10
Creating a Repository Written by Chris Belanger & Bhagat Singh

You've come a long way in your Git journey, all the way from your first commit, to learning about what Git does behind the scenes, to managing some rather complicated merge scenarios. But in all your work with repositories, you haven't yet learned exactly *where* a repository comes from. Sure, you've cloned a repository, and you've forked repositories and worked with remotes, but how do you create a repository and a remote *from scratch*?

This chapter shows you how to create a brand-new repository on your local machine, and how to create a remote to host your brand-new repository for all to see.

Getting started

Many people will blindly tell you that the easiest way to create a repository is to "Go to GitHub, click 'New Repository', and then clone it locally." But, in most cases, you'll have a small project built up on disk before you ever think about turning it into a full-fledged repository. So this chapter will put you right into the middle of your project development and walk you through turning a simple project directory into a full-fledged repository.

But, first, you'll need a project! Check the **starter** folder for this chapter; inside, you'll find a small starter project that is the starting webpage for the sales page for this book.

Copy the entire **git-apprentice-web** directory from the **starter** folder into your main **GitApprentice** folder.

Now, open up your terminal program and navigate into the **git-apprentice-web** directory. If you've been following along with the book so far, you're likely still in the **GitApprentice/ideas** folder, so execute the following command to get into the **git-apprentice-web** subdirectory:

```
cd ../git-apprentice-web/
```

Once there, execute the following command to tell Git to set this directory up as a new repository:

```
git init
```

Git tells you that it has set up an empty repository:

```
Initialized empty Git repository in /Users/chrisbelanger/GitApprentice/git-apprentice-web/.git/
```

Why does Git tell you it's an empty repository, when there are files in that directory? Think back to how you staged files to add to a repository: You have to use the `git add` command to tell Git what to include in the repository; Git wouldn't just assume it should pick up any old file lying around. And the same is true, here; Git has created an empty repository, just waiting for you to add some files.

As of late 2020, GitHub now uses `main` as the default branch name for all new repositories. But if you have a plain vanilla install of Git on your local workstation, you're likely configured with `master` as your default branch name.

To check this, simply execute the following to see what `git init` set as your first branch name:

```
git branch
```

In my case, Git responds with the following:

```
* master
```

To fix this, execute the following command:

```
git branch -M main
```

Although Git gives you no output, this command changes the local name of your branch from `master` to `main`. Again, it pays to be paranoid with Git, so execute `git branch` again to confirm that your branch has been renamed to `main`.

Now, before you add any files, you'll want to get two things in your repository that are good hygiene for any repository that's designed to be shared online: a **LICENSE** file, and a **README** file.

Creating a LICENSE file

It's worth understanding why you need a license file, before you go and create one blindly.

Having a license file in your repository makes it clear how others may, or may not, use your code. In this modern, digital age, some people believe that copying/stealing/borrowing/reusing anything is fair game, but most people will want to respect your license terms, even though you may be providing the code freely online.

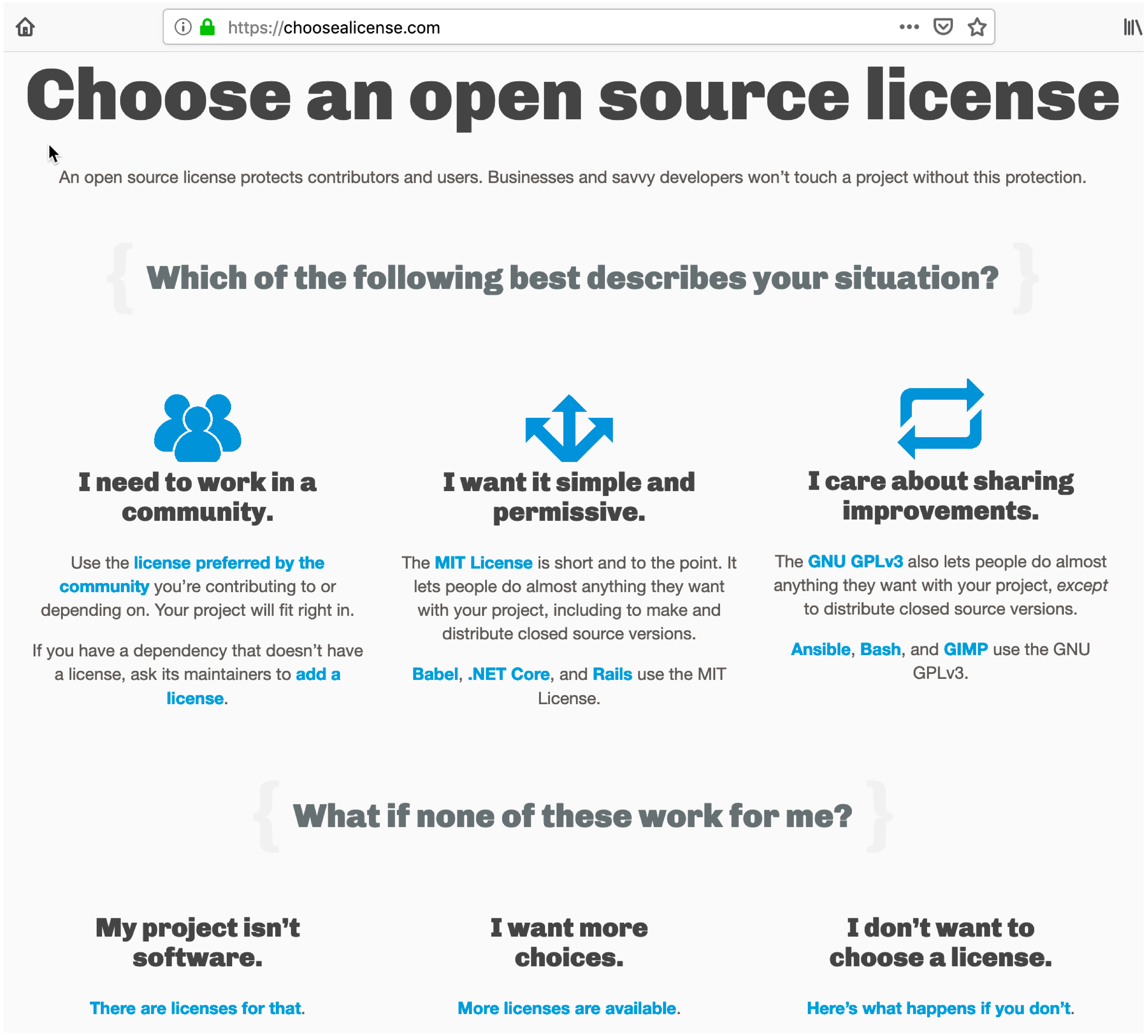
Having a license outlines how others may contribute to your project and what their rights are. The interesting bit comes in when you *don't* include a license to your work. If you create a project and stick it up on GitHub, without a license, you're stating that *no one* has the license to use your code in *any* situation — they can look at it, but that's about it.

That's all well and good if "look but don't touch" is truly what you want, but if you're inviting others to collaborate with you, then having no license means that once someone else touches the code *it's not clear who owns the copyright anymore*. Having a license file included with your code makes it clear where the ownership of this code lies.

True, having a license included with your project won't protect you from code burglars who just want to take your work and use it without your permission. But what it *does* do is indicate the terms of use and reuse of your project to anyone who wants to collaborate in a fair manner, or use your work in any other manner. It's a live-and-let-live kind of thing.

Now, with that said, what kind of license should you choose? That's not always an easy question to answer. Most of the time, your projects will have just code in them, but what if they contain images? What if they contain hardware designs? 3D printing files? Your open-source book manuscript? Fonts you designed and want to open-source? What if your project is a mix of these or more?

There's a great site out there that will help you navigate the ins and outs of your project, and help you choose a license for your new project. Navigate to https://choosealicense.com/, and you'll see a lot of options:



https://choosealicense.com

Choose an open source license

An open source license protects contributors and users. Businesses and savvy developers won't touch a project without this protection.

{ Which of the following best describes your situation? }

I need to work in a community.

Use the **license preferred by the community** you're contributing to or depending on. Your project will fit right in.

If you have a dependency that doesn't have a license, ask its maintainers to **add a license**.

I want it simple and permissive.

The **MIT License** is short and to the point. It lets people do almost anything they want with your project, including to make and distribute closed source versions.

Babel, **.NET Core**, and **Rails** use the MIT License.

I care about sharing improvements.

The **GNU GPLv3** also lets people do almost anything they want with your project, *except* to distribute closed source versions.

Ansible, **Bash**, and **GIMP** use the GNU GPLv3.

{ What if none of these work for me? }

My project isn't software.

There are licenses for that.

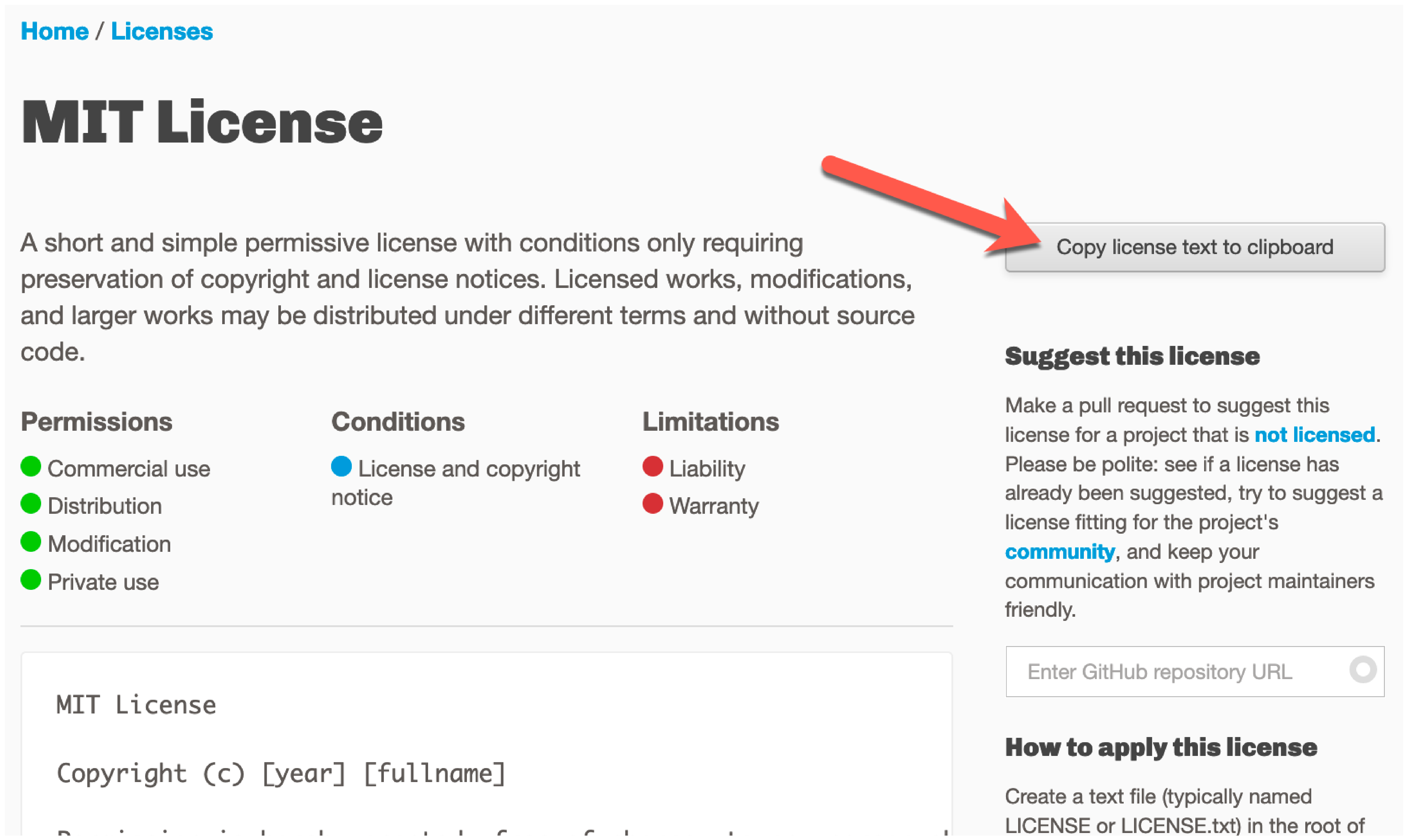
I want more choices.

More licenses are available.

I don't want to choose a license.

Here's what happens if you don't.

You can explore the site at your leisure, but, in this case, I am happy for others to learn from and reuse my work in any way they like as I build up my webpage. So select the **MIT License** link, and you'll be taken to the main license page for the MIT License, which is one of the most common and most permissive licenses.



Home / Licenses

MIT License

A short and simple permissive license with conditions only requiring preservation of copyright and license notices. Licensed works, modifications, and larger works may be distributed under different terms and without source code.

Copy license text to clipboard

Permissions	Conditions	Limitations
🟢 Commercial use	🔵 License and copyright notice	🔴 Liability
🟢 Distribution		🔴 Warranty
🟢 Modification		
🟢 Private use		

```
MIT License

Copyright (c) [year] [fullname]
```

Suggest this license

Make a pull request to suggest this license for a project that is **not licensed**. Please be polite: see if a license has already been suggested, try to suggest a license fitting for the project's **community**, and keep your communication with project maintainers friendly.

Enter GitHub repository URL

How to apply this license

Create a text file (typically named LICENSE or LICENSE.txt) in the root of

Click the **Copy license text to clipboard** button to copy the text of the MIT license to your clipboard.
Now, return to your terminal program, create a new file named **LICENSE** (yes, uppercase, and no extension required) in the root folder, and populate it with the contents of the clipboard. Save your work when you're done.
That takes care of the license file. Now, it's time to turn your attention to the README file.

Creating a README file

The README is much more straightforward than the license file. Inside the README, you can put whatever details you want people to know about you, your project, and anything that will help them get started using your project.
The common convention is to craft README files in Markdown, primarily so that they can be rendered in an easy-to-read format on the front page of your repository on GitHub, GitLab or other cloud hosts.
Create a new file in the root directory of your project named **README.md** and populate it with the following information (changing whatever you like to suit):

```
# git-apprentice-web

This is the main website for the Git Apprentice book, from raywenderlich.com.

contact: @crispytwit
```

Save your changes and exit out of the editor.
You've got your current project, LICENSE file, and the README file — looks like you're ready to commit your files to the repository.
To see what's outstanding for your first commit, execute `git status` to see what Git's view of your working area looks like:

```
~GitApprentice/git-apprentice-web $ git status
On branch main

No commits yet

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

        LICENSE
        README.md
        css/
        images/
        index.html

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

That looks as you'd expect: The basic files for the project are there, along with the new LICENSE and README.md file.
By this point, you should be able to stage and commit this collection of files to your new repository. Try to stage and commit the complete set of files on your own first, before following the instructions below. Remember: If you mess things up, you can simply use `git reset` to revert your changes.
Stage the files for commit with the following command:

```
git add .
```

This adds everything in the current directory and subdirectories.
Now, commit your changes to the repository, providing a sensible commit message:

```
git commit -m "Initial commit of the web site, README and LICENSE"
```

Since this is your very first commit into the repository, Git shows you a bit of different output:

```
[main (root-commit) 443f9b3] Initial commit of the web site, README and LICENSE
 5 files changed, 111 insertions(+)
 create mode 100644 LICENSE
 create mode 100644 README.md
 create mode 100644 css/style.css
 create mode 100644 images/SFR_b+w_-_penguin.jpg
 create mode 100644 index.html
```

The very first commit to the repository is a bit special, since it doesn't have *any* parents. Recall earlier when you learned that every commit in Git has at least one parent? Well, this is a special case in which Git creates a root commit for the repository, upon which all future commits will be based.
And that's it! You've made your first commit to your repository. But you're not done — you want to get this repository pushed up to a remote for the world to *ooh* and *ahh* over. You'll do that in the second half of this chapter.

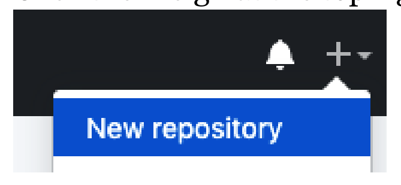
Create mode

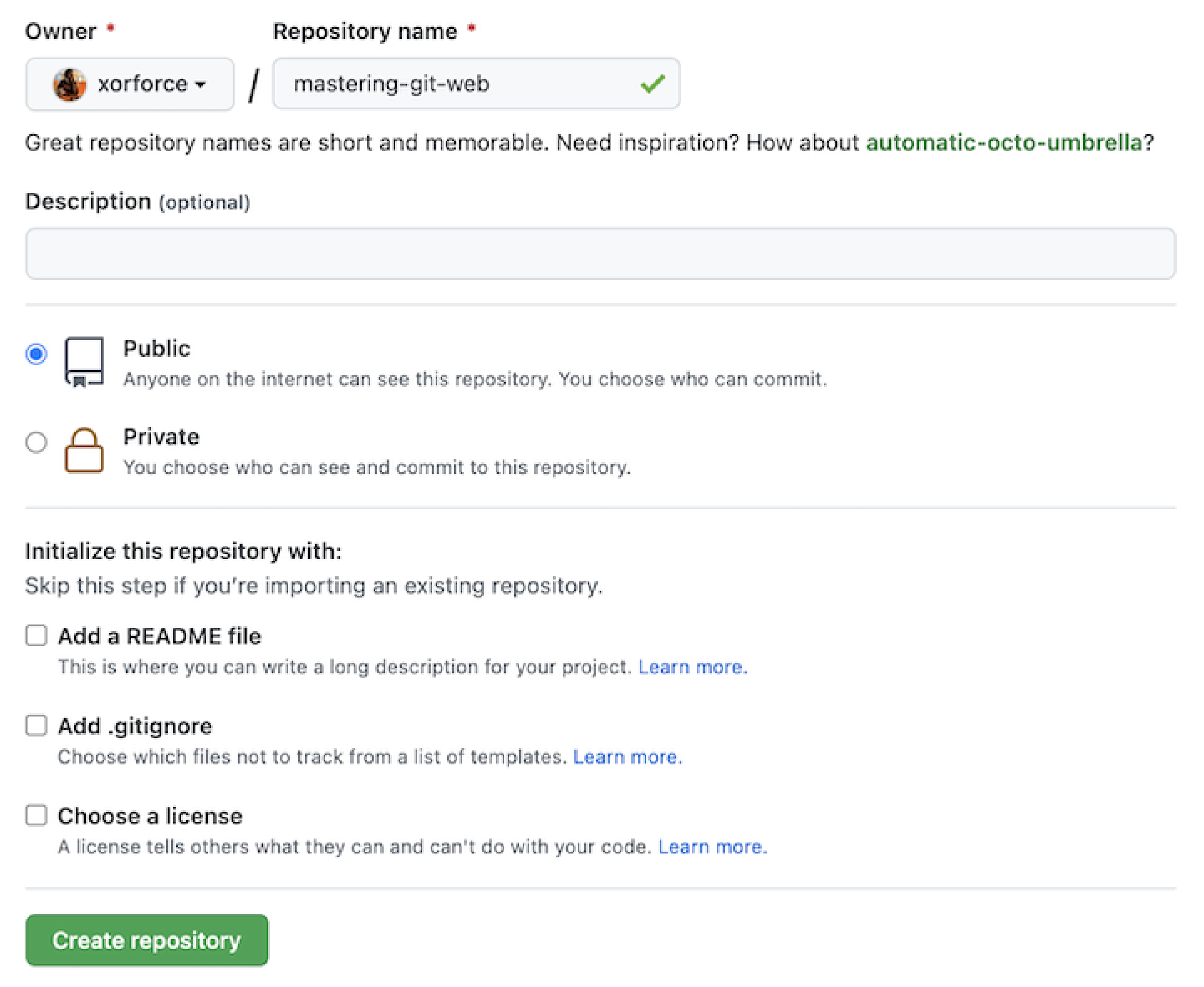
That `create mode` is something you've seen before in the output from `git commit`, and have probably wondered about. It's of academic interest only at this point; it really doesn't affect you much at this stage of your interaction with repositories.
But in the interest of being obsessively thorough, here's what that number with `create mode` means:
The number after `create mode` is an octal (base 8) representation of the type of file you're creating, along with the read/write/execute permissions of that file.

The first part of that binary number is a 4-bit value that indicates the *kind* of file you're creating. In this case, you're creating a regular file, which Git labels with `1000` in binary. There are other types, including symlinks and gitlinks, which you aren't using yet in your Git career.
The next part of that binary number is three unused bits: `000`.
The last part of that binary number is made of nine bits, and represents the UNIX-style permissions of this file. The first three bits hold the owner's read/write/execute permission bits, the next three bits hold the group's read/write/execute bits, and the final three bits hold the global read/write/execute bits.
So since you own the file, Git sets the first three bits to `110` (read, write, but no execution since this isn't an executable binary or script file).
To allow anyone in your group to read but not write to this file, Git assigns `100` (read, no write, no execute).
To allow anyone in the world to read but not write to this file, Git assigns `100` (read, no write, no execute).
When all of that binary is concatenated together, you have `1000` with `000` with `110100100 = 1000000110100100` as the full binary string.
Convert `1000000110100100` to octal (base 8), and you have `100644` as a compact way to indicate the type and permissions of this file.
See? I *told* you it was of academic interest only.

Creating and syncing a remote

At the moment, you have your own repository on your local system. But that's a bit like practicing your guitar in your room your whole life and never jamming out at a party so you can wow your guests with a performance of "Wonderwall." You need to get this project out where others can see and potentially collaborate on it.
Head over to GitHub to create a new remote repository for your project, and log in to your account.
Click the + sign at the top right-hand corner of the screen, and select **New repository**.



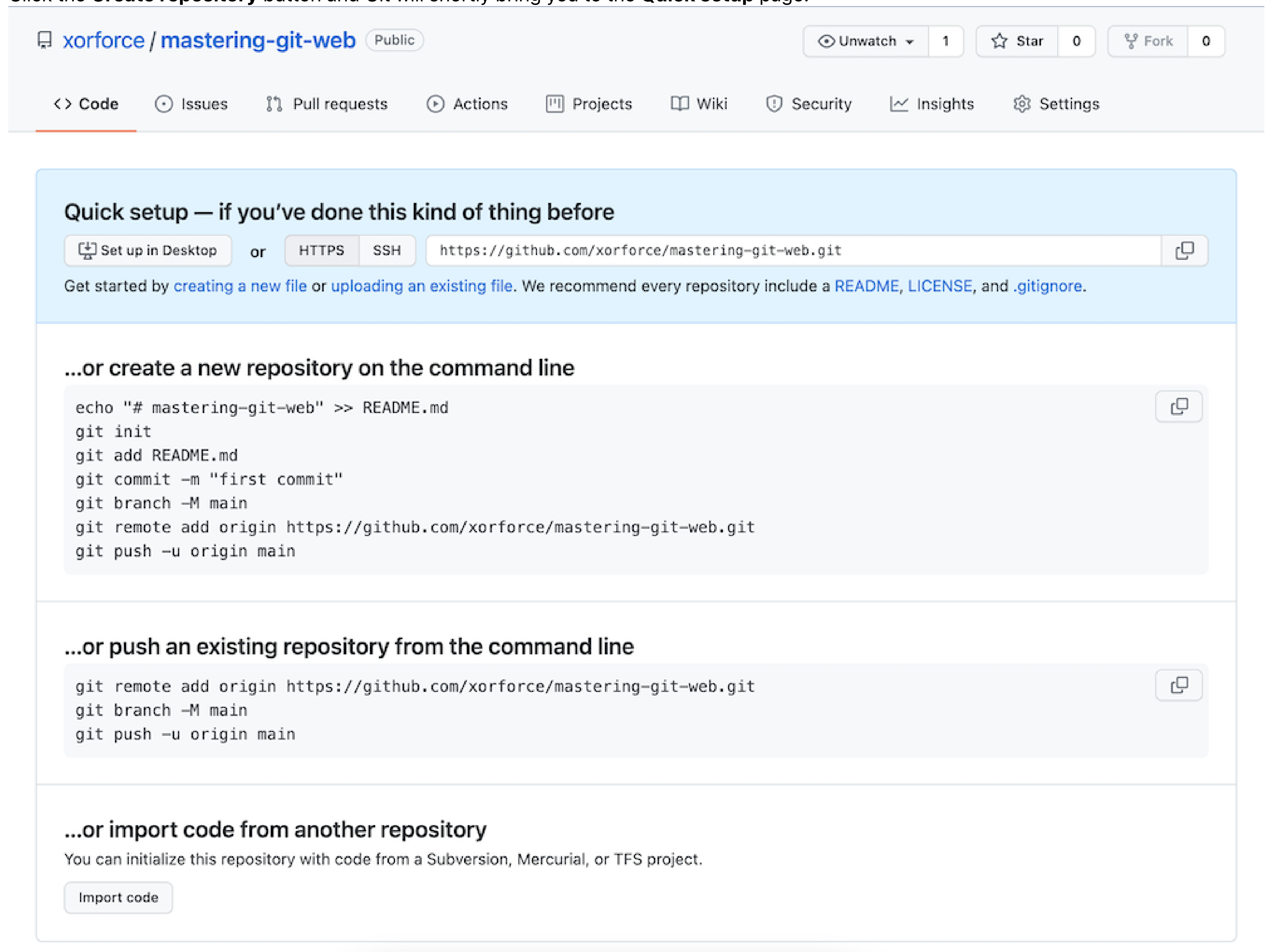

Owner * Repository name *
xorforce / mastering-git-web
Great repository names are short and memorable. Need inspiration? How about automatic-octo-umbrella?
Description (optional)

(●) Public
Anyone on the internet can see this repository. You choose who can commit.
( ) Private
You choose who can see and commit to this repository.

Initialize this repository with:
Skip this step if you're importing an existing repository.
[ ] Add a README file
This is where you can write a long description for your project. Learn more.
[ ] Add .gitignore
Choose which files not to track from a list of templates. Learn more.
[ ] Choose a license
A license tells others what they can and can't do with your code. Learn more.

Create repository

A few details to follow, here:
Give your repository a good name; in this case, I'm going to use the same name as my project's directory name, `git-apprentice-web`, although this isn't strictly necessary.
Leave the repository set to **Public**, so that anyone can see it.
Finally, leave everything in the **Initialize this repository with:** section unchecked, since you will be importing the repository from your local workstation, which already exists and already has a LICENSE and a README.
Click the **Create repository** button and Git will shortly bring you to the **Quick setup** page.



Quick setup — if you've done this kind of thing before
Set up in Desktop or HTTPS SSH https://github.com/xorforce/mastering-git-web.git
Get started by creating a new file or uploading an existing file. We recommend every repository include a README, LICENSE, and .gitignore.

...or create a new repository on the command line

```
echo "# mastering-git-web" >> README.md
git init
git add README.md
git commit -m "first commit"
git branch -M main
git remote add origin https://github.com/xorforce/mastering-git-web.git
git push -u origin main
```

...or push an existing repository from the command line

```
git remote add origin https://github.com/xorforce/mastering-git-web.git
git branch -M main
git push -u origin main
```

...or import code from another repository
You can initialize this repository with code from a Subversion, Mercurial, or TFS project.
Import code

This gives you several instructions on how to get some content into your repository. In your case, you already have an existing repository, so you can use the instructions under **...or push an existing repository from the command line**. Because you're all about that command line Git mastery, right?
Ensure the **HTTPS** option is selected in the top section of this page, next to the repository's URL. Copy the URL provided to your clipboard.
Return to your terminal program, and execute the following to add a new remote to your local repository, substituting in the copied URL of your own repository where necessary:

```
git remote add origin https://github.com/<your-repo-name>/git-apprentice-web.git
```

Git gives you no output from that command, but you can verify that you've added a remote, using the following command:

```
git remote -v
```

You should see your remote shown in the output:

```
origin  https://github.com/<your-username>/git-apprentice-web.git (fetch)
origin  https://github.com/<your-username>/git-apprentice-web.git (push)
```

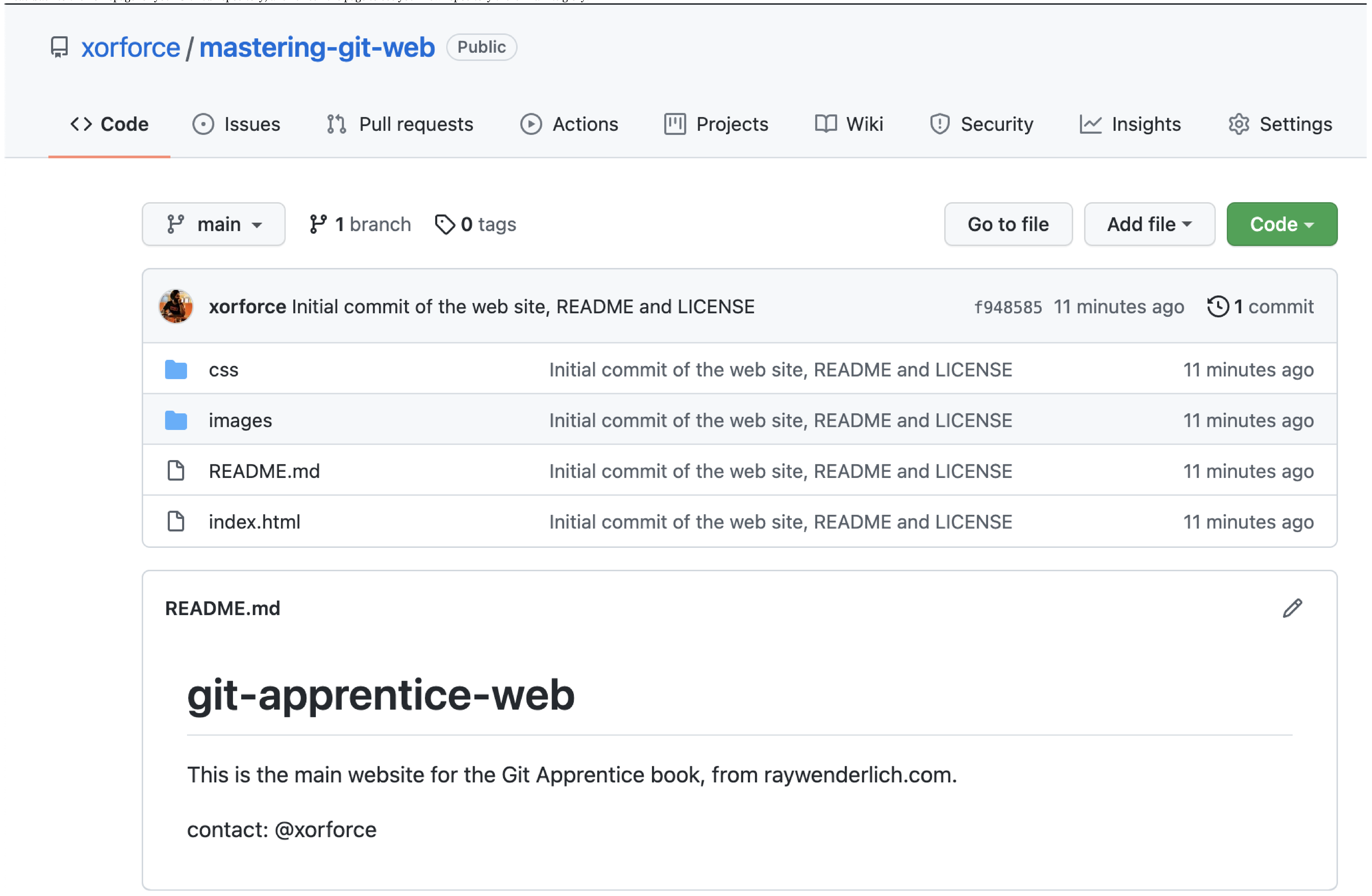
OK - so your local repository is ready to be pushed to the remote. Now, execute the final command from the Quick setup page:

```
git push -u origin main
```

This pushes your changes, as you'd expect, with some corresponding output. The `-u` switch is the shorthand equivalent of `--push-upstream`, which ensures that every branch in your local repository tracks against the corresponding branch in the remote repository. Otherwise, Git won't automatically "know" to track your local branches against the remote ones.
The `origin` option is simply the name of the remote to which you want to push; remember, `origin` is simply the conventional default name of the remote Git uses when it sets up your repository with `git init`, and not a standard.
`main` is the name of the local branch you want to push to your remote.
You can verify that Git has pushed and started tracking your local branch against the remote branch by looking at the final lines in the output from your `git push` command:

```
 * [new branch]      main -> main
Branch 'main' set up to track remote branch 'main' from 'origin'.
```

Head back to the homepage for your GitHub repository, and refresh the page to see your new repository there in all its glory:



xorforce Initial commit of the web site, README and LICENSE f948585 11 minutes ago 1 commit

css	Initial commit of the web site, README and LICENSE	11 minutes ago
images	Initial commit of the web site, README and LICENSE	11 minutes ago
README.md	Initial commit of the web site, README and LICENSE	11 minutes ago
index.html	Initial commit of the web site, README and LICENSE	11 minutes ago

README.md

git-apprentice-web

This is the main website for the Git Apprentice book, from raywenderlich.com.

contact: @xorforce

At this point, your repository is ready for you, or anyone else, to view, clone, and contribute to.

Key points

Use `git init` to set up a Git repository.
It's accepted practice to have a **LICENSE** file and a **README.md** file in your repository.
Use `git add` followed by `git commit` to create the first commit on your new repository.
`create mode` is simply Git telling you what file permissions it's setting on the files added to the repository.
You can create an empty remote on GitHub to host your repository, and you can choose to not have GitHub populate your remote with a LICENSE and README.md by default.
Use `git remote add origin <remote-url>` to add a remote to your local repository.
Use `git remote -v` to see the remotes associated with your local repository.
If your Git installation uses `master` as the default branch in new repositories and you want to push to a newly created GitHub repository with `main` as the default branch, you'll need to execute `git branch -M main` to rename the local `master` branch to `main` to match your remote.
Use `git push --set-upstream origin main` or `git push -u origin main` to push the local commits in your repository to your remote, and to start tracking your local branch against the remote branch.

Where to go from here?

You've come full circle with your introduction to Git! You started out with cloning someone else's repo, made a significant amount of changes to it, learned how to stage and commit your changes, how to view the log, how to branch, how to pull and push changes, and now you're back where you started, except that *you* are the creator of your very own repository. That feels good, doesn't it?
If you're an inquisitive sort, you probably have a lot of unanswered questions about Git, especially how it works under the hood, what merge conflicts are, how to deal with partially complete workfiles, and how to do things that you've heard about online, such as squashing commits, rewriting history, and using rebasing as an alternative to merging.
The next book in the Git Series is called Advanced Git (https://www.raywenderlich.com/books/advanced-git). That book takes you further under the hood of Git, shows you a little more about the internals of Git, and walks you through some scenarios that scare a lot of developers off of using Git in an advanced way. But you'll soon see that the elegance and relative simplicity of Git let you do some *amazing* things that can greatly improve the life of you and your distributed development team.

11. Conclusion 9. Syncing With a Remote