up.
- Conceptually not much to juggle.
- All work instantly available to all collaborators upon push.

Drawbacks:

- More potential for merge conflicts.
- Unless you're rebasing (more on that later), you'll have a lot of merge commits.
- You can't push non-working code since it will break everything for everyone else.

Initial setup:

- One person makes the GitHub repo
- The owner of the GitHub repo adds all the team members as collaborators.
- Everyone clones the repo.

Workflow:

- Work is delegated to all collaborators. The work should be as non-overlapping as possible.
- Everyone periodically pulls `main` and resolves any merge conflicts.
- Everyone pushes their work to `main`.

In real life, this approach is probably only used on very small teams, e.g. three people at most, with frequent and easy communication between all members. If you're working on a small team in school, it could very well be enough, but I'd still recommend trying a different approach just for the experience.

The other approaches are not that much more complex, and give you a lot more flexibility.

12.3 Approach: Everyone Uses Their Own Branch

In this scenario, we treat `main` as the working code, and we treat contributors' branches as where work is done. When a contributor gets their code working, they merge it back into `main`.

Benefits:

- You get to work on your own branch without worrying about messing up other people's work.
- You can commit non-working code since no one else can see it. (You might be wrapping up the work day and want to push some incomplete code for a backup, for example.)
- Less merge conflict potential since fewer merges are happening than if everyone were committing to `main`.

Drawbacks:

- If your branch diverges too far from `main`, merging might become painful.

- Unless you're rebasing, the incremental work on your branch might "pollute" the commit history on `main` with a lot of tiny commits.

Initial setup:

- One person makes the GitHub repo
- The owner of the GitHub repo adds all the team members as collaborators.
- Everyone clones the repo.
- Everyone makes their own branch, possibly naming it after themselves.
- Everyone pushes their branch to GitHub, making them remote-tracking branches. (We do this so that your work is effectively backed up on GitHub when you push it.)

Workflow:

- Work is delegated to all collaborators. The work should be as non-overlapping as possible.
- As collaborators finish their tasks, they will:
  - Test everything on their branch.
  - Merge the latest `main` into their branch; do a pull to make sure you have it. (The collaborator might already have the latest `main` if no one else has merged into it, which will cause Git to say there's nothing to do. This is fine.)
  - Test everything, and fix it if necessary.
  - Merge their functioning branch into `main`.
  - Push.
    - If someone else has modified `main` while you were testing, Git will complain that you have to pull before you can push. If there's a conflict at this point, you'll have to resolve, test, and push it. And you'll have to merge `main` back into your branch so that your branch is up-to-date.

The result will look something like Figure 12.1 to start, where all the collaborators have made their own branches off of `main`.

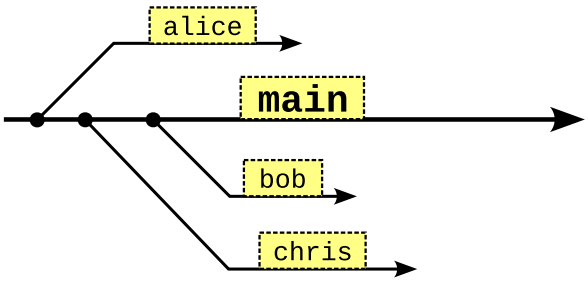

Figure 12.1: Collaborators branching off `main`.

Let's say Chris (on branch `chris`) finishes up their work and wants other contributors to be able to see it. It's time to merge into `main`, as we graphically see in Figure 12.2.

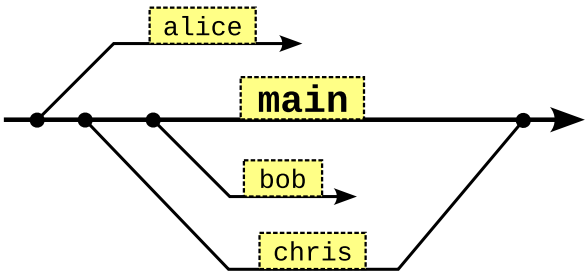

Figure 12.2: Chris merges back into `main`.

After that, other contributors looking at `main` will see the changes.

12.4 Approach: Everyone Merges to the Dev Branch

In this scenario, we treat `main` as the published code that we're going to distribute, often tagged with a release version number, and we treat a `dev` branch as the working, unreleased code. And, as in the previous scenario, everyone has their own branches they're developing on.

The idea is basically we're going to have two versions of the working code:

1. The public, released version that's on `main`.
2. The private, internal version that's on `dev`.

And then, of course, we'll have one branch per collaborator.

Another way of thinking about it is that we're going to have our internal build on `dev` that is good for testing and then, when it's all ready, we'll "bless" it and merge it into `main`.

So there will be a lot of merges into `dev` from all the developer branches, and then every so often there will be a merge from `dev` into `main`.

The developers will never directly merge into `main`! Usually that is performed by someone in a managerial role.

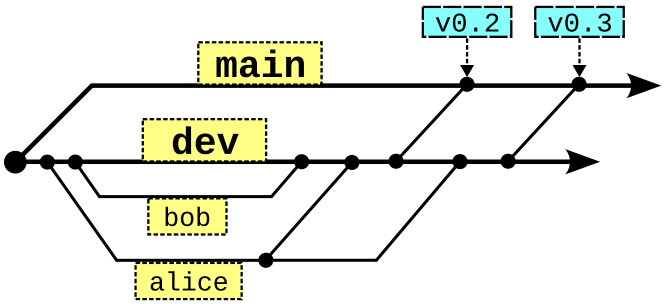

Figure 12.3: Working on the `dev` branch.

Overall the process works as in Figure 12.3. This is a busy image, but notice how Bob and Alice are only merging their work into the `dev` branch, and then every so often, their manager merges the `dev` branch into `main` and tags that commit with a release number. (More on tagging later.)

Benefits:

- All the benefits of everyone having their own branch.
- You have an internal branch from which you can make complete builds for internal or external testing.

Drawbacks:

- A little more complexity and management.
- If your branch diverges too far from `dev`, merging might become painful.
- Unless you're rebasing, the incremental work on your branch might "pollute" the commit history on `dev` with a lot of tiny commits.

Initial setup:

- One person makes the GitHub repo
- The owner of the GitHub repo adds all the team members as collaborators.
- Everyone clones the repo.
- Everyone makes their own branch, possibly naming it after themselves.
- Everyone pushes their branch to GitHub, making them remote-tracking branches. (We do this so that your work is effectively backed up on GitHub when you push it.)

Workflow:

- Work is delegated to all collaborators. The work should be as non-overlapping as possible.
- As collaborators finish their tasks, they will:
  - Test everything on their branch.

- Merge the latest `dev` into their branch; do a pull to make sure you have it. (The collaborator might already have the latest `dev` if no one else has merged into it, which will cause Git to say there's nothing to do. This is fine.)
- Test everything, and fix it if necessary.
- Merge their functioning branch into `dev`.
- Push.
    - If someone else has modified `dev` while you were testing, Git will complain that you have to pull before you can push. If there's a conflict at this point, you'll have to resolve, test, and push it. And you'll have to merge `dev` back into your branch so that your branch is up-to-date.

Managerial Workflow:

- Coordinate with all devs to get a candidate release in `dev` tested out and ready.
- Merge that candidate release (some commit) from `dev` into `main`.
- Tag the `main` commit with some version number, optionally.

Chapter 13

Appendix: Making a Playground

In programming circles in general, a *playground* is a place you can go to mess with code and tech and not worry about messing up your production system.

And there are places you can go online to find these, but with Git, I find it's just as easy to make your own local repo.

Here's a way to make a new local repo called `playground` out of the current directory. (You should **not** be under a Git repo at this time; create the playground outside other existing repos.)

```
$ git init playground
  Initialized empty Git repository in /user/playground/.git/
```

`playground` isn't a special name. You can call it `foo` or anything. I'll just use it for this example.

What that command did was create a new subdirectory called `playground` and create a Git repo in it.

Let's continue at the end: how do you delete the repo? You just remove the directory.

```
$ rm -rf playground   # delete the playground repo
```

And let's create it again:

```
$ git init playground
```

We have all the power!

> **Note:** This repo only exists on this computer; it has no remotes and no way to push. You could add that stuff later, if you wanted, but playgrounds tend to be temporary areas where you're just trying things out.

Let's go into the playground and check it out.

```
$ cd playground
$ ls -la
  total 4
  drwxr-xr-x  3 user group   18 Jul 13 14:43 .
  drwxr-xr-x 22 user group 4096 Jul 13 14:43 ..
  drwxr-xr-x  7 user group  119 Jul 13 14:43 .git
```

There's a directory there called `.git` that has all the metadata in it.

> **Note:** If we wanted to change this directory from a Git repo to just a normal directory, we could run this:
>
> ```
> $ rm -rf .git       # Delete the .git directory
> ```
>
> Again, we have all the power! But let's show some restraint and not do that yet.

What can we do?

What *can't* we do? Let's make a file and see where we stand:

```
$ echo "Hello, world" > hello.txt   # Create a file

$ ls -l

  total 4
  -rw-r--r-- 1 user group 13 Jul 13 14:47 hello.txt

$ git status

  On branch main

  No commits yet

  Untracked files:
    (use "git add <file>..." to include in what will be committed)
      hello.txt

  nothing added to commit but untracked files present (use "git add"
  to track)
```

Now we have an untracked file.

We can `git add` it, we can `git commit` it, we can create branches, we can merge them and make conflicts and resolve them and `git rebase` and `git reset` and all kinds of stuff.

We don't have a remote, so the only things we can't do involving pushing and pulling.

But it turns out we can even make that happen! Let's see how.

13.1 Cloning Bare Repos

A *bare repo* is one without a working tree. You can't go in there to see files, because they don't exist in there in a normal sense. The only thing that's there is metadata and the commit snapshots.

You can clone, push, and pull bare repos.

Let's make one (again, you could name it anything you want), noting the `--bare` command line option:

```
$ git init --bare origin_repo
  Initialized empty Git repository in /user/origin_repo/
```

If you look in there (to be clear, you have no reason to) you'll just see metadata and directories.

Before we can use it, we'd better clone it. For ease, we'll do this from the same directory we created it.

```
$ git clone origin_repo playground
  Cloning into 'playground'...
```

```
  warning: You appear to have cloned an empty repository.
  done.
```

It is empty, naturally. We haven't made any commits.

Now we have two repos in this directory:

- `origin_repo`: the bare repo we cloned, and:
- `playground`: the repo we cloned from it.

Let's jump in there and see what's up:

```
$ cd playground
$ git remote -v
  origin    /user/origin_repo (fetch)
  origin    /user/origin_repo (push)
```

We have remotes! Of course we do. We cloned this repo, and Git automatically sets up the `origin` remote.

And remember that `origin` is just an alias for some remote that's identified somehow. We're used to seeing remotes that start with `https` or `ssh`, but here's an example of a remote that's just another subdirectory on your disk.

Let's make a file and commit it, and see if we can push!

```
$ echo "Hello, world" > hello.txt
$ git add hello.txt

$ git commit -m added
  [main (root-commit) 4a82a14] added
   1 file changed, 1 insertion(+)
   create mode 100644 hello.txt

$ git push
  Enumerating objects: 3, done.
  Counting objects: 100% (3/3), done.
  Writing objects: 100% (3/3), 907 bytes | 907.00 KiB/s, done.
  Total 3 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
  To /user/origin_repo
   * [new branch]      main -> main

$ git branch -va
  * main                4a82a14 added
    remotes/origin/main 4a82a14 added
```

And we've successfully pushed our file up to `origin`.

Finally, let's make another clone. First we'll `cd` back down to where `origin_repo` is and clone again this time into `playground2`:

```
$ git clone origin_repo playground2
  Cloning into 'playground2'...
  done.
```

Let's `cd` in there and see what we have. It's a clone of the repo, so we'd better see the `hello.txt` we pushed in there from `playground` earlier.

```
$ cd playground2

```

```
$ ls
  hello.txt

$ cat hello.txt
  Hello, world
```

Voila! It's there!

Since `playground` and `playground2` are both clones of the same repo, you can push from one and pull from the other to get the changes.

You can even make conflicting changes and try to `git pull` or `git pull --rebase` and see how things go wrong and how to fix them.

And if everything goes complete off the rails, you can just delete the directories and start again. It's a playground!

13.2 Automating Playground Builds

It can be tedious to continually destroy and recreate repos that you're trying to learn from. I suggest putting your commands in a *shell script*, which is just a text file that contains the commands to run.

Let's say you make a new text file called `buildrepo.sh` and you put the following text in it:

```
rm -rf playground    # Remove old playground
git init playground  # Create a new one
cd playground
echo "Hello, world!" > hello.txt   # Create hello.txt
echo "foobar" > foobar.txt         # Create foobar.txt
git add hello.txt foobar.txt
git commit -m added
echo "foobar again" >> foobar.txt  # Append text
git add foobar.txt
git commit -m updated
```

That's just a bunch of shell commands. But here's the fun bit: if you run `sh` (the shell) with `buildrepo.sh` as an argument, it will run all those commands in order!

```
$ sh buildrepo.sh
  Initialized empty Git repository in /user/playground/.git/
  [main (root-commit) 2239237] added
   2 files changed, 2 insertions(+)
   create mode 100644 foobar.txt
   create mode 100644 hello.txt
  [main 0533186] updated
   1 file changed, 1 insertion(+)
```

> **Protip:** If you run `sh -x buildrepo.sh` it will also show you the commands it is running.

After that, we can `cd` in there and see what happened:

```
$ cd playground
$ git log
  commit 05331869d77973dfbac38a31c40a44f99225e85d
  Author: User Name <user@example.com>
  Date:   Sat Jul 13 15:19:42 2024 -0700
```

```
        updated

    commit 2239237cc44d11e9479dcc610e5d02ad283766ce
    Author: User Name <user@example.com>
    Date:   Sat Jul 13 15:19:41 2024 -0700

        added

  $ cat foobar.txt
    foobar
    foobar again
```

By putting the initialization commands in a shell script, it's almost like having a "saved game" at that point. You can just rerun the shell script any time you want the same playground set up.

Chapter 14

Appendix: Getting Out of Editors

If you try to `git commit` and don't specify `-m` for a message, or if you `git pull` and there's a non-fast-forward merge, or if you `git merge` and there's a non-fast-forward merge and you don't specify `-m`, or what I'm sure are a host of other reasons, you might get popped into an editor.

And you might not be familiar with that editor.

So here's how to get out of it.

- **Nano**: If the editor says "Nano" or "Pico" in the upper left, then edit the commit message (if you want), then then hit `CTRL-X`, and then hit `Y` to save, then `ENTER` to accept the given filename.
- **Vim**: If the screen has a bunch of `~` characters down the left and a crazy-looking file name at the bottom maybe with the word `All`, you're in Vim or some other vi ("vee eye") variant. Press `i`, then type a message (if you want), then hit the `ESC` key in the upper left, then type two capital `Z`s in a row. `ZZ`. That should save and exit. Learning Vim[1] is beyond the scope of this guide, but this author thinks it's worth it for the editing speed you can achieve.

[1] https://www.openvim.com/

Chapter 15

Appendix: Errors and Scary Messages

15.1 Detached Head

Did you get this alarmingly guillotinesque message?

```
You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by switching back to a branch.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -c with the switch command. Example:

  git switch -c <new-branch-name>

Or undo this operation with:

  git switch -

Turn off this advice by setting config variable advice.detachedHead to false

HEAD is now at 0da5af9 line 1
```

This means that you've checked out a commit directly instead of checking out a branch. That is, your `HEAD` is no longer attached to a branch, i.e. it is "detached".

To get out of this, you can:

1. Undo the checkout that got you detached:

   ```
   git switch -
   ```

2. Switch to another branch entirely:

   ```
   git switch main
   ```

3. Make a new branch here and check it out:

   ```
   git switch -c newbranch
   ```

And now your `HEAD` is no longer detached.

15.2 Upstream Branch Name Doesn't Match Current

Did you accidentally run `git branch -c newbranch` when you meant to run `git switch -c newbranch`? Because if you did, it could land you here:

```
fatal: The upstream branch of your current branch does not match
the name of your current branch.  To push to the upstream branch
on the remote, use

    git push origin HEAD:main

To push to the branch of the same name on the remote, use

    git push origin HEAD

To choose either option permanently, see push.default in 'git help config'.

To avoid automatically configuring an upstream branch when its name
won't match the local branch, see option 'simple' of branch.autoSetupMerge
in 'git help config'.
```

Let's check our branch names to see what's going on:

```
$ git branch -vv
  main      fc645f2 [origin/main] line 2
* newbranch 7c21054 [origin/main: behind 1] line 1
```

That tells us our local branch names and, in brackets, the corresponding remote-tracking branch. Notice anything fishy?

It seems `main` corresponds with `origin/main`.

And that `newbranch` **also** corresponds with `origin/main`! How?!

Well, when you did `git branch -c newbranch`, that *copies* the current branch (`main` in this example) into the other branch, *including its remote-tracking branch.* Bad news, since you really want `newbranch` to correlate to `origin/newbranch`, if anything.

You have a few options.

1. You want to push `newbranch` up to the `origin` and track it as `origin/newbranch`.

   Just do this to push and change the remote-tracking branch name:

   ```
   $ git push -u origin newbranch
   ```

2. You just want this to be a local branch and don't need it on the remote.

   In this case, just unset the upstream:

   ```
   $ git branch --unset-upstream newbranch
   ```

15.3 Current Branch Has No Upstream Branch

Trying to push and getting this message?

```
fatal: The current branch topic1 has no upstream branch.
To push the current branch and set the remote as upstream, use
```

```
    git push --set-upstream origin topic1

To have this happen automatically for branches without a tracking
upstream, see 'push.autoSetupRemote' in 'git help config'
```

This just means there's no upstream tracking branch for `topic1`—it's just a local branch.

If you do want to push this branch, just follow the suggested instruction.

If you are pushing from the wrong branch by accident, switch to the right one first.

Index