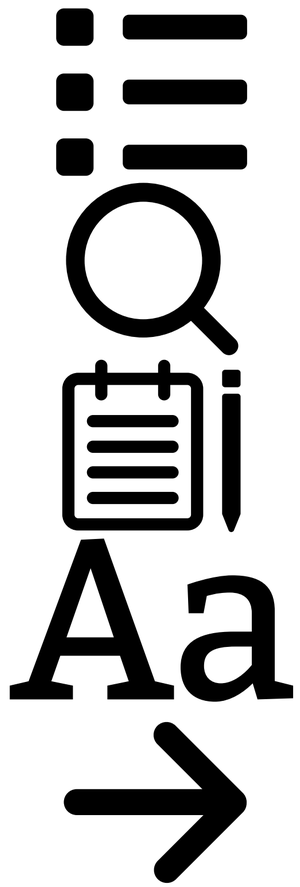

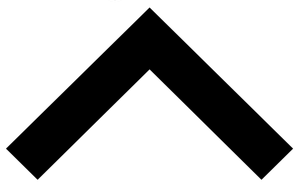

7

Branching Written by Bhagat Singh & Chris Belanger

One of the driving factors behind Git's original design was to support the messy, non-linear approach to development that stems from working on large-scale, fast-moving projects. The need to split off development from the main development line, make changes independently and in isolation of other changes on the main development line, easily merge those changes back in, and do all this in a lightweight manner, was what drove the creators of Git to build a very lightweight, elegant model to support this kind of workflow.

In this chapter, you'll explore the first half of this paradigm: **branching**. You've touched on branching quite briefly in Chapter 1, "A Crash Course in Git," but you probably didn't quite understand what you, or Git, were *doing* in that moment.

Although you can hobble through your development career never really understanding how branching in Git actually works, branching is *incredibly* important to the development workflows of many development teams, both large and small, so knowing what's going on under the hood, and having a solid mental model of your repository's branching structure will help you immensely as your projects grow in size and complexity.

What is a commit?

That question was asked and answered in a shallow manner a few chapters ago, but it's a good time to revisit that question and explore commits in more detail.

Recall that a commit represents the state of your project tree — your directory — at a particular point in time:

```
├── LICENSE
├── README.md
├── articles
│   ├── clickbait_ideas.md
│   ├── ios_article_ideas.md
│   └── live_streaming_ideas.md
├── books
│   └── book_ideas.md
└── videos
    ├── content_ideas.md
    └── platform_ideas.md
```

You probably think about your files primarily in terms of their content, their position inside the directory hierarchy, and their names. So when you think of a commit, you're likely to think about the state of the files, their content and names at a particular point in time. And that's correct, to a point: Git also adds some more information to that "state of your files" concept in the form of metadata.
Git metadata includes such things like "when was this committed?" and "who committed this?", but most importantly, it includes the concept of "where did this commit originate from?" — and that piece of information is known as the commit's **parent**. A commit can have one or two parents, depending on how it was branched and merged back in, but you'll get to that point later.
Git takes all that metadata, including a reference to this commit's parent, and wraps that up with the state of your files as the commit. Git then **hashes** that collection of things using **SHA1** to create an ID, or **key**, that is unique to that commit inside your repository. This makes it extremely easy to refer to a commit by its hash value, or as you saw in the previous chapter, its short hash.

What is a branch?

The concept of a branch is massively simple in Git: It's simply a reference, or a label, to a commit in your repository. That's it. Really. And because you can refer to a commit in Git simply through its hash, you can see how creating branches is a terribly cheap operation. There's no copying, no extra cloning, just Git saying "OK, your new branch is a label to commit `477e542`". Boom, done.
As you make commits on your branch, that label for the branch gets moved forward and updated with the hash of each new commit. Again, all Git does is update that label, which is stored as a simple file in that hidden **.git** repository, as a really cheap operation.
You've been working on a branch all along — did you realize that? Yes, `main` or whatever you've chosen as the originating branch of your repository, is nothing but a regular branch. It's only by convention, and the default name that Git applies to this default branch when it creates a new repository, that we say "Oh, the `main` branch is the *original* branch."
Note: As of version 2.28.0, Git now provides a setting by which you can control the label to be used when you create the first branch in a new repository. This defaults to `master`, but you can choose to set this to `main` or whatever you like.
The setting is `init.defaultBranch`, and you can change it with the following command: `$ git config --global init.defaultBranch main`
This sets the default branch name to `main`.
This only affects new repositories that you create; it doesn't change the default branch name of any existing repositories.
There's nothing special about `main`; again, Git simply knows that the `main` branch is a revision in your repository pointed to by a simple label held in a file on disk. Sorry to dash any notion that `main` was magic or something.

Creating a branch

You created a branch before in the crash-course chapter, but now you're going to create a branch and watch exactly what Git is doing.
The command to create a branch in Git is, unsurprisingly, `git branch`, followed by the name of your branch.
Execute the following command to create a new branch:

```
git branch testBranch
```

Git finishes that action with little fanfare, since a new branch is not a big deal to Git.

How Git tracks branches

To see that Git actually *did* something, execute the following command to see what Git's done in the background:

```
ls .git/refs/heads/
```

This directory contains the files that point to all of your branches. I get the following result of two files in that directory:

```
main            testBranch
```

Oh, that's interesting — a file named **testBranch**, the same as your branch name. Take a look at **testBranch** to see what's inside, using the following command:

```
cat .git/refs/heads/testBranch
```

Wow — Git *is* really bare-bones about branches. All that's in there is a single hash value. To take this to a new level of pedantry, you can prove that the label **testBranch** is pointing to the actual latest commit on your repository.
Execute the following to see the latest commit:

```
git log -1
```

You'll see something like the following (your hash will be different than mine):

```
commit 477e542bfa35942ddf069d85fbe3fb0923cfab47 (HEAD -> main, testBranch)
Author: Chris Belanger <chris@razeware.com>
Date:   Wed Jan 23 16:49:56 2019 -0400

    Adding .gitignore files and HTML
```

Let's pick this apart a little. The commit referenced here is, indeed, the same hash as contained in **testBranch**. The next little bit, `(HEAD -> main, testBranch)`, means that this commit is pointed to by *both* the `main` and the `testBranch` branches. The reason this commit is pointed to by both labels is because you've only created a new branch, and not created any more commits on this branch. So the label can't move forward until you make another commit.

Checking your current branch

Git can easily tell you which branch you're on, if you ever need to know. Execute the following command to verify you're working on **testbranch**:

```
git branch
```

Without any arguments or options, `git branch` simply shows you the list of local branches on your repository. You should have the two following branches listed:

```
* main
  testBranch
```

The asterisk indicates that you're still on the **main** branch, even though you've just created a new branch. That's because Git won't switch to a newly created branch unless you tell it explicitly.

Switching to another branch

To switch to **testBranch**, execute the `checkout` command like so:

```
git checkout testBranch
```

Git responds with the following:

```
Switched to branch 'testBranch'
```

That's really all there is to creating and switching between branches.
Note: Admittedly, the term `checkout` is a bit of a misnomer, since if you've ever owned a library card, you know that checking out a book makes that book inaccessible to anyone else until you return it.
That term is a holdover from the way that some older version control systems functioned, as they used a lock-modify-unlock model, which prevented anyone else from modifying the file at the same time. It worked really well for preventing merge conflicts, but pretty much killed any form of distributed, concurrent development.

Speaking of old version control systems, if any of you used PVCS Version Manager back in the day (c. 2000 or so), drop me a line and we can swap horror stories about the amazingly sparse documentation, the endless fighting with semaphores, and all the other fun bits that came along with that piece of software.
That's enough poking around with **testBranch**, so switch back to **main** with the following command:

```
git checkout main
```

You really don't need **testBranch** anymore, since there are other, real branches to be explored. Delete **testBranch** with the following command:

```
git branch -d testBranch
```

Time to take a look at some real branches. You already have one in your repository, just waiting for you to go in and start doing some work… what's that? Oh, you don't remember seeing that branch when you last executed `git branch`? That's because `git branch` by itself only shows the local branches in your repository.
When you first cloned this repository (which was a fork from the original **ideas** repository), Git started tracking both the local repository, as well as the **remote** repository — i.e., the forked repository that you created on GitHub. Git knows about the branches on the remote as well as on your local system.
So because of this synchronization between your local repository and the remote repository, Git knows that any commits you make locally — and will likely push back to the remote — belong on a particular, matching, remote branch. Equally well, Git knows that any changes made on a branch on the remote — perhaps by a fellow developer somewhere in the world — belong in a specific, matching directory on your local system.

Viewing local and remote branches

To see all of the branches that Git knows about on this repository, either local or remote, execute the following command:

```
git branch --all
```

Git will respond with something similar to the following:

```
* main
  remotes/origin/HEAD -> origin/main
  remotes/origin/clickbait
  remotes/origin/main
  remotes/origin/master
```

Git shows you all of the branches in your local and remote repositories. In this case, the remote only has two branches: **clickbait** and the deprecated **master** branch. All of the other branches listed are effectively **main** or pointers to **main**.
You have some work to do on the clickbait branch. If everyone else is doing it, you should, too, right? To get this branch down to your machine, tell Git to start tracking it, and switch to this branch all in one action, execute the following command:

```
git checkout --track origin/clickbait
```

Git responds with the following:

```
Branch 'clickbait' set up to track remote branch 'clickbait' from 'origin'.
Switched to a new branch 'clickbait'
```

Explaining origin

OK, what is this `origin` thing that you keep seeing?
`origin` is another one of those convenience conventions that Git uses. Just like **main** is the default name for the first branch created in your repository, `origin` is the default alias for the location of the remote repository from where you cloned your local repository.
To see this, execute the following command to see where Git thinks `origin` lives:

```
git remote -v
```

You should see something similar to the following:

```
origin  https://www.github.com/belangerc/ideas (fetch)
origin  https://www.github.com/belangerc/ideas (push)
```

You'll have something different in your URLs, instead of `belangerc`. But you can see here that `origin` is simply an alias for the URL of the remote repository. That's all.
To see Git's view of all local and remote branches now, execute the following command:

```
git branch --all -v
```

Git will respond with its understanding of the current state of the local and remote branches, with a bit of extra information provided by the `-v` (verbose) option:

```
* clickbait                e69a76a Adding suggestions from Mic
  main                     477e542 [ahead 8] Adding .gitignore files and HTML
  remotes/origin/HEAD      -> origin/main
  remotes/origin/clickbait e69a76a Adding suggestions from Mic
  remotes/origin/main      f65a790 Updated README.md to reflect current working book title.
  remotes/origin/master    c470849 Going to try this livestreaming thing
```

Git tells you that you are on the **clickbait** branch, and you can also see that the hash for the local **clickbait** branch is the same as the remote one, as you'd expect.
Of interest is the **main** branch, as well. Git is tracking your local **main** branch against the remote one, and it knows that your local **main** branch is eight commits ahead of the remote. Git will also let you know if you're behind the remote branch as well; that is, if there are any commits on the remote branch that you haven't yet pulled down to your local branch.

Viewing branches graphically

To see a visual representation of the current state of your local branches, execute the following command:

```
git log --oneline --graph
```

The tip of the graph, which is the latest commit, tells you where you are:

```
* e69a76a (HEAD -> clickbait, origin/clickbait) Adding suggestions from Mic
```

Your current `HEAD` points to the clickbait branch, and you're at the same point as your remote repository.

A shortcut for branch creation

I confess, I took you the long way 'round with that command `git checkout --track origin/clickbait`, but seeing the long form of that command hopefully helped you understand what Git actually *does* when it checks out and tracks a branch from the remote.
There's a much shorter way to checkout and switch to an existing branch on the remote: `git checkout clickbait` works equally well, and is a bit easier to type and to remember.

When you specify a branch name to `git checkout`, Git checks to see if there is a local branch that matches that name to switch to. If not, then it looks to the `origin` remote, and if it finds a branch on the remote matching that name, it assumes that is the branch you want, checks it out for you, and switches you to that branch. Rather nice of it to take care of all that for you.
There's also a shortcut command which solves the two-step problem of `git branch <branchname>` and `git checkout <branchname>`: `git checkout -b <branchname>`. This, again, is a faster way to create a local branch.
Now that you have seen how to create, switch to, and delete branches, it's time for the short challenge of this chapter, which will serve to reinforce what you've learned and show you what to do when you want to delete a local branch that already has a commit on it.

Challenge

Challenge: Delete a branch with commits

You don't want to muck up your existing branches for this challenge, so you'll need to create a temporary local branch, switch to it, make a commit, and then delete that branch.

. Create a temporary branch with the name of **newBranch**.

. Switch to that branch.

. Use the `touch` command to create an empty **README.md** file in the root directory of your project.

. Add that new **README.md** file to the staging area.

. Commit that change with an appropriate message.

. Checkout the **main** branch.

. Delete **newBranch** — but Git won't let you delete this branch in its current state. Why?

. Follow the suggestion that Git gives you to see if you can delete this branch.
Remember to use `git status`, `git branch` and `git log --oneline --graph --all` to help get your bearings as you work on this challenge.
If you get stuck, or want to check your solution, you can always find the answer to this challenge under the **challenge** folder for this chapter.

Key points

A commit in Git includes information about the state of the files in your repository, along with metadata such as the commit time, the commit creator, and the commit's parent or parents.
The hash of your commit becomes the unique ID, or key, to identify that particular commit in your repository.
A branch in Git is simply a reference to a particular commit by way of its hash.
`main` is simply a convenience convention, but has come to be accepted as the original branch of a repository.
Use `git branch <branchname>` to create a branch.
Use `git branch` to see all local branches.
Use `git checkout <branchname>` to switch to a local branch, or to checkout and track a remote branch.
Use `git branch -d <branchname>` to delete a local branch.
Use `git branch --all` to see all local and remote branches.
`origin`, like `main`, is simply a convenience convention that is an alias for the URL of the remote repository.
Use `git checkout -b <branchname>` to create and switch to a local branch in one fell swoop.

Where to go from here?

Get used to branching in Git, because you'll be doing it often. Lightweight branches are pretty much *the* reason that Git has drawn so many followers, as it matches the workflow of concurrent development teams.
But there's little point in being able to branch and work on a branch, without being able to get your work joined back up to the main development branch. That's **merging**, and that's exactly what you'll do in the next chapter!

8. Merging 6. Git Log & History