1

A Crash Course in Git Written by Bhagat Singh & Chris Belanger

It can be a bit challenging to get started with command-line Git if you haven't done much work on the command line before. Since you'll be interacting with Git through the command line throughout this book, this chapter will take you through a quick crash course on how to do it.

There's a common workflow that serves as the foundation of most interactions you'll have with Git:

Create a fork of an existing repository.
Copy a remote repository to your own computer.
Create a separate working area in the repository where you can make changes without affecting anyone else.
Flag those changes to be saved to the local copy of the repository.
Save those changes in your local copy of the repository.
Synchronize those changes with the remote repository.
Optionally, notify the repository owner that your changes are ready to be reviewed.

This chapter will take you through all the above actions to help you get familiar with the basics of working with Git through the command line.

Although this chapter won't explain everything in detail, it will give you enough familiarity with a Git repository and the basic Git workflow to better understand the chapters to come.

What are remote repositories?

A remote repository is simply a collection of all the files of a project, hosted somewhere other than your local machine. They could be hosted internally on your network, but more often, you'll work with remote repositories hosted on cloud services like GitHub and GitLab.

Having a centralized remote repository makes sharing and contributing to a project easy. Instead of sending files to interested people, you simply point them to the hosted remote repository to get them up and running as quickly as possible.

The first step is to create your own personal online copy, or **fork**, of the remote repository. That gives you a place to work online and lets you follow the instructions in this chapter without affecting any of the millions of other people reading this book and following along themselves.

Forking the remote repository

Navigate to the following URL in a browser:

https://github.com/raywenderlich/programmer-jokes

You'll see a screen like the following:

raywenderlich/programmer-jok

github.com/raywenderlich/programmer-jokes

Search or jump to... / Pull requests Issues Marketplace Explore

raywenderlich / programmer-jokes Public

Unwatch 4 Star 8 Fork 319

Code Issues Pull requests 244 Actions Projects Wiki Security Insights

main 2 branches 0 tags Go to file Add file Code

crispy8888 Adds programmer jokes. cb7e6aa on 29 May 2020 2 commits

LICENSE	Initial commit	16 months ago
README.md	Adds programmer jokes.	16 months ago

README.md

programmer-jokes

In order to understand recursion you must first understand recursion.

There are 10 kinds of people in this world: Those who understand binary, and those who don't.

An SEO expert walked into a bar, pub, liquor store, brewery, alcohol, beer, whiskey, vodka...

Why did the two functions stop calling each other? Because they had constant arguments.

About

Sample repository for the Git Apprentice book

Readme

MIT License

Releases

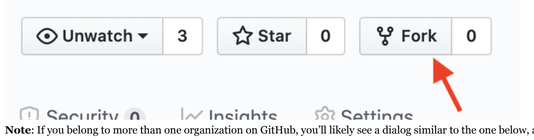
No releases published
Create a new release

Packages

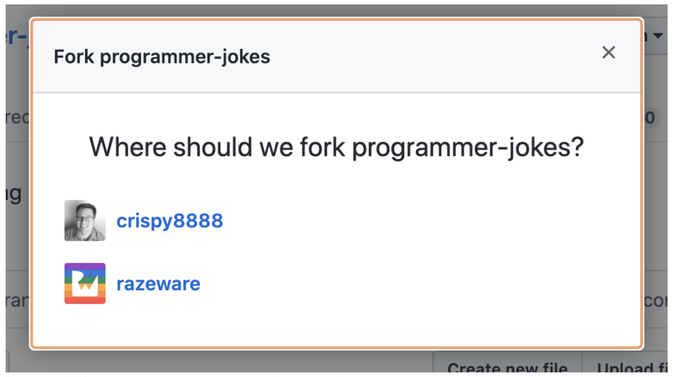
No packages published
Publish your first package

© 2021 GitHub, Inc. Terms Privacy Security Status Docs Contact GitHub Pricing API Training Blog About

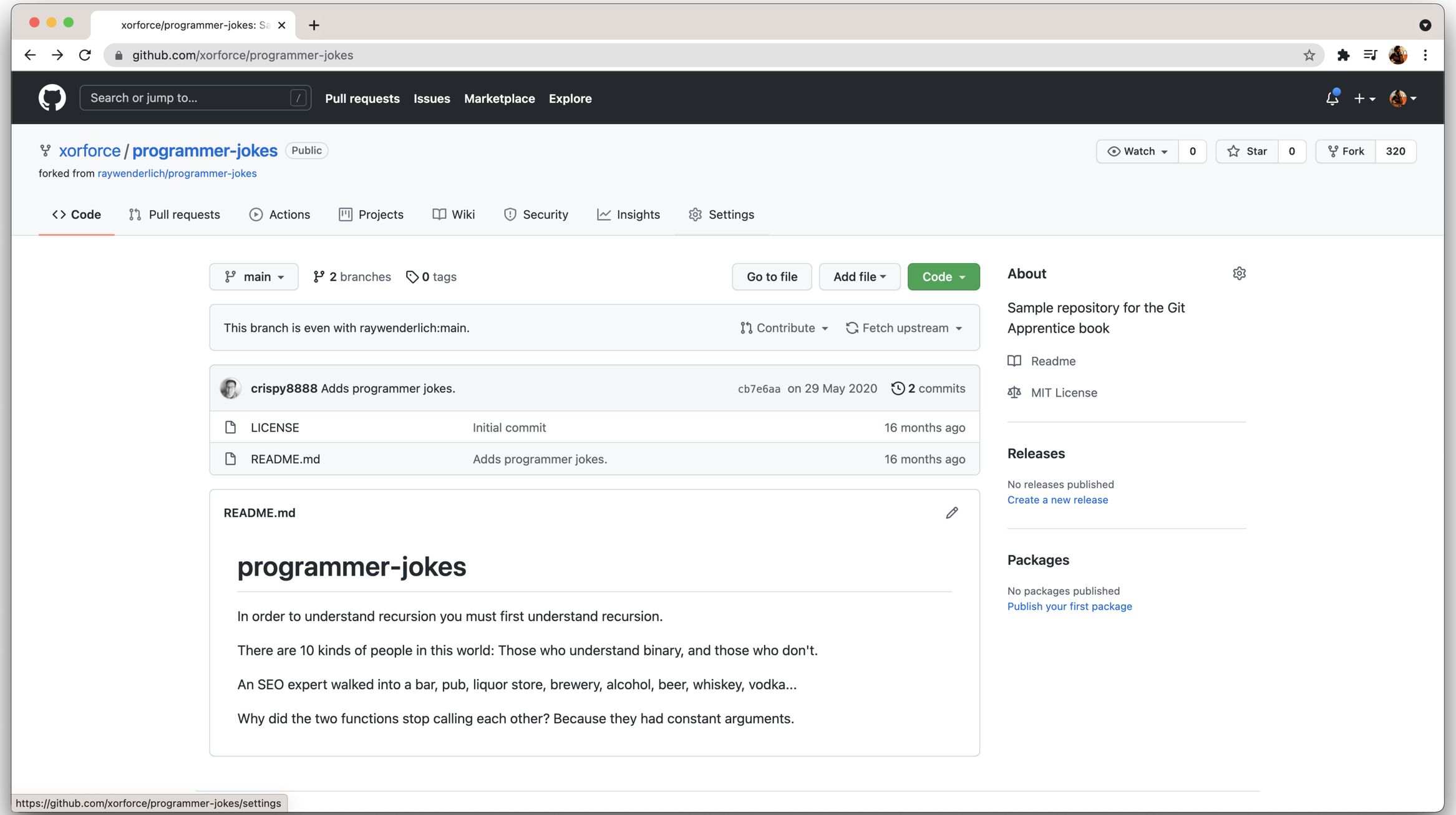
This is the main GitHub page for the project you'll use in this book. You'll cover all the details about GitHub later.
Ensure you're logged in with your own GitHub username, then click the **Fork** button in the top right-hand corner of the page:



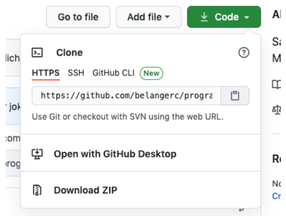
Note: If you belong to more than one organization on GitHub, you'll likely see a dialog similar to the one below, asking you where to fork the **programmer-jokes** repository:



In this case, GitHub isn't really asking you *where* to fork to physically; it's asking under which account you want to create the fork. Choose your own username.
You'll see a progress screen while GitHub creates your repository fork under your account. When GitHub's done creating that fork, you'll see another screen that looks a lot like the original page, except that now you're working from a different location.



And that's just what you want. This is an exact replica of the original programmer-jokes repository, except this copy lives under your own account. That means you can do anything you like to this repository, even delete it, without affecting the original repository that lives under the `raywenderlich` organization.
To get started, you'll need to copy, or **clone**, this remote repository to your local workstation. To do that, you'll need the remote URL of this repository. It's easy to get — simply click the **Code** button on the page, then click the small clipboard icon next to the `https://github.com/username...` URL in the dialog:



You now have the remote URL of this repository in your clipboard.
You're done with this webpage for a bit — you're now ready to start working with Git on the command line.
Open Terminal, PowerShell or the appropriate console prompt on your system and get ready to follow along.

Cloning the repository

At the command prompt, type the following command *without* pressing the **Enter** key:

```
git clone
```

After that, press the spacebar to insert a space character. Then paste what's in your clipboard to the command line using **Command-V** or **Control-V**, depending on what the **Paste** command is on your operating system.
You should have something similar to the following in your command prompt:

```
git clone https://github.com/<your-username>/programmer-jokes.git
```

Now, to break that down a little:
`git` is the name of the command-line Git tool. Every interaction you have with Git on the command line will start with `git` and be followed by the Git command you want to execute.
`clone` is the name of the command you want to execute. `clone` tells Git to copy a specific named repository to your local machine.
`https://github.com/<your-username>/programmer-jokes` is the full URL to the repository you want to clone. Breaking that down further, `https://github.com` is the cloud service that hosts the repository, `<your-username>` is the owner of the fork of this repository, and `programmer-jokes` is the name of the repository you want to clone.
Press **Enter** or **Return** to execute that command. Git gives you a bit of output on the command line to tell you what it's done:

```
Cloning into 'programmer-jokes'...
remote: Enumerating objects: 7, done.
remote: Total 7 (delta 0), reused 0 (delta 0), pack-reused 7
Receiving objects: 100% (7/7), done.
```

The details of that output aren't important, but do take a look at that first line:

```
Cloning into 'programmer-jokes'...
```

Git's telling you that it's cloning the remote repository into a new directory it's created: **programmer-jokes**.
Navigate into that directory from the command line with the following command:

```
cd programmer-jokes
```

Next, execute the following to get a list of the files in that directory in long format — just because it's easier to read:

```
ls -l
```

You'll see output similar to the following:

```
-rw-r--r--  1 chrisbelanger  staff  1070 29 May 11:25 LICENSE
-rw-r--r--  1 chrisbelanger  staff   370 29 May 11:25 README.md
```

There are two files in this repository: **LICENSE**, which has some boring legal information about the contents of the repository, and **README.MD**, which is a simple text file that contains some groan-worthy programming jokes.
Now that you have the repository cloned to your local machine, the next step is to create a separate working space, or **branch**, where you can change the contents of **README.md** without fear of messing up the original contents of the repository.

Creating a branch

Branches are, conceptually, *copies* of the original contents of the repository. You can work in a branch without affecting the original contents of the repository until you are ready to merge all your work back together again.
If you've ever made a copy of an important document before you started editing it, the concept of branching is exactly the same.
At the command line, execute the following to create a new branch:

```
git branch my-joke
```

Breaking this down:
git, again, is the name of the command-line tool.
branch is the name of the command you want Git to execute.
my-joke is the name of the branch you want to create. The name you give to a branch isn't important, but you generally want to give it a descriptive name, just as you would when creating new folders on your desktop.
You can see that Git's created a new branch by executing the following command:

```
git branch
```

This looks similar to the command above, but in this case, you haven't supplied a branch name. Git understands this to mean, "Oh, you don't want to create a branch, you just want to look at all the branches I know about."
Git responds with the following output:

```
* main
  my-joke
```

`main` is the original copy of the repository, while `my-joke` is the branch you just created. The asterisk `*` indicates which branch you're currently working in. Right now, you're still on `main`, but that's not what you want — you want to change to `my-joke` so you don't affect `main`.
To switch to the `my-joke` branch, execute the following:

```
git checkout my-joke
```

Ah, a new command: `checkout`. You might have expected a command like `switch-branch`, but Git thinks of switching branches in terms of "checking out". It's similar to how you check out a book at a library: That copy of the book is now exclusively yours to work with until you return it to the library.
Git responds to the `checkout` command with the following:

```
Switched to branch 'my-joke'
```

If you're the paranoid type, like me, you can confirm this with the command below:

```
git branch
```

Git puts your fears to rest with the following response:

```
  main
* my-joke
```

The asterisk tells you that you're now working securely inside the `my-joke` branch, and that your changes won't affect the main copy of the repository.
Now, it's time for you to add a stunning joke to **README.md**.

Making and staging changes

README.md is simply a text file. Open it in a text editor of your choice and you'll see it has the following contents:

```
# programmer-jokes

In order to understand recursion you must first understand recursion.

There are 10 kinds of people in this world: Those who understand binary, and those who don't.

An SEO expert walked into a bar, pub, liquor store, brewery, alcohol, beer, whiskey, vodka...

Why did the two functions stop calling each other? Because they had constant arguments.
```

They're *hilarious*, right!? Well, you can definitely improve upon them by adding your own joke to this list.
Just in case you don't have a handy stash of programmer jokes at your disposal, add the following line to the text file:

```
Why couldn't the confirmed bachelor use Git? Because he was afraid to commit!
```

Save your changes and exit the text editor.
Git's pretty smart, but it also knows not to assume too much. Just because you've modified a file, Git isn't going to assume that you want this in the repository. Instead, it will wait for you to flag those changes to be made to the repository.
Execute the command below to flag the change you made to the file, remembering that case is important:

```
git add README.md
```

The `add` command tells Git to add, or **stage**, the changes you've made to **README.md** to the list of things to add to the repository. In this case, you only have one change to one file, but in practice, you'll usually have lots of changes to lots of files.
Now that Git is aware that you want to make that change, you'll need to **commit** it to your local repository.

Committing changes

Git's got your back here. It knows that sometimes you might add a change, but then have second thoughts, or you might have other changes that you want to stage at the same time. That's why Git separates the act of staging files from the act of committing those changes.
Committing is the act of saying, "Yes, I have these changes ready, and I want to formally record those changes in my local copy of the repository."
Not only does Git record those changes formally, it also allows you to provide a **commit message** to give some context about the content of those changes to others — or even to your future self.
Commit your changes now with the following command:

```
git commit -m "Adding my new joke"
```

That tells Git to formally record your current set of changes — although in this case, you only have one change, which is the joke you added.
Git responds with output similar to the following:

```
[my-joke f8f8854] Adding my new joke
 1 file changed, 1 insertion(+)
```

That may look confusing, but there's a lot of information in there:
my-joke is the branch you're committing your changes to.
f8f8854 is the unique identifier for your commit, also known as the **commit hash** of your changes. You use this identifier to uniquely reference this specific commit in the future. Note, if you're following along and typing commands as you read, your hash will be different.
Adding my new joke is the commit message you added above.
1 file changed, 1 insertion(+) provides some context about what was changed in this commit: one file, with one line added.
You've formally recorded these changes in your local repository, but now you need to get them synchronized, or **pushed**, back to the remote repository.

Pushing your changes

Most of Git's actions are performed from the perspective of your local workstation. So the action you're taking is to **push** your local commit to the remote server.
Execute the following command to send your local changes to your forked remote repository:

```
git push --set-upstream origin my-joke
```

That's a little obtuse, for sure. Don't be too concerned at this point; you'll cover what this means in future chapters. Essentially, here's what the various pieces mean:
push tells Git to put your local changes on the server.
--set-upstream tells Git to form a tracking link for this branch between your local repository and the remote repository.
origin is a convention that references the remote repository.
my-joke is the branch you want to push.
Git responds with a pile of output:

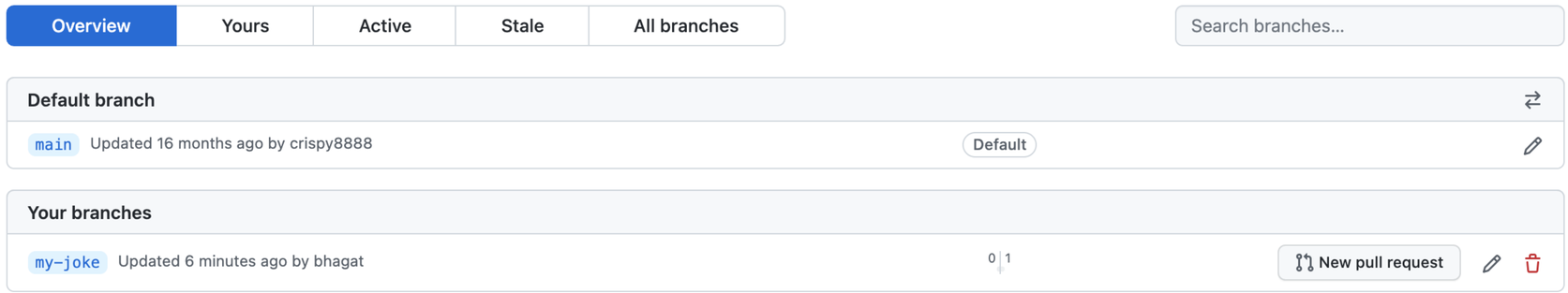
```
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 8 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 389 bytes | 389.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
remote:
remote: Create a pull request for 'my-joke' on GitHub by visiting:
remote:      https://github.com/<your-username>/programmer-jokes/pull/new/my-joke
remote:
To https://github.com/<your-username>/programmer-jokes.git
 * [new branch]      my-joke -> my-joke
Branch 'my-joke' set up to track remote branch 'my-joke' from 'origin'.
```

Note: You may be prompted by the command line to enter your GitHub username and password. If so, then provide them and hit Enter or Return after you do.
Git's successfully pushed your changes to the remote repository, but there's one thing left to do: Signal to anyone else using this repository that you have something you'd like to integrate, or **pull**, into the remote repository. You do that with a mechanism called a **pull request**.

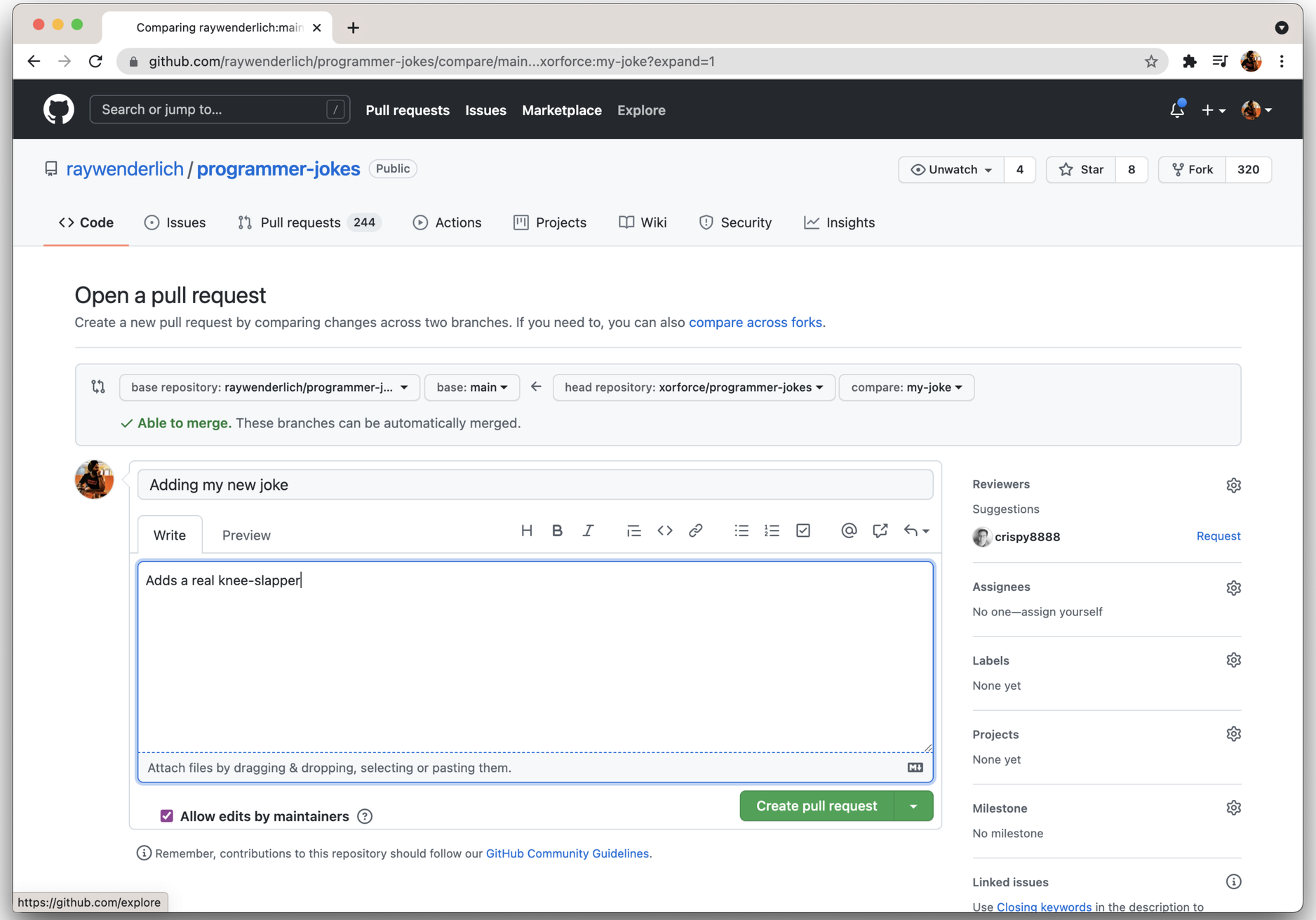
Creating a pull request

Return to your browser, and if you don't already have it open, open the main GitHub page for your forked repository at `https://github.com/<your-username>/programmer-jokes`.
If you look *really* closely, You'll notice that the page has changed a little, to reflect the fact that your changes made it successfully to the remote repository:

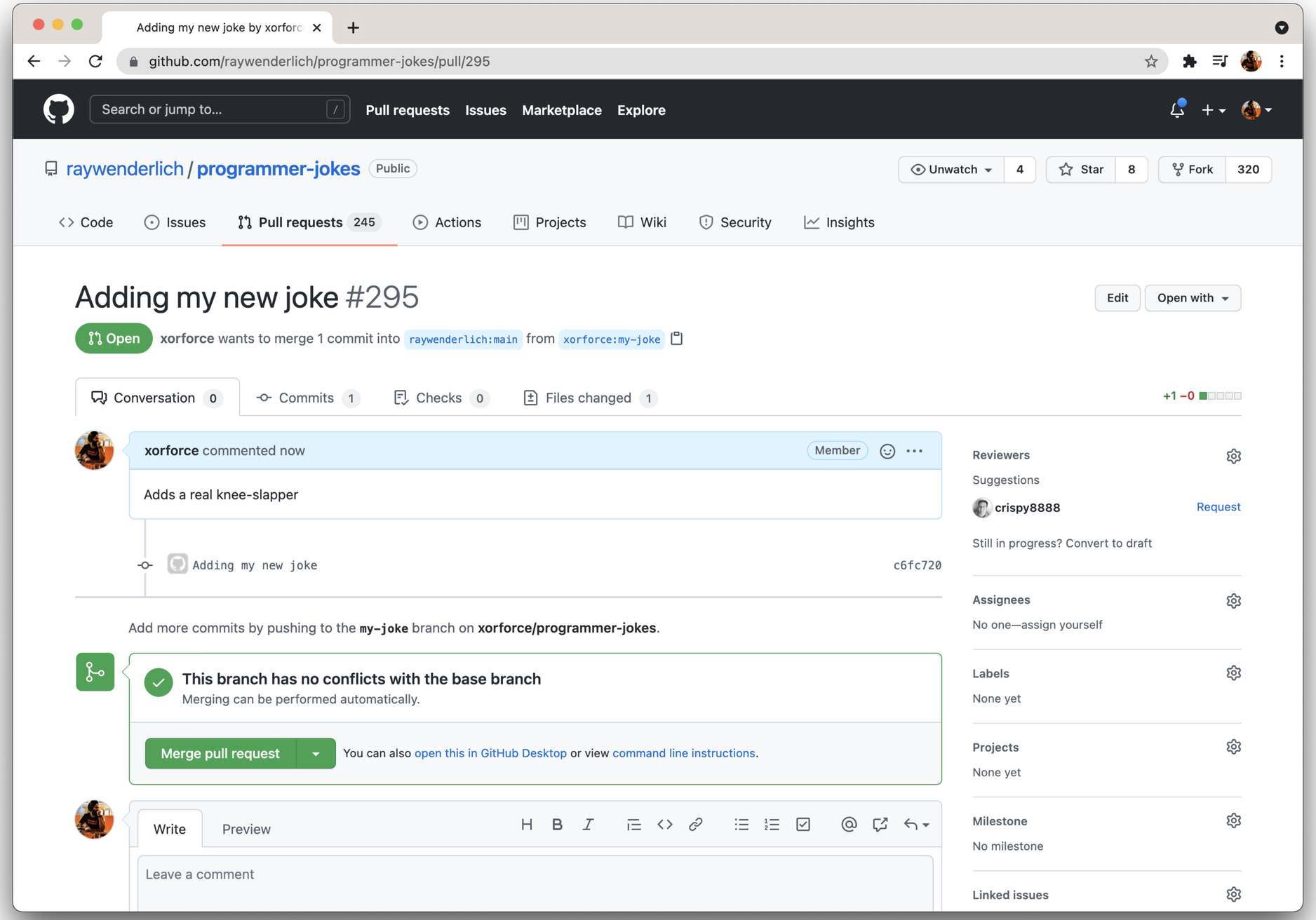
You can see GitHub is telling you that there's now two branches in this repo. Click on that "3 branches" link, and you'll see the following page, showing your new branch, along with a "New pull request" button:



Click that button and you'll be taken to another page, where you can enter some details about your change. Enter **Adds a real knee-slapper** to the large text box to give some extra detail about the changes contained in this pull request, then click the **Create pull request** button:



GitHub takes you to yet another page, where you can see that your pull request is now active, along with various information related to your pull request:



That's as far as you need to go with this short crash course in Git. If you didn't understand quite everything that happened along the way, don't worry! This chapter was just to get you familiar with the basic fork → clone → branch → add → commit → push → pull request mechanism of Git. The rest of the book will look at each of these steps in detail, along with much, much more information about the more obscure commands in Git. You'll also get a tour of the internals of Git, which may help you understand a little better why Git does what it does.
Head on to the next chapter, where you'll start by looking at the `clone` command in more depth. See you there!

2. Cloning a Repo vi. Introduction

Have a technical question? Want to report a bug? You can ask questions and report bugs to the book authors in our official book forum here.