00001000

CHAPTER 8

Source Code Management With Git and GitHub

Learning Objectives

- *State the purpose of source code management*
- *State the purpose of a local repository*
- *State the purpose of a remote repository*
- *Create a remote repository in GitHub*
- *Generate SSH public and private keys with the ssh-keygen tool*
- *Configure private SSH key to load into the ssh-agent automatically*
- *Configure Git and GitHub to use SSH*
- *Clone a GitHub repository with an SSH URL*
- *List the steps involved with a simple Git workflow*
- *Apply the git commands: clone, status, add, commit, and push*
- *Checkout a branch with the git checkout command*
- *List the steps involved with a branch and merge workflow*
- *Open a pull request on GitHub*
- *Merge a branch in GitHub*
- *Avoid common Git pitfalls*

Introduction

If you intend to be a professional software engineer, you need to cultivate a practical understanding of source code management (SCM). Unfortunately, it's a topic often overlooked or ignored completely in school. It's assumed you will learn it later, on the job, sometime down the road. I completely disagree with that approach.

I believe you gain significant benefits by being exposed to SCM tools and processes as early in your education as possible, preferably when first learning how to program. The concepts are easy to grasp, the processes are easy to follow, and the tools, once you understand how they work, are easy to use. Like any skill learned early and practiced often, SCM soon becomes second nature.

Learning and making a habit of SCM delivers many benefits. Primarily, you gain a real, professional skill that sets you apart from the pack. Having a practical understanding of SCM enables you to integrate quickly with a development team. All other things being equal, a hiring manager will most likely choose the candidate who understands SCM over the one who doesn't. Tech companies often gauge how well new developers are doing via a metric called "*Time to First Commit*", which measures the time it takes a new recruit from the day they're hired until they open their first pull request. Seasoned team members appreciate someone who can join a team, quickly get the lay of the land, run with a JIRA ticket, and push their changes to the repository without a lot of hand holding and without screwing things up. In other words, if you know what you're doing when you join the team, you'll quickly gain the respect of your peers.

SCM provides flexibility. If you use multiple computers to work on code in different locations, you can use SCM to host a remote repository online and have local copies of the repository on your desktop machine at home or at work, and another local copy of the repository on your laptop. By adopting a simple process, you can work on code in one location, push your changes to the remote repository, change to another machine at a different location, update the local repository, work on another set of changes, and on and on.

SCM supports disaster recovery. Because your projects are stored on a remote repository, you can suffer a handful of catastrophes and be assured your code base is secure. To make this happen, you must adopt and follow a regular routine of committing and pushing code changes at the end of each coding session.

Finally, SCM doesn't just apply to source code. I use SCM tools and processes, specifically Git and GitHub, to manage the files of this book, business development projects, and other non-technical writing projects. Any project you'd like to host on a remote repository and check out and work on from multiple machines at different times and in different locations, will benefit from SCM tools and processes.

I've talked a lot about source code management in this introduction, but what is SCM exactly?

1 Source Code management (SCM)

Source Code Management, or simply SCM, is a set of tools and processes designed to track and manage changes to software project source code files and related artifacts. SCM goes by many names including *version control*, *configuration management*, and *revision tracking*. All SCM tools have the same primary goal, namely, to track and maintain a history of changes made

to source code files or other project artifacts placed under *change management*. Another goal of SCM is conflict resolution, meaning how changes made to the same file by two or more developers are mitigated and resolved.

The tools of choice I've selected for this book include *Git* as the source code manager and *GitHub* as the remote repository site. There are other tools and remote repository sites, but Git and GitHub have quickly become de facto industry standards. Today, 84% of Fortune 100 companies use Git for SCM and the number continues to grow. Not only is Git the SCM of choice for the commercial sector, many agencies of the U.S. Federal Government use Git for source code management as well.

1.1 SCM Architecture And Processes

Figure 8-1 offers an overview of an SCM architecture and processes.

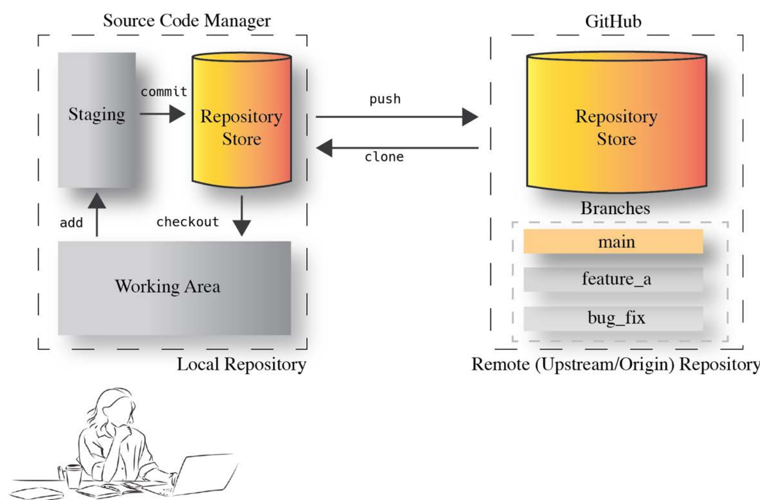

Figure 8-1: Source Code Management System Architecture and Processes

Referring to figure 8-1 — The primary architectural components of any SCM system include a remote repository and a local repository. The selected SCM tools dictate the range of operations that can be performed on local and remote repositories.

A software engineer begins their work by first making a copy of the *remote repository* on their local machine. They perform work on selected files in a working area or workspace. When finished, they add the modified files to a staging area. Next, they commit the modified files to their local repository. Finally, at some convenient point or when they deem work on an assigned task is complete, they push their changes to the remote repository.

The remote repository may have one or more *branches*. One branch is usually designated the *main* or *master* branch. Developers create additional branches as necessary to isolate related work such as new features or bug fixes. Later, when work on a branch is complete, it is *merged* with the main branch and deleted. This usually happens only after a code *peer review* gives the all clear. The exact process employed depends largely on development team dynamics, experience, and policy.

0 0 0 0 1 0 0 0

Quick Review

Source Code Management, or SCM, is a set of tools and processes designed to track and manage changes to software project source code files and related artifacts. The SCM tools selected for this chapter include Git and GitHub.

2 Git

Git is a powerful, lightweight, fast, flexible, and distributed source code management system. Its power intimidates both neophytes and experienced SCM users, but in reality, you can get a lot done with Git with only a small handful of commands following simple workflows.

2.1 Git Is Fast

Git is fast because it tracks changes to repository assets differently from other SCM systems, and most operations are performed on the local repository. You can learn more about how this is done here: *https://git-scm.com/about/small-and-fast*.

2.2 Git Is Flexible

Git is flexible because it does not lock you in to one particular workflow. Git supports multiple workflows, and individuals and teams can adopt a workflow that best suits their needs. You can learn more about the different types of Git workflows here: *https://git-scm.com/about/distributed*. The type of workflow you adopt depends on the size of the project and whether or not you work as an individual or with a team of developers. If you're a student or professional developer working independently, your workflow will be quite simple and easy to implement, and the one I'll be presenting in this chapter.

2.3 Git Is Distributed

Git is distributed in that every developer has a copy of the full repository on their local machine. Figure 8-2 illustrates this concept.

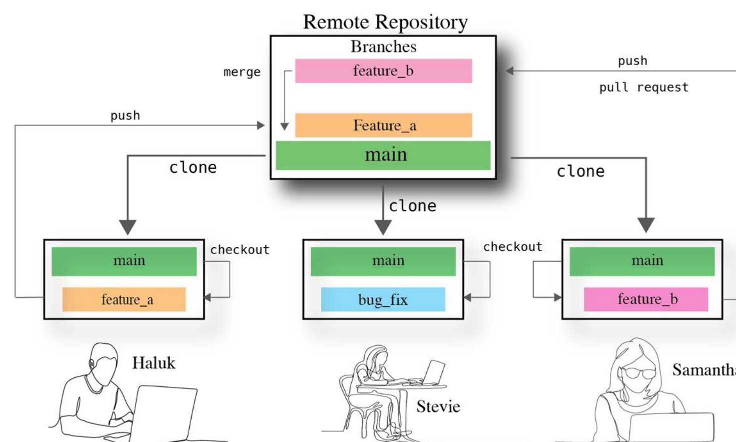

Figure 8-2: Distributed Git

© 2023 Pulp Free Press

Referring to figure 8-2 — Three developers, Haluk, Stevie, and Samantha are members of a small development team working remotely. To begin work, they each start by cloning the remote repository's `main` branch to their local machine. At this point, they each have a complete copy of the remote repository. They have agreed to work according to the following process or *workflow*. After they clone the repository, they create a new branch on their local machines using the `git checkout -b` command. They pick a branch name based on a branch naming convention. Teams usually use a work ticketing system like *JIRA* and name the branches according to ticket numbers, but in this case, I've just used the names `feature_a`, `bug_fix`, and `feature_b` for the branch names. Haluk is assigned to work on `feature_a`. He creates a branch by that name on his local machine and works on that branch, adding new files and editing existing files as necessary. All *adds* and *commits* will go against the `feature_a` branch unless he switches back to the `main` branch. When he's ready, he pushes his branch to the remote repository with the `git push` command.

Samantha is working on `feature_b` and creates a branch by that name on her local machine. When she completes her work, she too pushes her changes to the remote repository with the `git push` command. She also submits a *pull request*, and after her code has been *peer reviewed* by her teammates, the `feature_b` branch is merged with the `main` branch. At this point, Samantha can delete the `feature_b` branch from both her local and remote repositories as it's no longer required.

If another developer's changes are merged with the `main` branch then developers will need to update their local copy of the main branch with a `git pull` command before starting work on a new branch. This cycle continues as new work is assigned, performed, and completed.

Note that Stevie has yet to push her `big_fix` branch to the remote repository, which is why it is currently located only on her local machine.

2.4 Local Repository Organization

Figure 8-3 shows how a local repository is organized.

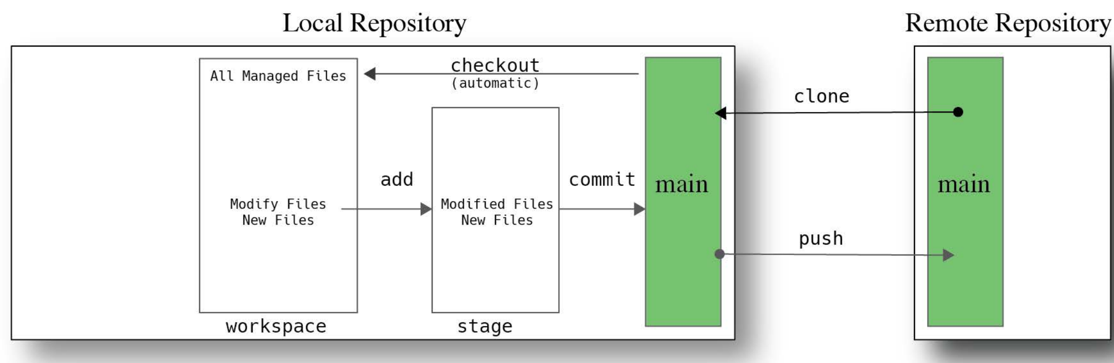

Figure 8-3: Local Repository Organization

Referring to figure 8-3 — When a developer clones the `main` branch with the `git clone` command, its contents is automatically checked out and made available in the local *workspace*. To you, the developer, this looks just line an ordinary directory, but the files in that directory are being tracked by Git, this includes edits to existing files along with new and deleted files. Modifications, additions, and deletions are then *staged* with the `git add` command. At some point the staged files are committed to the local repository with the `git commit` command. The cycle of

working on files, adding, and committing can be repeated as often as required. At some point, usually when work is complete, the changes made to the local repository are pushed to the remote repository. In this case, the changes made to the local copy of the `main` branch are pushed to remote copy of the `main` branch.

This simple workflow is sufficient for individual developers as there is little chance of introducing conflicting changes to the remote repository when only one developer is working on a project.

2.5 Checking Out A New Local Branch

A more complex workflow used by both individuals and teams is shown in figure 8-4.

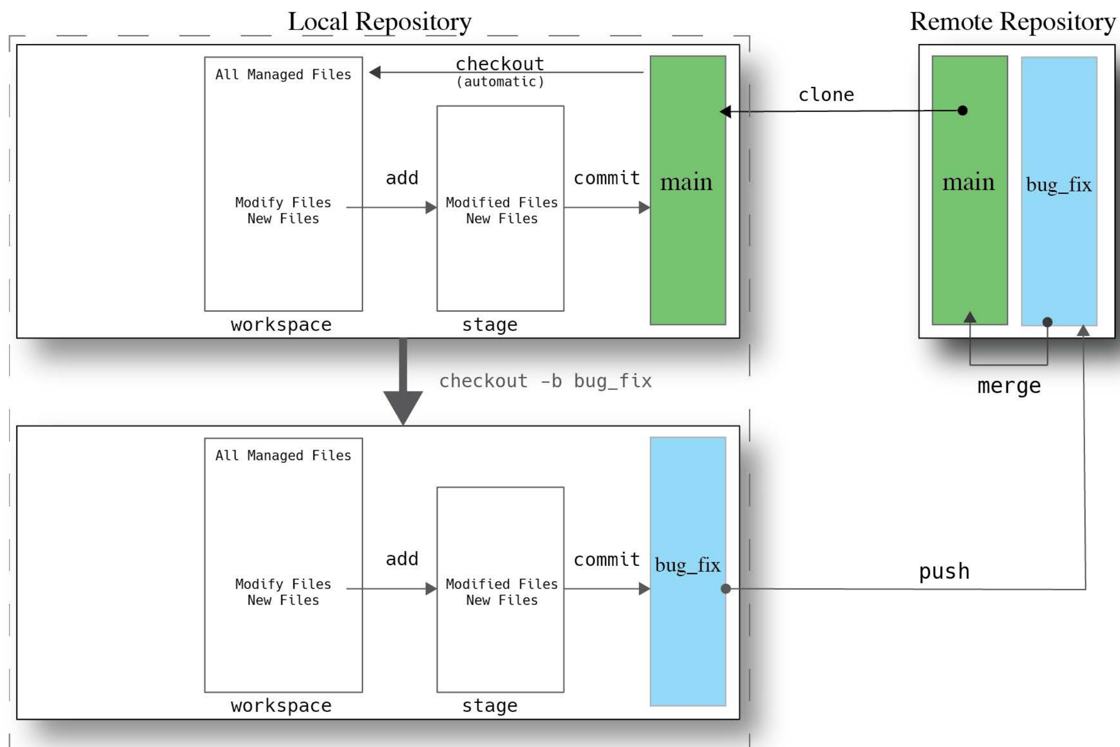

Figure 8-4: Checkout New Local Branch

Referring to figure 8-4 — Work starts as usual by first using `git clone` to clone the `main` repository branch. The Musketeer's agreement between team members is they do not work on the `main` branch. Instead, they checkout a new branch based on some naming scheme using the `git checkout -b` command. In this case, a developer is working on a bug fix and so created a new branch on their local machine named `bug_fix`. A new branch is a complete copy of the repository. In other words, the `bug_fix` branch is a complete copy of the `main` branch. The developer then modifies files, adds new files, or deletes files as required. All such modifications are tracked by Git in the new branch's workspace. When ready, the developer can stage their modifications with the `git add` command, and commit changes to the local `bug_fix` branch with the `git commit` command. When work is complete, the developer pushes the `bug_fix` branch to the remote repository with the `git push` command. At some point, the developer will submit a *pull request*, and when their code has been peer reviewed, the bug_fix branch is *merged* with the `main` branch

and then deleted if no longer required. At this point, other developers would need to do a `git pull` to update their local `main` branches.

2.6 Practice Makes Perfect

As with any new skill, programming or otherwise, you will not get proficient at Git by just looking at a few diagrams and reading a narrative. You'll need to dive in, create a remote repository, clone it, add some files, make some edits, commit the changes, and push those changes to the report repository. You'll want to write these commands down along with the workflow steps you decide to adopt in your Engineer's Notebook.

Quick Review

Git is a powerful, lightweight, fast, flexible, and distributed source code management system. Its power intimidates both neophytes and experienced SCM users, but in reality, you can get a lot done with Git with only a small handful of commands.

Work generally starts with cloning a remote repository. There's usually one branch in the remote repository designated `main` or `master`. Cloning a remote repository makes a complete copy of the repository on your local machine and does an automatic checkout of tracked files into your workspace, ready for editing. As work progresses, you edit files, add new files, or remove files as required. Stage changes with the `git add` command. Save changes to your local repository with the `git commit` command, and push local repository changes to the remote repository with the `git push` command.

3 Configure SSH Keys For GitHub

Before you can push changes from your local repository to your remote repository you'll need to create and configure Secure Shell (SSH) keys to ensure you have a secure connection between your local machine and GitHub.

Generating and configuring SSH keys is a fairly simple process, but if you have little to no experience using terminal commands, that's where you'll run into problems. If you follow the steps outlined in this section you will succeed! If you'd rather watch a video, I have created one just for you: *https://youtu.be/icxmJDmQ0GI* These steps work in Windows, Linux, and macOS, but I am making the following assumptions.

3.1 Assumptions

- You have verified Git is installed and working
- Windows users are using the Git Bash terminal
- You have a GitHub account

3.2 Preconditions

The process starts in your home directory. Launch a terminal and, if not already there, change to your home directory `'~'` by executing the following command:

© 2023 Pulp Free Press

```
cd
```

This will change to your home directory. Verify you are there before proceeding.

3.3 Process Overview

The rest of this section guides you step-by-step through the following process:

- Verify the existence of, or create, the *~/.ssh* directory
- Create a *~/tmp* directory in which to practice SSH key generation
- Generate public and private SSH keys with a passphrase
- Copy the public and private SSH keys to the *~/.ssh* directory
- Add the public key to your GitHub account
- Add the private key to your local machine's SSH Agent
- Test your SSH key

3.4 Verify or Create .SSH Directory

List the contents of your home directory including hidden files and folders with the following command:

```
ls -al
```

If you see the *~/.ssh* directory listed, you'll need to proceed with caution because you may already have SSH keys in that directory. To check, list the contents of the *~/.ssh* directory with the following command:

```
ls -al .ssh
```

If the *~/.ssh* directory is not empty, note the names of the files. You may have a *config* and/or *known_hosts* file along with other files that may be public and private key files. Public key files have a *.pub* file extension. You'll simply need to ensure the names of the keys you generate in this section have a different name from the ones already in the *~/.ssh* directory, that's all.

If the *~/.ssh* directory does not exist, create it with the following command:

```
mkdir .ssh
```

Verify it was created by listing the contents of your home directory. Note that the '.' is important in the *~/.ssh* directory name because it's a hidden directory, and that's the directory in which your computer system expects to find SSH keys and configuration files.

3.5 Create tmp Directory

You'll want to create a *~/tmp* directory in which to practice SSH key generation until you get the swing of it and get the keys you want. If the *~/tmp* directory doesn't already exist, create it with the following command:

```
mkdir tmp
```

Now, change to the *~/tmp* directory with the following command:

```
cd tmp
```

Your prompt should now show you're in the *~/tmp* directory. It is in here that you're going to practice generating SSH keys until you completely understand the process.

3.6 Generate SSH Keys

Before proceeding, verify you're in the *~/tmp* directory. Also, before generating keys, you need to understand what the key generation command actually does by default so you don't think you're going crazy. I don't recommend using the default values. Instead, you should take control, figure out how the command works, and save yourself a lot of grief. This leads me to post the following Pro Tip.

Pro Tip: Be Careful! When you generate a set of SSH keys, the default output location is the ~/.ssh directory. That's OK if there are no keys in that directory, but you should explicitly specify a key name when you run the ssh-keygen tool to ensure the keys are generated in the ~/tmp directory.

To generate SSH keys you'll use the following command:

```
ssh-keygen -t ed25519 -C "your_github_email@example.com"
```

The `ssh-keygen` command has a lot of options and a lot of uses, but the `-t` and `-C` options shown above are all you need to generate keys for use with Git and GitHub. Table 8-1 explains the purpose of each command option.

Section	Description
`ssh-keygen`	Command that generates SSH keys
`-t ed25519`	`-t` Specifies the type of key generation algorithm. In this case, it's specifying the ed25519 algorithm. If ed25519 is not supported on your system use `-t rsa`.
`-C "your_github_email@example.com"`	`-C` (dash Capital `'C'`) Add a comment. Use your GitHub account email.

Table 8-1: ssh-keygen Command

Referring to table 8-1 — The `-t` switch specifies the cryptographic algorithm used to generate the keys. Older systems may not support the `ed25519` algorithm. In that case use `-t rsa` to generate an RSA key. If your computer's operating system is relatively new then `ed25519` will work just fine. The `-C` switch specifies a comment. Use your GitHub account email.

OK, when ready, execute the `ssh-keygen` command. I will use *my* GitHub account email.

```
ssh-keygen -t ed25519 -C "rick@pulpfreepress.com"
```

Your terminal output at this point will look similar to figure 8-5.

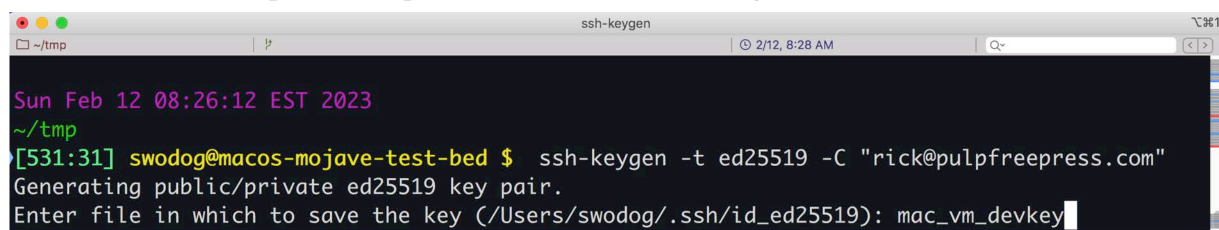

Figure 8-5: Generating SSH Keys with `ssh-keygen` Tool — Custom Key Name

Referring to figure 8-5 — Notice that on the last line the tool indicates that if you don't enter a custom key name, it will generate the public and private keys with the names `id_ed25519` and `id_ed25519.pub` to the *~/.ssh* directory. I'm creating keys named `mac_vm_devkey` and `mac_vm_devkey.pub`, which will be saved in the current working directory, which in this case is *~/tmp*. I recommend you use a custom key name as well. You can have as many SSH keys as required, usually one for each machine you use for coding or interacting with GitHub.

OK, when you're ready, hit return. You'll be prompted to enter a *passphrase*. I strongly recommend using a passphrase because it makes your private key more secure. Be sure not to forget your passphrase. If you do, you'll need to generate new SSH keys. Figure 8-6 shows my terminal after I've entered my passphrase and listed the contents of the *~/tmp* directory.

```
Sun Feb 12 08:26:12 EST 2023
~/tmp
[531:31] swodog@macos-mojave-test-bed $  ssh-keygen -t ed25519 -C "rick@pulpfreepress.com"
Generating public/private ed25519 key pair.
Enter file in which to save the key (/Users/swodog/.ssh/id_ed25519): mac_vm_devkey
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in mac_vm_devkey.
Your public key has been saved in mac_vm_devkey.pub.
The key fingerprint is:
SHA256:EzvGWcIiMkXJ2Fr9tlEqXzuySP6jQz3MFEkgqINsY2c rick@pulpfreepress.com
The key's randomart image is:
+--[ED25519 256]--+
|   *+o.o..       |
|  o.=...o .      |
|o.oo. ..++.      |
|+=.E ..o**.      |
|o.+    OS+ .     |
|      o.Ooo      |
|     + . + .     |
|      + o        |
|      .+..       |
+----[SHA256]-----+

Sun Feb 12 08:39:34 EST 2023
~/tmp
[532:32] swodog@macos-mojave-test-bed $  ls -al
total 16
drwxr-xr-x   4 swodog  staff  128 Feb 12 08:39 .
drwxr-xr-x+ 28 swodog  staff  896 Feb 12 08:05 ..
-rw-------   1 swodog  staff  464 Feb 12 08:39 mac_vm_devkey
-rw-r--r--   1 swodog  staff  104 Feb 12 08:39 mac_vm_devkey.pub
```

Figure 8-6: SSH Key Generation Complete — *~/tmp* Directory Listed

Referring to figure 8-6 — Your output will look slightly different. You'll have a different email address, different SSH key names, a different key fingerprint, and a different randomart image. Note that the public key is the file that ends with the `.pub` suffix. Just keep that in mind as you proceed with the next few steps. Almost done! Hang in there!

3.7 Copy Keys To .ssh Directory

If you're happy with your SSH keys you can copy them to the *~/.ssh* directory. To do that from the *~/tmp* directory use the following command:

```
cp * ~/.ssh
```

This will copy everything in the *~/tmp* directory to the *~/.ssh* directory. Verify your SSH keys are now in the *~/.ssh* directory.

3.8 Add Public Key To GitHub

To use your new SSH keys with GitHub, you'll first need to add the public (`.pub`) key to GitHub's list of SSH keys. Login to GitHub, click on your profile picture, and from the dropdown menu select **Settings**. In the left-hand column select **SSH and GPG keys**. This will open the SSH and GPG keys page as shown in figure 8-7.

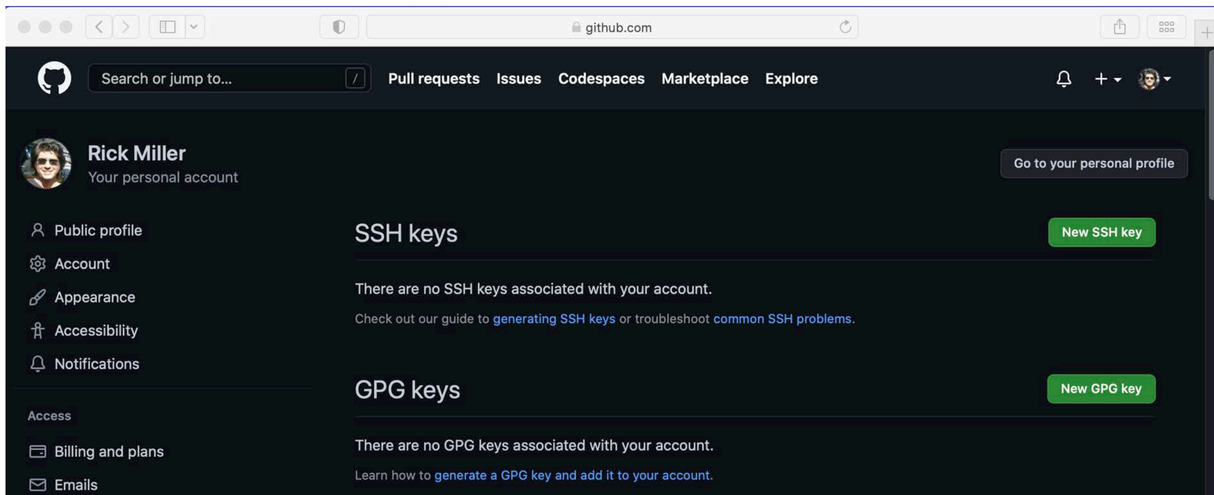

Figure 8-7: GitHub SSH & GPG Keys Page — No Keys Added

Referring to figure 8-7 — If this is your first rodeo then you'll have no SSH keys listed. See the **New SSH key** button? I'll come back to that in a bit, but first, you'll need to copy the contents of your public SSH key. To do this, navigate to the *~/.ssh* directory if you're not already there, and type the following command...

```
cat keyname.pub
```

...where `keyname` is the name of *your* SSH key. Your output will look similar to Figure 8-8.

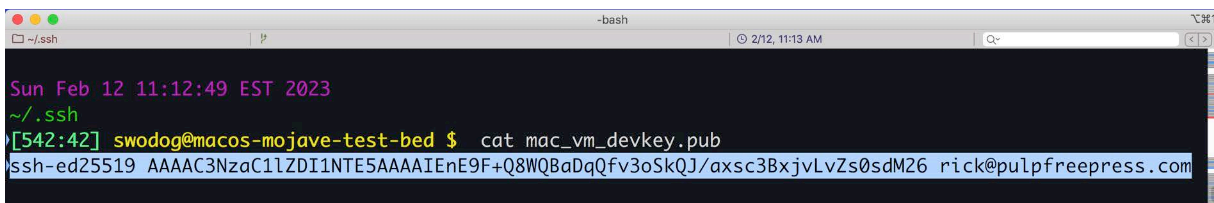

Figure 8-8: List SSH Public Key Contents to the Console with the `cat` Command

Referring to figure 8-8 — Extend your terminal window's width to see the whole public key string on one line as shown above. Next, select the entire string as shown, then right-click and copy the string. This method works on all three operating systems: macOS, Windows (Git Bash), and Linux. Next, return to the GitHub SSH and GPG keys page, and click the **New SSH key** button to launch the SSH keys/Add new page as shown in figure 8-9.

Referring to figure 8-9 — Enter a title for your key in the **Title** textbox. Leave the **Key type** dropdown set to *Authentication Key*, and in the **Key** textbox, right-click and paste your public key string as shown in figure 8-10.

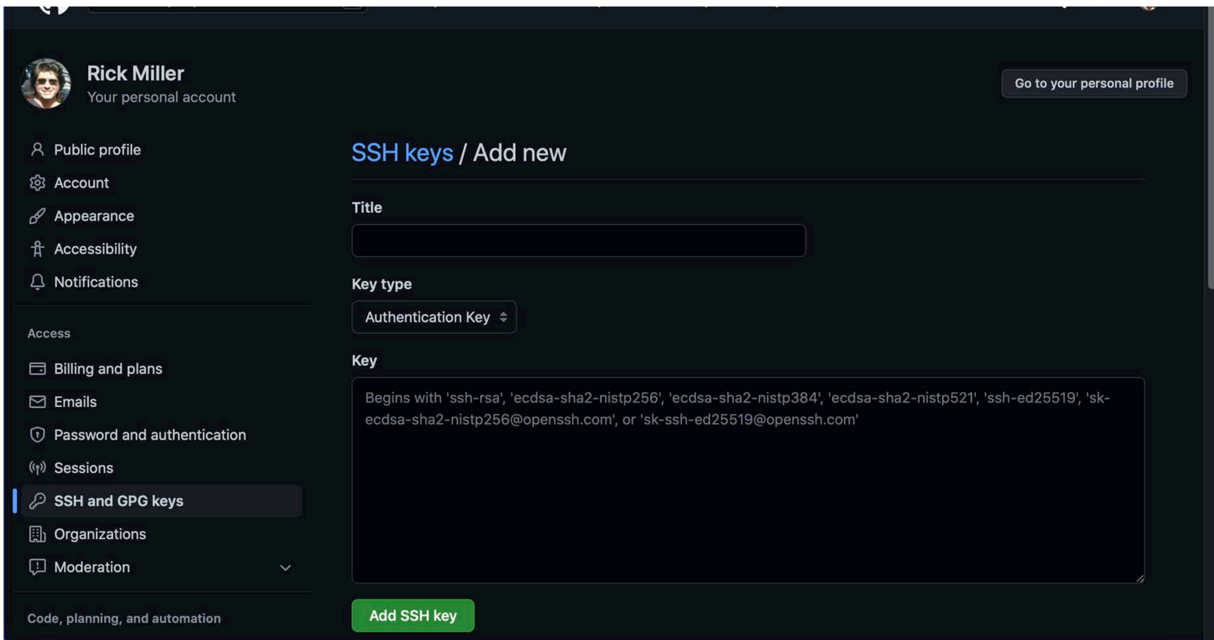

Figure 8-9: GitHub SSH keys/Add new Page

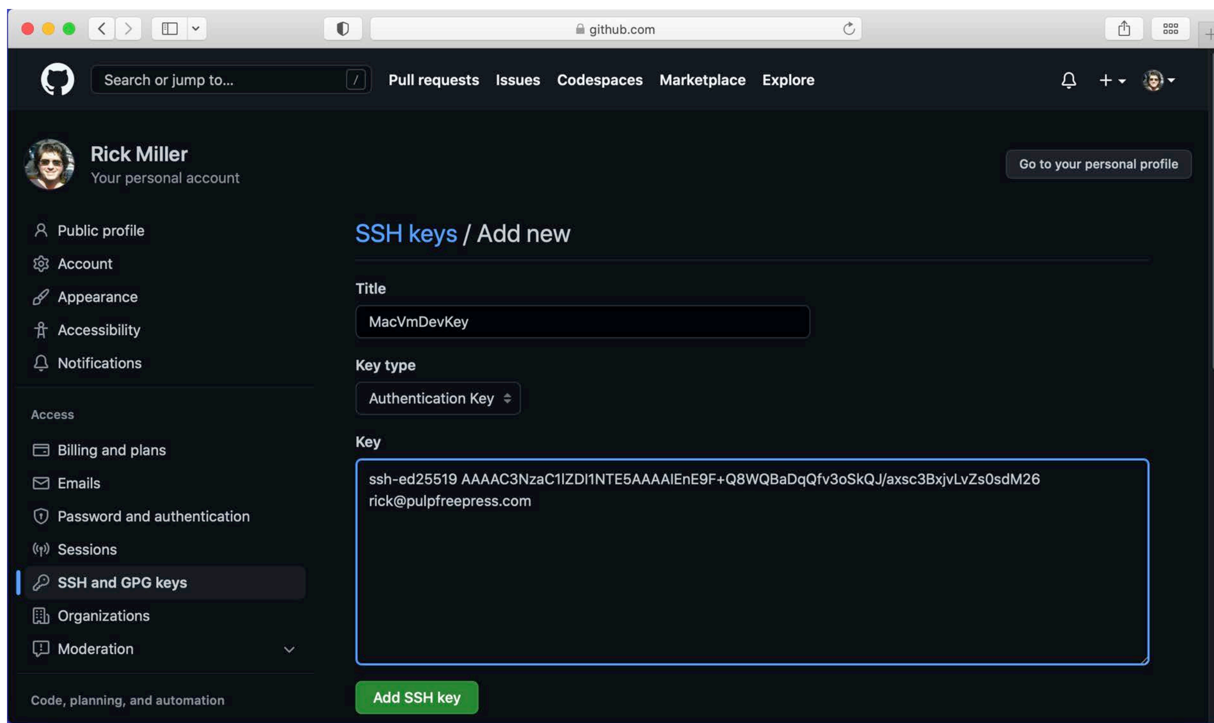

Figure 8-10: New SSH Key Details Filled In

Referring to figure 8-10 — Make sure everything is good-to-go, then click the **Add SSK key** button. You should now see your new SSH key listed on the SSH and GPG keys page as shown in figure 8-11.

Referring to figure 8-11 — If everything looks good, you're done with GitHub — for now. If you made a mistake along the way, don't sweat it! Repeat the process a few times and you'll get the hang of it. SSH key generation really is one of those things you need to figure out and then once you do it you won't do it again for so long you'll forget how you did it. Well, a lot of soft-

© 2023 Pulp Free Press

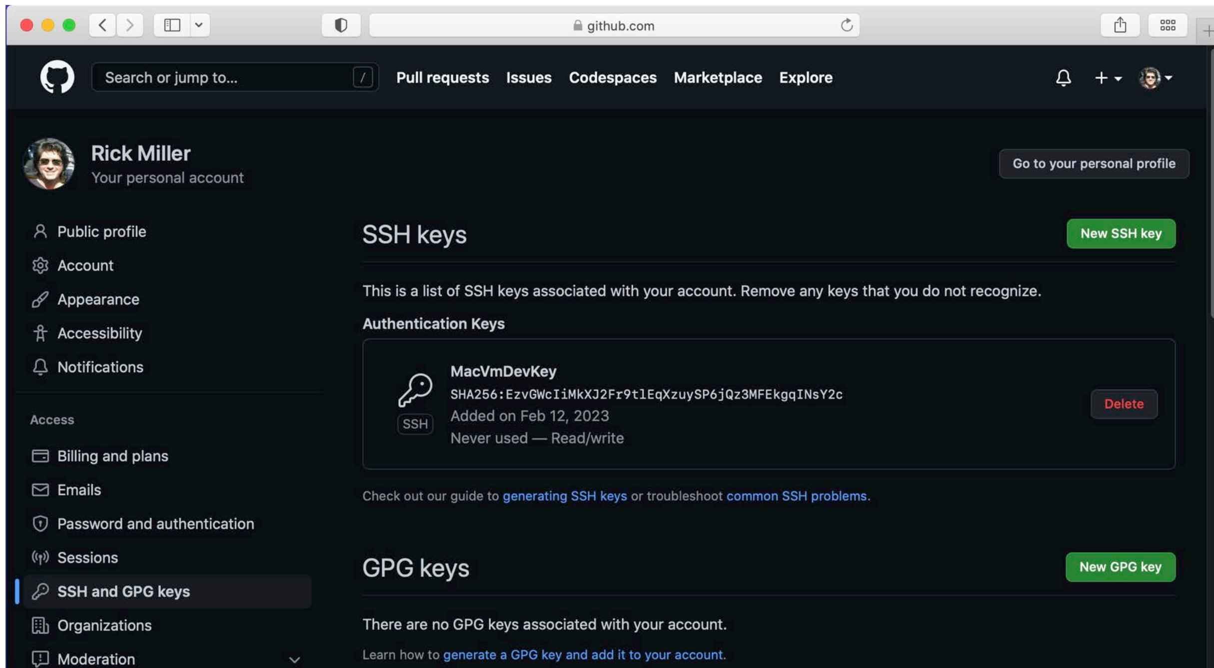

Figure 8-11: Your New SSH Key Should Now Be Listed

ware engineering is like that. Write it down in your Engineer's Notebook! Now, you need to test your new SSH keys.

3.9 Start SSH Agent on Local Machine

Start the SSH agent on your local machine with the following command:

```
eval "$(ssh-agent -s)"
```

The result of running this command will be a process ID (pid) similar to what's shown in figure 8-12.

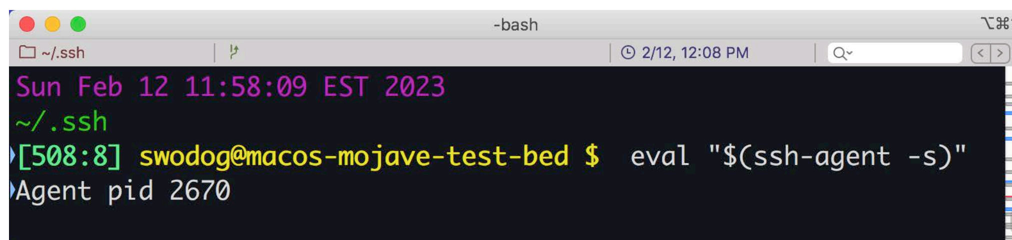

Figure 8-12: Starting SSH Agent on Local Machine

Referring to figure 8-12 — Your Agent pid value will be different from what's shown above and that's fine. You can now add your private key to your SSH agent.

3.10 Add Private Key To SSH Agent

Before you can use your new SSH keys with GitHub, you'll need to add the private key to your local machine's SSH agent with the following command...

```
ssh-add keyname
```

...where `keyname` is the name of *your* private SSH key. When prompted, enter your private key passphrase. Your session will look similar to figure 8-13.

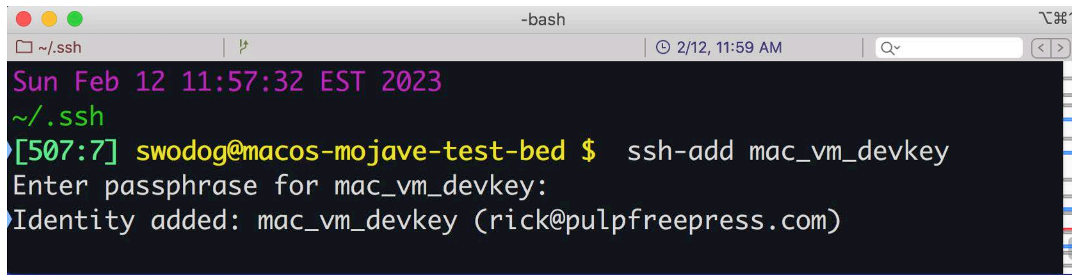

Figure 8-13: Add Private SSH Key to SSH Agent with `add-ssh` Command

Referring to figure 8-13 — If you don't get the `Identity added` message, ensure you started the SSH agent and make sure your passphrase is correct. If all goes well (*The mating call of a coding boot camp graduate!*), you can proceed to test your SSH key.

3.11 Test SSH Keys

Alright, a quick audit: You've generated your SSH keys, added the public key to GitHub, and added your private key to your computer's SSH agent. — Test your SSH keys with the following command:

```
ssh -T git@github.com
```

The output should look similar to figure 8-14.

Figure 8-14: Testing SSH Keys with `ssh -T` Command

Referring to figure 8-14 — When you execute the test, you'll be prompted with the following message: `"Are you sure you want to continue connecting (yes/no/[fingerprint])?"` Type `yes` and hit return. You should see the message `"...You've successfully authenticated..."` If instead it reads `"...access denied..."`, then ensure you added your SSH key to the local machine's SSH agent.

When the test successfully completes, you'll have a new file named *known_hosts* in your *~/.ssh* directory as shown in figure 8-15.

Referring to figure 8-15 —I'll leave it as an exercise to explore the contents of the *known_hosts* file. OK, that's it for SSH key generation. Like I said earlier, you can generate as many SSH keys as required. If you use both a desktop machine and a laptop to code, create a different set of SSH keys for each machine.

```
-bash
~/.ssh                                   2/13, 9:04 AM
Mon Feb 13 09:02:35 EST 2023
~/.ssh
[503:3] swodog@macos-mojave-test-bed $  ls -al
total 24
drwxr-xr-x   5 swodog  staff  160 Feb 12 12:30 .
drwxr-xr-x+ 28 swodog  staff  896 Feb 13 08:48 ..
-rw-r--r--   1 swodog  staff  185 Feb 12 12:30 known_hosts
-rw-------   1 swodog  staff  464 Feb 12 12:28 mac_vm_devkey
-rw-r--r--   1 swodog  staff  104 Feb 12 12:28 mac_vm_devkey.pub
```

Figure 8-15: *~/.ssh* Directory Listing Showing *known_hosts* File

3.12 Load SSH Private Key Into SSH Agent Automatically

To avoid having to manually load your SSH private key into the SSH agent every time you launch a terminal, you can configure it to be automatically loaded. All three operating systems run OpenSSH, but macOS has the KeyChain application, which stores your private key password, so its SSH configuration is slightly different. I'll start with macOS.

3.12.1 macOS

For macOS, in your *~/.ssh* directory, create a file named *config* and add the lines shown in example 8.1.

8.1 ~/.ssh/config (macOS)

```
1    Host *
2        AddKeysToAgent yes
3        UseKeychain yes
4        IdentityFile ~/.ssh/private_key
```

Referring to example 8.1 — Replace `private_key` with the name of *your* private key. On line 3, the `UseKeyChain` option stores your private key password in the macOS KeyChain application which eliminates the need to reenter your password every time you launch a terminal.

3.12.2 Linux Mint

Linux Mint users just need to add a *config* file in their *~/.ssh* directory with the lines shown in example 8.2

8.2 ~/.ssh/config (Linux Mint)

```
1    Host *
2        AddKeysToAgent yes
3        IdentityFile ~/.ssh/private_key
```

Referring to example 8.2 — Replace `private_key` with the name of *your* private key. If you have a passphrase, you'll need to enter it the first time you launch a terminal.

3.12.3 Windows Git Bash

Getting the desired behavior on Windows with Git Bash takes some finesse. I found the following solution *here*. First, create a *config* file in your *~/.ssh* directory and add the lines shown in example 8.3

8.3 ~/.ssh/config (Windows Git Bash)

```
1    Host *
2       AddKeysToAgent yes
3       IdentityFile ~/.ssh/private_key
```

Referring to example 8.3 — Replace `private_key` with the name of *your* private key. Next, create a *~/.bashrc* file and add the lines shown in example 8.4.

8.4 ~/.bashrc (Windows Git Bash)

```
1    # Start SSH Agent
2    #----------------------------
3
4    SSH_ENV="$HOME/.ssh/environment"
5
6    function run_ssh_env {
7      . "${SSH_ENV}" > /dev/null
8    }
9
10   function start_ssh_agent {
11     echo "Initializing new SSH agent..."
12     ssh-agent | sed 's/^echo/#echo/' > "${SSH_ENV}"
13     echo "succeeded"
14     chmod 600 "${SSH_ENV}"
15
16     run_ssh_env;
17
18     ssh-add ~/.ssh/private_key;
19   }
20
21   if [ -f "${SSH_ENV}" ]; then
22     run_ssh_env;
23     ps -ef | grep ${SSH_AGENT_PID} | grep ssh-agent$ > /dev/null || {
24       start_ssh_agent;
25     }
26   else
27     start_ssh_agent;
28   fi
29
```

Referring to example 8.4 — On line 18, replace `private_key` with *your* private key. Finally, edit your *~/.bash_profile* and **add** the lines shown in example 8.5.

8.5 ~/.bash_profile (Windows Git Bash)

```
1    # Add these lines
2    test -f ~/.profile && . ~/.profile
3    test -f ~/.bashrc && . ~/.bashrc
```

Save the file, close all terminal windows, sign out of Windows, and sign in again. Launch a Git Bash terminal, and when prompted enter your passphrase. You should be good-to-go.

 © 2023 Pulp Free Press

3.13 SSH Key Generation Command Summary

Table 8-2 summarizes the commands used in this section to generate and test SSH keys for use with GitHub.

Command	Description
`ssh-keygen -t ed25519 -C "your_github_e-mail@example.com"`	Generate an SSH key with the ed25519 algorithm. This results in a private and public key. The public key ends with the `.pub` file suffix.
`eval "$(ssh-agent -s)"`	Starts the SSH agent. You need to start the SSH agent before adding your private key.
`ssh-add keyname`	Add the private key to the SSH agent
`ssh -T git@github`	Test the SSH connection to GitHub. This test will pass if your public key is added to GitHub and your private key is added to the SSH agent.

Table 8-2: SSH Key Generation Command Summary

Quick Review

GitHub requires the use of SSH keys to enable secure communications between local and remote repositories. SSH keys and configuration files are located in the *~/.ssh* directory. Use a temporary directory, *~/tmp*, to practice SSH key generation until you're confident you're generating the SSH keys you want. You can then copy the public and private keys to the *~/.ssh* directory when you're ready to use them. To use your SSH keys with GitHub, you'll need to add the public SSH key to GitHub and add the private key to your computer's SSH agent.

4 Create GitHub Repository

In this section, I walk you through the creation of a GitHub repository and discuss some things you should consider before you create the repository. I'll also discuss the purpose of the `README.md` and `.gitignore` files as they are important repository files.

4.1 Repository Organization Considerations

You should pause to consider how you'll use a repository before you create one. If you are a student, you may need to create a repository to house class projects. If this is the case, then you can create one repository with multiple subdirectories as shown in figure 8-16

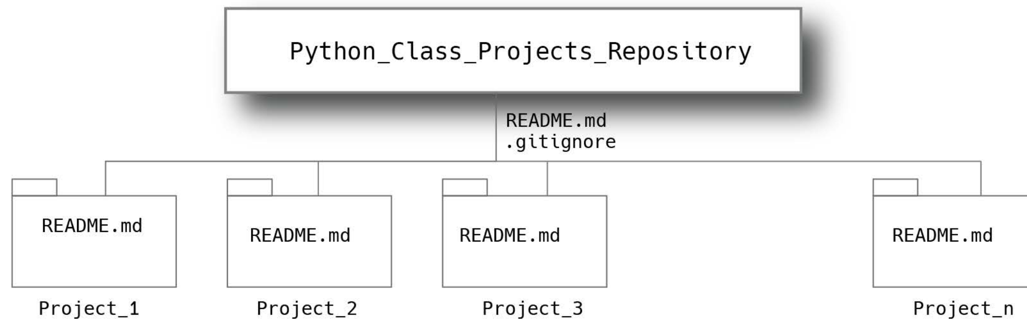

Figure 8-16: Repository with Multiple Project Directories

Referring to figure 8-16 — This repository structure is ideal as each class project is contained within it's own directory. You can have one or more *README.md* files, one at the root of the repository to provide an overview, and one in each project directory to document individual projects. The same goes for the *.gitignore* files, but I'm only showing one at the root of the repository above. Each project folder contains source code and other project artifacts as required. I'll discuss project structure in more detail in *Chapter 9: Project Structure*.

Essentially, a repository is a directory with a *.git* file at its root. There are effective directory structures for small projects and effective directory structures for larger projects. The Git workflows employed with each type of project will differ based on project complexity and the size of the development team. The repository structure shown in figure 8-16 is ideal for the student or individual contributor working on small to medium-sized projects.

If your project is large and complex, you may want to create a dedicated repository just for that project. In that case, subdirectories will impose organization upon the different parts of the project, either by architectural layer, like the user interface (UI) or web layer, mid-tier, back-end, and cloud infrastructure. For the purposes of this book, the repository structure shown in figure 8-16 will serve you well.

4.2 Create Python Class Projects Repository

There are several ways to create a repository on GitHub. You can initialize a Git repository on your local machine and push it to GitHub, you can copy an existing repository, or, and my preferred way of doing business, start by creating the remote repository on GitHub and clone it to your local machine. That's the easiest way for beginners to get a repository up and running.

 © 2023 Pulp Free Press

4.3 Create New GitHub Repository

Login to GitHub and navigate to your Repositories page. Click on the **New** button in the upper right-hand corner. This will open the Create a New Repository page as shown in figure 8-17.

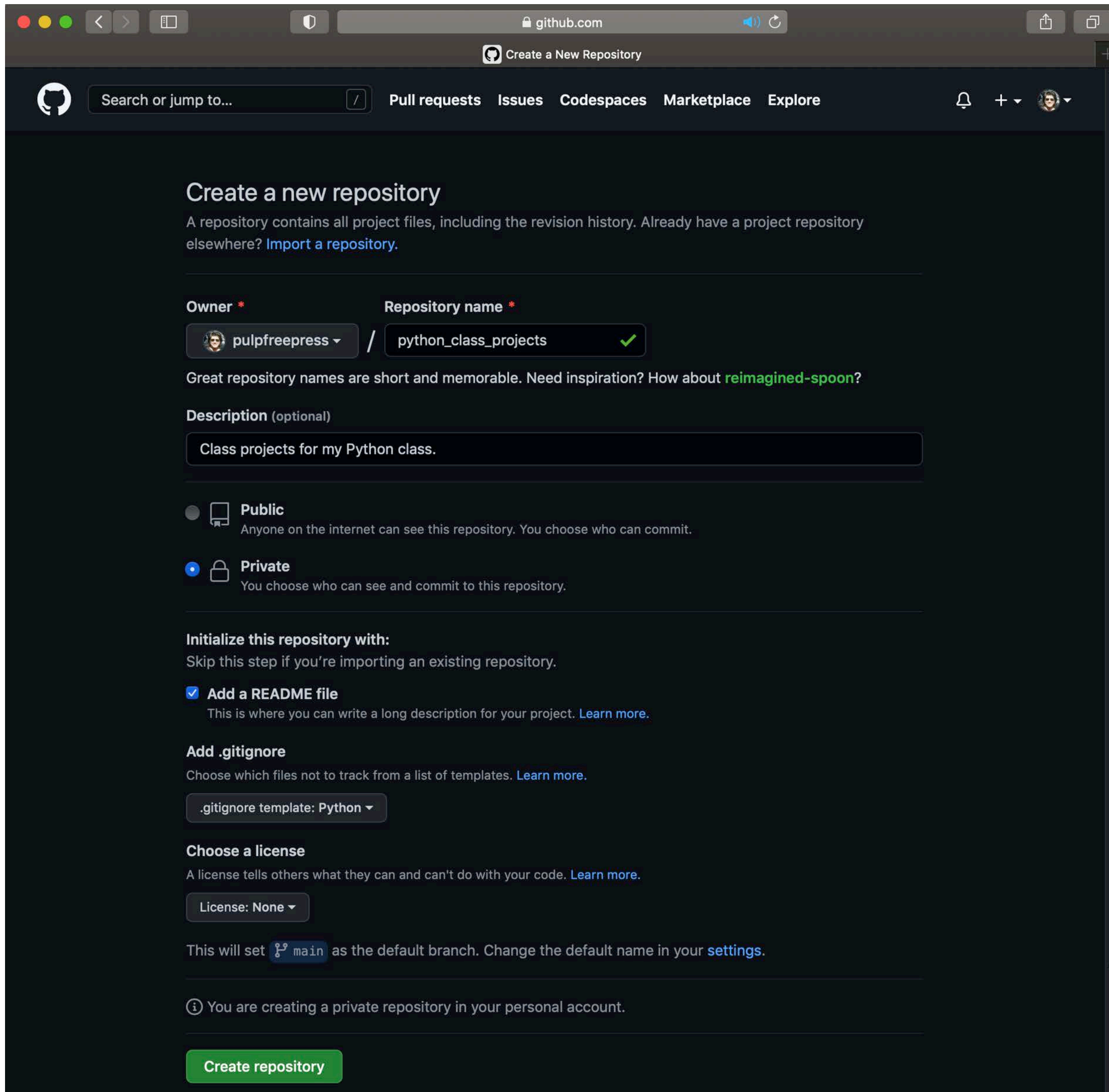

Figure 8-17: Create a New Repository Page — GitHub

Referring to figure 8-17 — Starting at the top, the **Owner** dropdown is set to your GitHub account. In the **Repository name** field type in the name of your repository. In this example, I'm naming the repository `python_class_projects`. Enter a description. Click either the **Public** or **Private** radio button. Just something to think about, if you make your class projects repository public then your classmates may sneak a peek. You may want to keep your repository private until the semester ends. In the **Initialize this repository with** section click the **Add a README.md file** checkbox, and from the **Add .gitignore** dropdown select `.gitignore template: Python`. Leave the **Choose a license** dropdown set to `none`. Give everything a good once-over then click

the **Create repository** button. This will take you to your new repository's page as shown in figure 8-18.

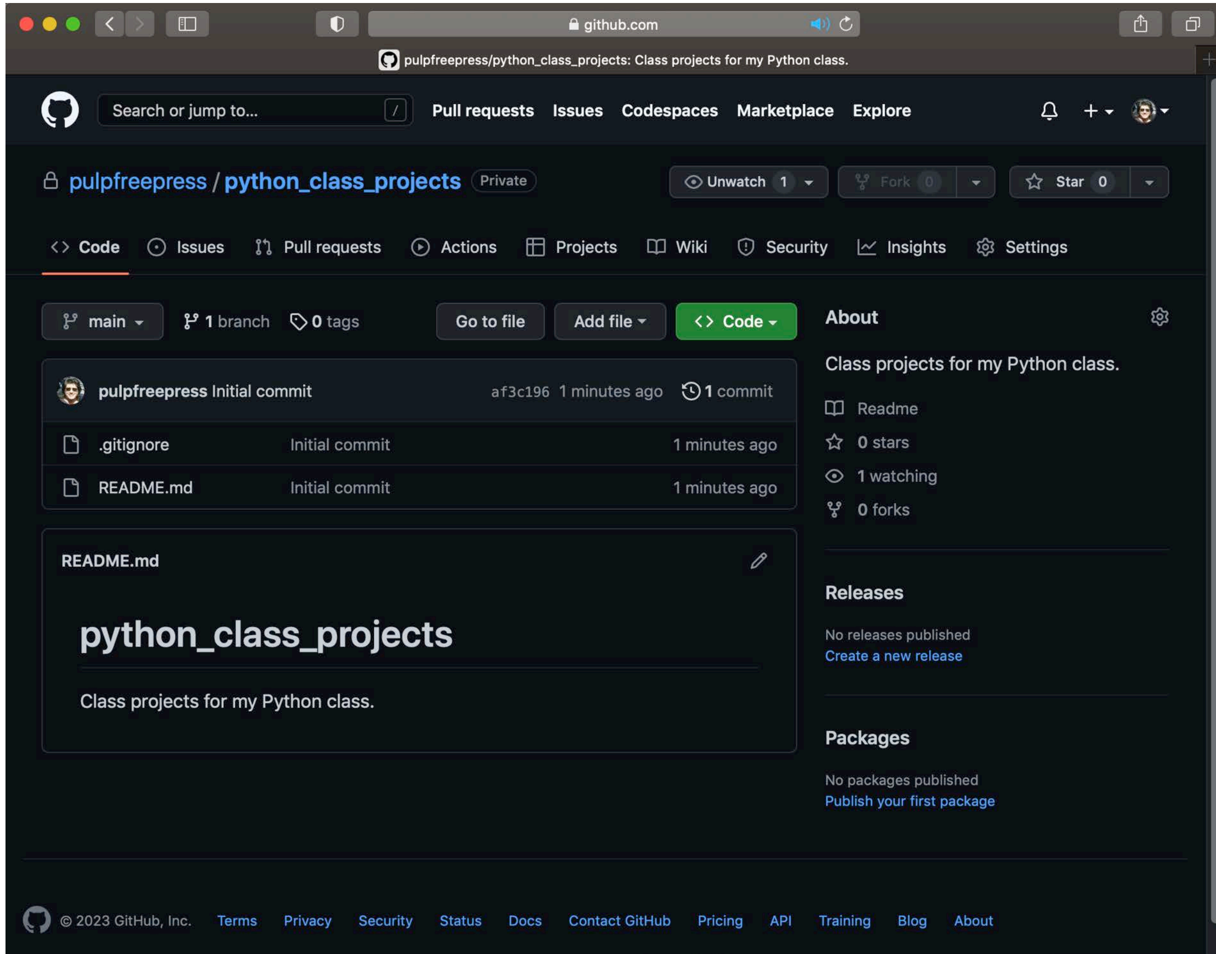

Figure 8-18: New python_class_projects Repository Page

Referring to figure 8-18 — Your new repository is up and running and populated with *README.md* and *.gitignore* files. You can now clone this repository to your local machine.

4.4 Clone The Repository With SSH

Still on your new repository page, click the <> **Code** dropdown, select SSH, and click the copy button to the right of the repository URL as shown in figure 8-19.

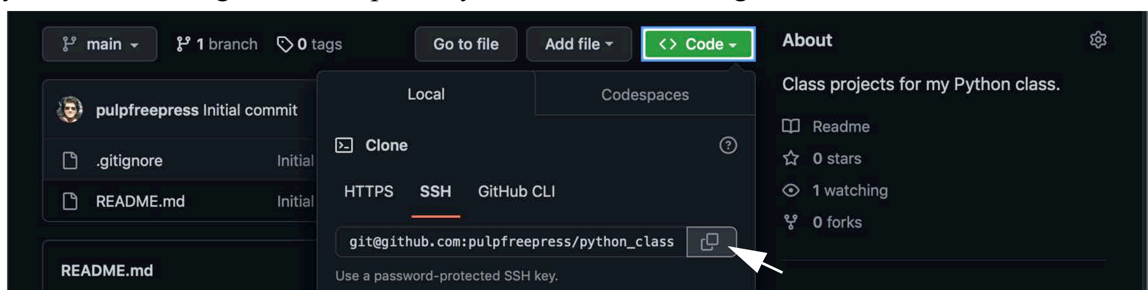

Figure 8-19: Copy Repository SSH URL

Referring to figure 8-19 — The GitHub SSH repository URL has the following format:

```
git@github.com:account/repository_name.git
```

Your repository SSH URL will be different from the one shown in figure 8-19. Even if you named your repository the same name I used in this section, your account name will be different. Just something to keep in mind.

OK, once you've copied the repository SSH URL, navigate to your *~/dev* folder and clone the repository with the following command:

```
git clone [paste SSH URL here]
```

So, when I clone the repository this is what my complete command looks like:

```
git clone git@github.com:pulpfreepress/python_class_projects.git
```

When I execute this command on my local machine, I see the output shown in figure 8-20.

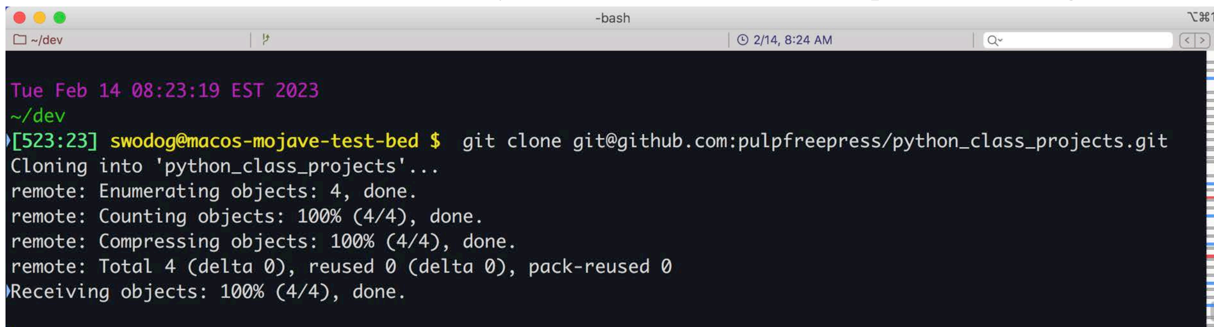

Figure 8-20: Results of Cloning The Repository

Next, list the contents of your *~/dev* directory. You should see a subdirectory with the same name as the repository you just cloned as shown in figure 8-21.

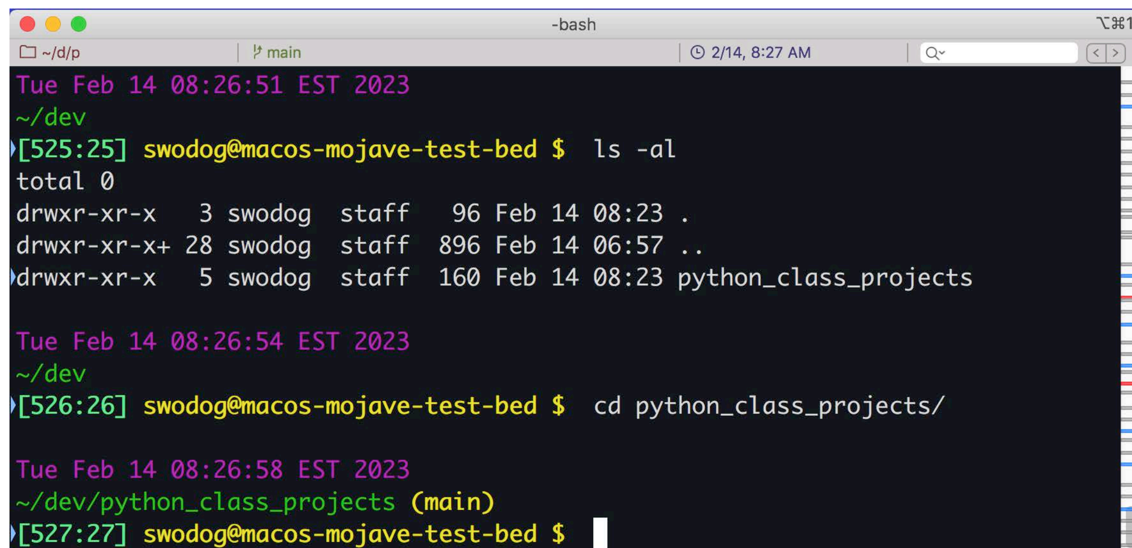

Figure 8-21: After Cloning a Repository — main Branch

Referring to figure 8-21 — Since I just cloned a repository named *python_class_projects*, I see a subdirectory by that name in my *~/dev* directory. Change into your repository subdirectory. If your prompt is configured to do so, you should see an indication you are on the `main` branch. If you are using iTerm on macOS and have installed and configured Shell Integration and Status Bar

widgets, (See “iTerm2 Terminal Configuration” on page 24.), you’ll see the Git branch just below the terminal window’s title bar, if you in fact added that Status Bar widget. If you do not see the branch indication in your terminal prompt, the next section explains how to configure it.

Before moving on, list the contents of your local repository. You should see the same files in the local repository directory as you see in the remote repository on GitHub with one additional hidden directory named *.git* as shown in figure 8-22.

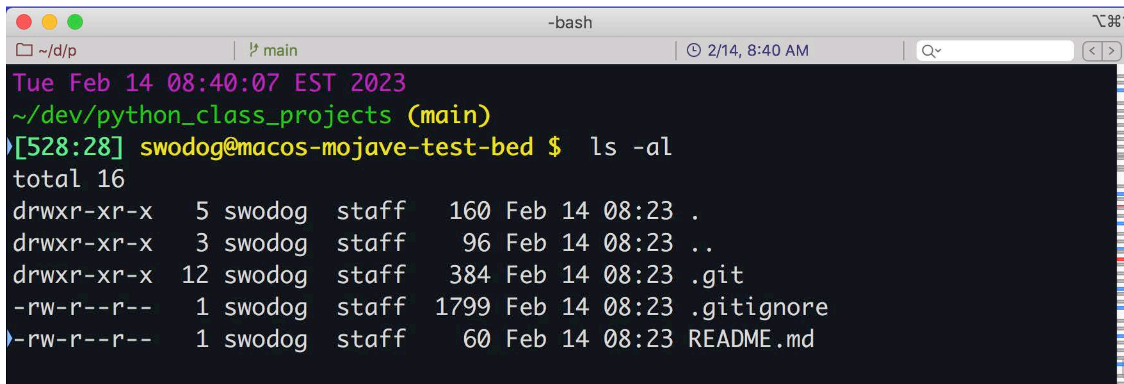

Figure 8-22: Contents of Newly-Cloned Repository

Referring to figure 8-22 — The *.git* directory stores repository configuration information and is where Git keeps track of changes made to local repository files. You generally have no need to modify anything in the *.git* directory, although rooting around in there can be quite educational.

4.5 Display Active Branch In Terminal Prompt

If you don’t see the active Git branch displayed as part of your terminal prompt, I highly suggest you do so as being able to see clearly what branch you’re working on will save your hide. Knowing what branch is currently active ranks right up there with knowing exactly where you’re located on your file system, or knowing where your cat goes when it gets dark outside. Configuring your prompt to display the active Git branch depends upon which operating system you’re using. I’ll start with Windows and the Git Bash terminal.

4.5.1 Windows and Git Bash Terminal

Good news! As you learned in chapter 1 (*See “Customize Bash Prompt” on page 52.*), the Git Bash terminal prompt comes pre-configured to display the active Git branch.

4.5.2 Linux Mint

If you have already followed the configuration steps in chapter 1 you can skip this section. If not, please continue.

To configure the bash prompt in Linux Mint to display the current Git branch, you’ll need to modify the prompt definition located in the *.bashrc* file. Open the *.bashrc* file with your editor of choice, navigate to line 60, comment out that line by adding a hashtag ‘#’ to the beginning, create a new line and paste the following prompt configuration:

```
PS1='\n\[\033[35m\]$(/usr/bin/date)\n\[\033[32m\]\w \[\033[1;33m\]
\W$(__git_ps1 " (%s)") \n\[\033[1;32m\][\!:\#]\[\033[1;33m\] \u@\h $
\[\e[0m\]'
```

Note this is all one line of code. Breaking it down — PS1 is the prompt variable. It is initialized here with a string containing a mix of *terminal codes*, *prompt variables*, and *command expansions*. It starts with a newline character '\n' to separate the prompt from the results of the previous command. Next is the color code for the color purple followed by an expansion of the current date followed by another new line character. The '\w' prompt variable prints the current working directory path ($PWD) followed by a color change to yellow. The '\W' variable prints the basename of ($PWD) or simply the current directory name followed by the expansion `$(__git_ps1 "(%s)"` which inserts the current Git branch. This is followed again by a new line character and the color code for bright green and the sequence `[\!:\#]` which denotes the shell command history number and the current command number. This is followed by another change to the color yellow followed by the *username@hostname* '\u@\h' and finally the prompt symbol '$' followed by a space and the final escape sequence '\[\e[0m\]', which removes all formatting and attributes before the cursor is displayed.

After you complete these modifications, your Linux Mint terminal prompt will look like figure 8-23.

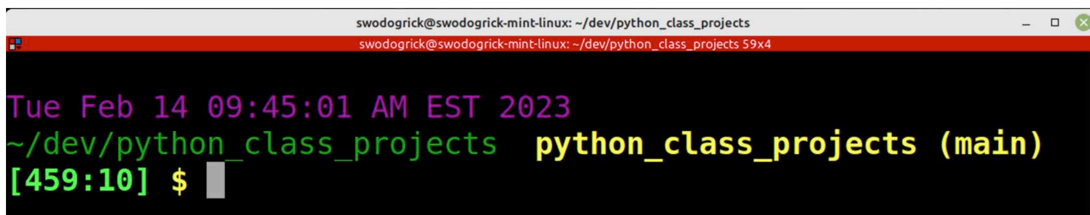

Figure 8-23: Linux Prompt with Active Git Branch

4.5.3 macOS

To customize the shell prompt, you'll need to load the *git-prompt.sh* file into your *~/.bash_profile*. The *git-prompt.sh* file is installed with the Xcode developer command-line tools. If you haven't installed Xcode or the developer command-line tools please do so now before proceeding. When you're ready, edit your *~/.bash_profile* and add the following line:

```
source /Library/Developer/CommandLineTools/usr/share/git-core/git-prompt.sh
```

This will load and execute the *git-prompt.sh* file. Now, copy and paste the following prompt configuration into your *~/.bash_profile* below the line you just copied:

```
PS1="\n\[\033[35m\]\$(/bin/date)\n\[\033[32m\]\w \[\033[1;33m\]\$(__git_ps1
'(%s)')\n\[$(iterm2_prompt_mark)\]\[\033[1;32m\][\!:\#]\[\033[1;33m\] \u@\h $
"
```

This prompt configuration gives you a prompt like the one shown in figure 8-24.

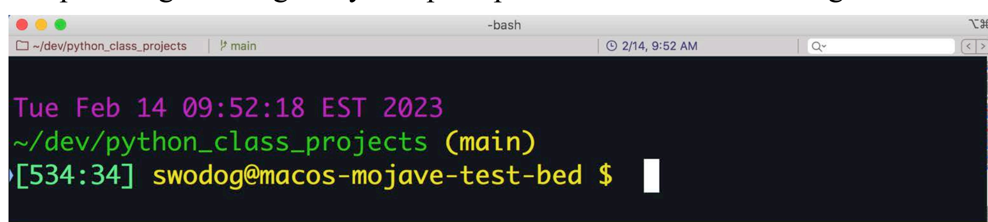

Figure 8-24: Active Git Branch Shown in macOS iTerm Command Prompt

Quick Review

You need to give some thought about how you intend to organize your repository. Repository organization is really directory organization. Small software development projects usually have a simple directory structure while large, complex development projects have a directory structure that better supports multiple teams with different development responsibilities working on the same repository. For students, a simple repository structure organized around assigned class projects is ideal.

If you're new to Git and GitHub, I recommend creating a repository on GitHub first and then cloning the repository to your local machine. When you create the repository, add *README.md* and *.gitignore* files.

Use the `git clone` command to clone the repository to your local machine. It helps to see the active branch displayed in your command prompt.

5 A Simple Git Workflow

In this section, I want to demonstrate a simple Git workflow ideal for students and individual contributors. It assumes you're the only one working on your repository although you may do so from multiple machines. The commands I'm going to discuss include: `git clone`, `git status`, `git add`, `git commit`, and `git push`. You can easily perform 99% of routine Git repository operations with just these five commands. Refer back to figure 8-3 to see a visual representation of what these commands do.

5.1 git clone

You've already used the `git clone` command to clone a repository. The first thing you need to get used to, and you'll forget from time to time, is the need to preface each Git command with `'git'`. The general form of the `git clone` command is:

```
git clone github_repository_url
```

The best way to get the `github_repository_url` is to go to the GitHub repository page and copy it. At some point, you'll memorize the GitHub SSH and HTTPS repository URL forms, but until then, copy it directly from the repository page to avoid making simple, forehead-smacking mistakes.

5.2 git status

You'll want to run the `git status` command after you make changes to your local repository to see a list of those changes. The command is simply:

```
git status
```

To see the effects of this command, I'll make some additions and modifications to the *python_class_projects* repository I created and cloned earlier in this chapter. First, I'm going to create a subdirectory called *project_1*, and in this directory I'll add a *README.md* file along with a *main.py* file. At this point, what's in these files is not important, as I just want you to see the

effects of using the `git status` command. Figure 8-25 shows the `git status` command executed from the root of the local *python_projects_repository* directory.

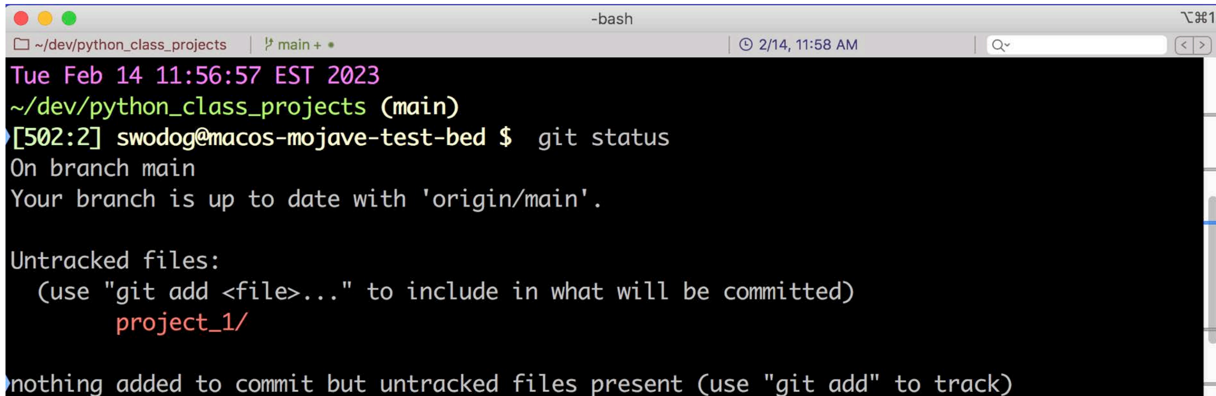

Figure 8-25: Running git status After Adding project_1 Folder with a Few New Files

Referring to figure 8-25 — I've changed my terminal color settings to make it easier to read the `git status` command output. In this case, it's reporting the entire *project_1* folder in the **Untracked files** section. The last line says use "`git add`" to track. **NOTE:** You may have files you do not want to track. You can add these files to the *.gitignore* file. Since I selected the Python *.gitignore* template when I created the repository on GitHub, it already lists a lot of common files found in typical Python projects. I recommend exploring the contents of this file to see what it contains. You can always add to it. One file macOS users need to add to it is the *.DS_Store* file, which gets created in a folder when you open it with a Finder window.

5.3 Git Add

Referring to figures 8-3 and 8-25— At this point, I've modified files in my workspace, but if I want to put these files under source code management and track their changes going forward, I'll need to **stage** them for the next commit. That's the purpose of the `git add` command. I also want to add everything I just created so this is how I'll use the `git add` command:

```
git add .
```

The dot means "Stage everything that's been modified, added, or deleted." Figure 8-26 shows the results of running the `git add .` command followed by `git status`.

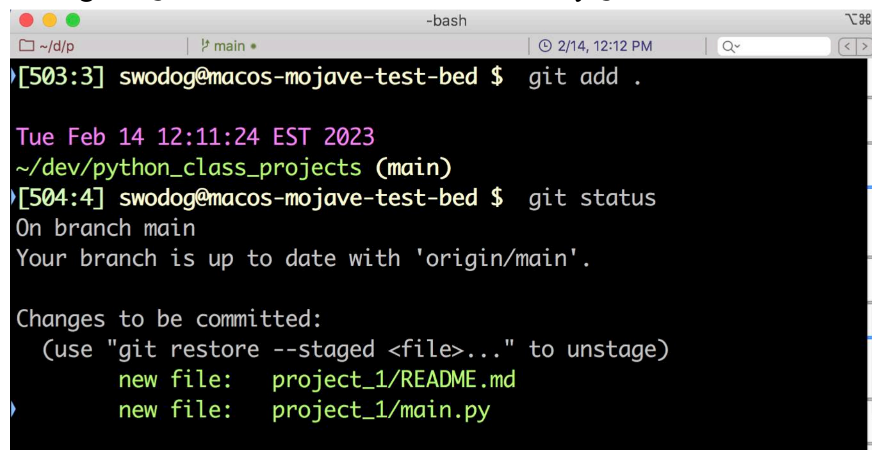

Figure 8-26: Running `git add .` Followed by `git status` Command

Referring to figure 8-26 — Running `git add .` produced no output. The `git status` command shows the staged changes. All the files in the *project_1* folder are staged for the next commit. This is considered typical usage of the `git status` command. You'll run `git status` before using `git add` to to see what's been added, modified, or deleted, and again afterwards to see what files were staged. Sometimes you'll stage files by mistake. As you can see above, the output from the `git status` command offers help on how to *unstage* a file. I'll offer advice on how to avoid or correct simple mistakes later in this chapter.

5.4 Git Commit

The `git commit` command actually saves the changes to the local repository. Files staged for commit with the `git add` command are the files that will be saved to the local repository with the `git commit` command. The `git commit` command takes the following general form:

```
git commit -m "Comment message about the commit."
```

To commit the changes I staged in the previous section, I'll use the `git commit` command like so:

```
git commit -m "Added project_1 folder."
```

Figure 8-27 shows the results of running this command.

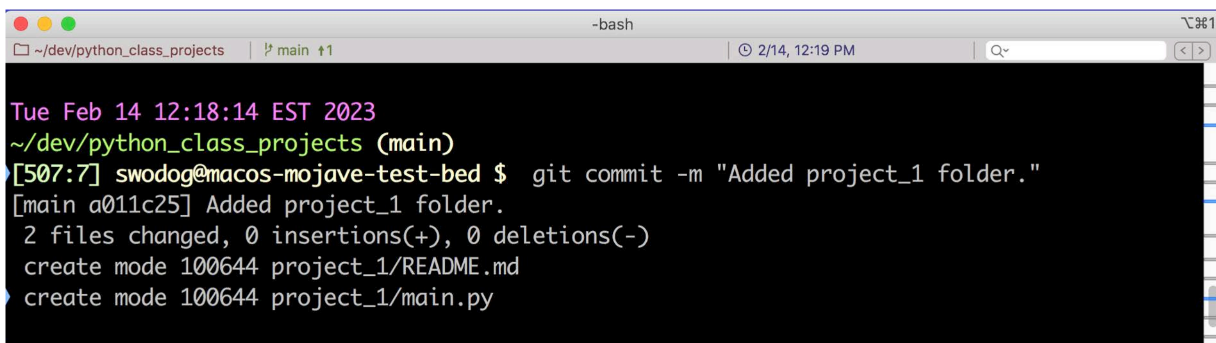

Figure 8-27: Results of Running `git commit` Command

Referring to figure 8-27 — At this point, all the additions, modifications, and deletions have been saved to the local repository. You can continue to work on the project, adding, modifying, or deleting files as required, and repeat the `git status`, `git add`, and `git commit` commands until you reach a point where you want to push all those changes to the remote repository with the `git push` command.

5.5 Git Push

The `git push` command transfers the state of your local repository to the remote repository. To use the command, you simply type:

```
git push
```

Figure 8-28 shows the results of running this command.

Referring to figure 8-28 — After you push your local repository changes to the remote repository, you can check the contents of the remote repository to verify the changes have been successfully been applied as shown in figure 8-29.

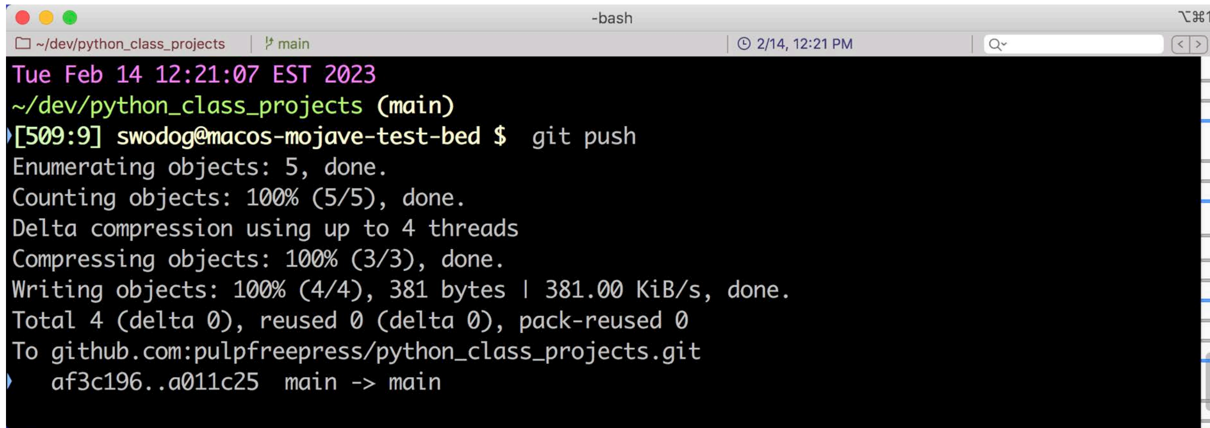

Figure 8-28: Pushing Local Repository `main` Branch Changes to Remote Repository with `git push` Command

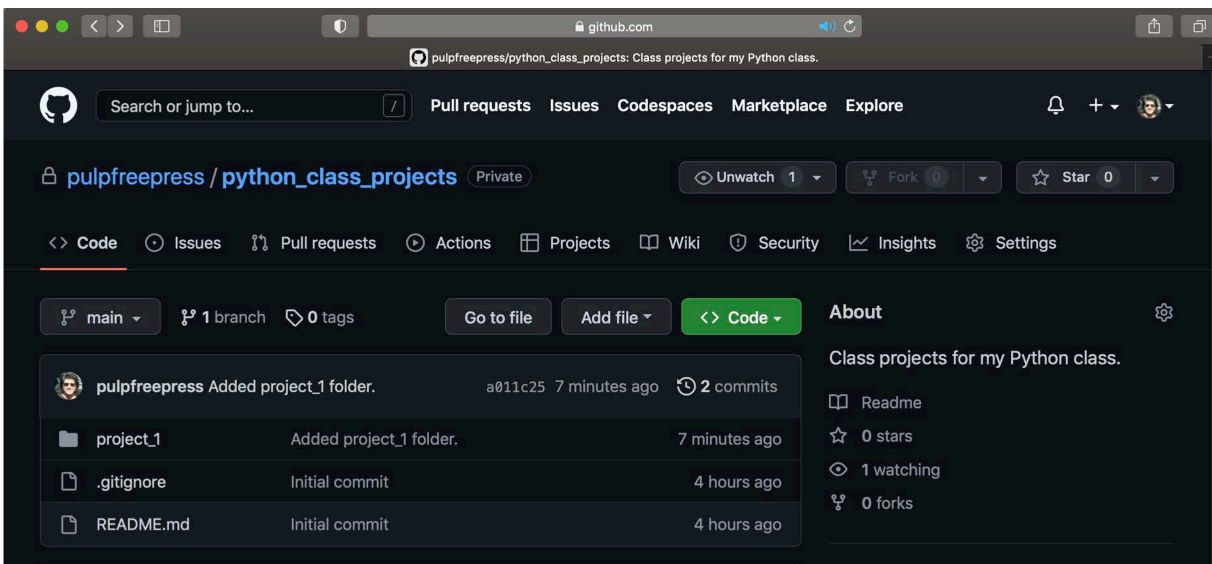

Figure 8-29: GitHub Remote Repository After Pushing Changes

5.6 Simple Git Workflow Command Summary

Table 8-3 summarizes the Git commands used in the simple workflow presented in this section.

Git Command	Description
`git clone` *`repository_url`*	Makes a complete copy of a remote repository branch on your local machine. The branch you typically clone is named either `main` or `master`. When you clone the repository, all tracked files are automatically checked out and made available in your workspace, where you can add, edit, or delete files as required.
`git status`	Shows the status of files in your workspace. Run this command often to see the state of files before and after staging them with the `git add` command.

Table 8-3: Simple Git Workflow Command Summary

Git Command	Description
`git add .`	Stages new, modified, or deleted files for the next commit. The `'.'` signifies all files. You can also specify individual files.
`git commit -m "message"`	Saves staged changes to the local repository.
`git push`	Transfers local repository changes to the remote repository. You can verify the push was successful by inspecting the remote repository.

Table 8-3: Simple Git Workflow Command Summary (Continued)

5.7 Parting Thoughts

Throughout this section, I have performed all work on the `main` branch. If you're a student or professional working on a personal project, this is perfectly acceptable. However, if you intend to collaborate with someone or want to test out an idea without mucking up the `main` branch, you'll need to create a new branch on which to work, and then later merge that branch into the `main` repository branch. I cover the concepts of branching and merging in the next section.

Quick Review

As a student or individual, you can easily accomplish 99% of everything you need to with Git using a simple workflow with the following commands: `git clone`, `git status`, `git add`, `git commit`, and `git push`. Use the `git status` command often, especially before and after staging files for commit with the `git add` command, to verify workspace file additions, changes, and deletions. If you do not what to track changes of a particular file or set of files, add them to the *.gitignore* file.

6 Branching And Merging

As discussed above, when working on a project as the sole developer, you are pretty safe working on the `main` branch. However, there will be times when you'd like to try out an idea in the code but aren't sure you want to add that feature into the application. This is where *branching* comes to the rescue.

Referring to figure 8-4 — Branching allows you to create and work against a *copy* of the repository. Later, if you like the results, you can push the new branch to the remote repository and merge it with the `main` branch. This is the workflow used by zillions of development teams around the world. Most development teams incorporate a *Peer Review* into this workflow. In a peer review, the developer who pushed the new branch to the repository submits a *Pull Request*. By team agreement or policy, other development team members formally review the proposed changes. If they approve the changes, they are merged into the `main` branch. At that point, the new branch is deleted. Other team members then need to update their local copy of the repository's `main` branch with a `git pull` to ensure it's up-to-date before proceeding with new work.

6.1 Creating A New Branch

To create a new branch, use the `git checkout` command with the `-b` switch, but first, a scenario. In the previous section I added a *project_1* folder along with several files to the *python_-class_projects* repository and pushed those changes to the remote repository. Now suppose I have an idea for a new feature in my project but don't want to modify any of the files in the `main` branch. The reason I don't want to modify the `main` branch files is because if I don't like the results I'd need to back out all those changes, which would be super pain in the ass, depending on how many files I modified, and likely introduce errors into working code. Creating a new branch effectively makes a copy of the `main` branch and isolates work to the new branch.

To do this, I will navigate to the local *python_class_projects* directory and create a new branch I'll name `feature_a` with the following command:

```
git checkout -b feature_a
```

Figure 8-30 shows the results of running this command.

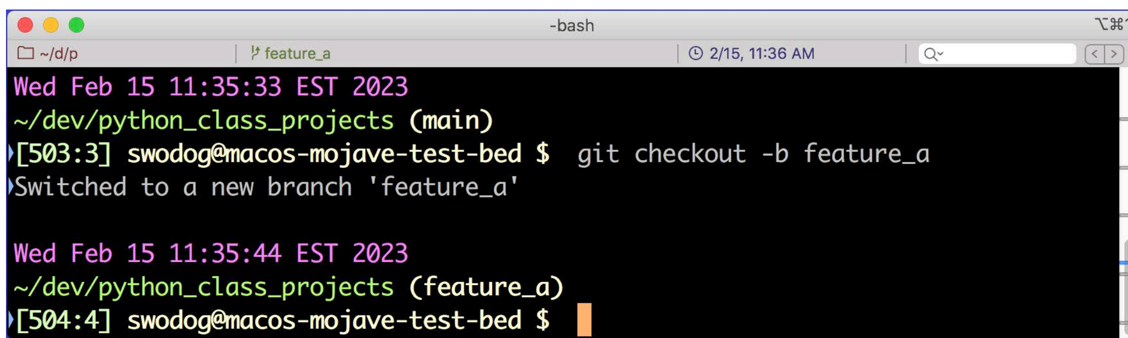

Figure 8-30: Creating New Branch with `git checkout -b` Command

Referring to figure 8-30 — Notice, I started out on the `main` branch. Running `git checkout -b feature_a`, created a new branch and automatically switched me into it. To list all current branches use the `git branch` command like so:

```
git branch
```

Figure 8-31 shows the results.

Figure 8-31: List Branches with `git branch` Command

Referring to figure 8-31 — You can see that the `feature_a` branch is highlighted and marked with an asterisk, which indicates it is the active branch.

To switch between branches use the `git checkout` command followed by the name of the branch you'd like to work on. For example, to switch back to the `main` branch use:

```
git checkout main
```

Follow that with a `git branch` command to see the active branch in the list of branches:

```
git branch
```

Figure 8-32 shows the results of running these two commands.

```
Thu Feb 16 07:10:21 EST 2023
~/dev/python_class_projects (feature_a)
[502:2] swodog@macos-mojave-test-bed $  git checkout main
Switched to branch 'main'
Your branch is up to date with 'origin/main'.

Thu Feb 16 07:10:30 EST 2023
~/dev/python_class_projects (main)
[503:3] swodog@macos-mojave-test-bed $  git branch
  feature_a
* main
```

Figure 8-32: Switching Branches with `git checkout` Command

Referring to figure 8-32 — I was in the `feature_a` branch and switched to the `main` branch by running `git checkout main`. Running `git branch` shows `main` is indeed the active branch, as does the command-prompt branch indicator. I will now switch back to the `feature_a` branch:

```
git checkout feature_a
```

Once there, I will make whatever modifications are necessary to implement "feature a", staging those changes with `git add` and saving them with `git commit`. When I'm ready, I push the `feature_a` branch to the remote repository using `git push`:

```
git push
```

Figure 8-33 shows the results of running this command.

```
Thu Feb 16 07:44:44 EST 2023
~/dev/python_class_projects (feature_a)
[536:36] swodog@macos-mojave-test-bed $  git push
Enumerating objects: 10, done.
Counting objects: 100% (10/10), done.
Delta compression using up to 4 threads
Compressing objects: 100% (7/7), done.
Writing objects: 100% (7/7), 740 bytes | 740.00 KiB/s, done.
Total 7 (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 'feature_a' on GitHub by visiting:
remote:      https://github.com/pulpfreepress/python_class_projects/pull/new/feature_a
remote:
To github.com:pulpfreepress/python_class_projects.git
 * [new branch]      feature_a -> feature_a
```

Figure 8-33: Pushing feature_a Branch to Remote Repository — Time for a Pull Request

Referring to figure 8-33 — Notice that when pushing a new branch to the remote repository, the output of the `git push` command suggests submitting a *pull request*. I'll show you how to do that in the following sections, but first, let's take a look at the remote repository. Figure 8-34 shows my *python_class_projects* GitHub page after pushing the `feature_a` branch.

© 2023 Pulp Free Press

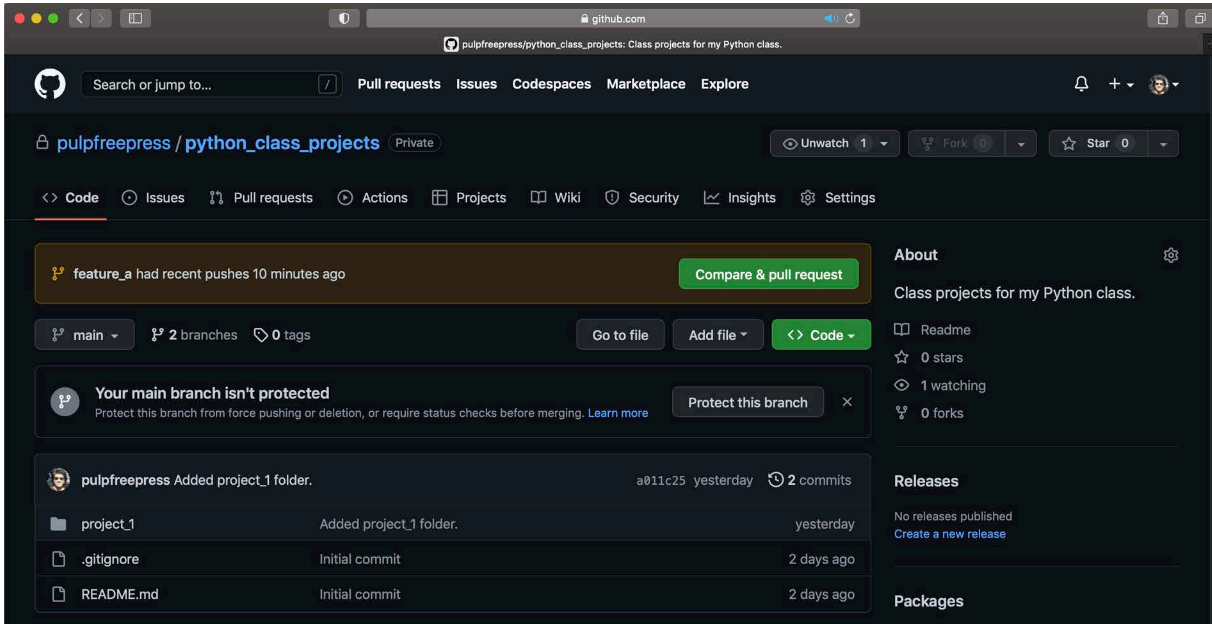

Figure 8-34: GitHub python_class_projects Repository Page After Pushing `feature_a` Branch

Referring to figure 8-34 — Notice there is now a notification on the repository page stating that the *feature_a branch had recent pushes 10 minutes ago*. To the right of that notification is a button that reads **Compare & pull request**. Click that button to go to the pull request page as shown in figure 8-35.

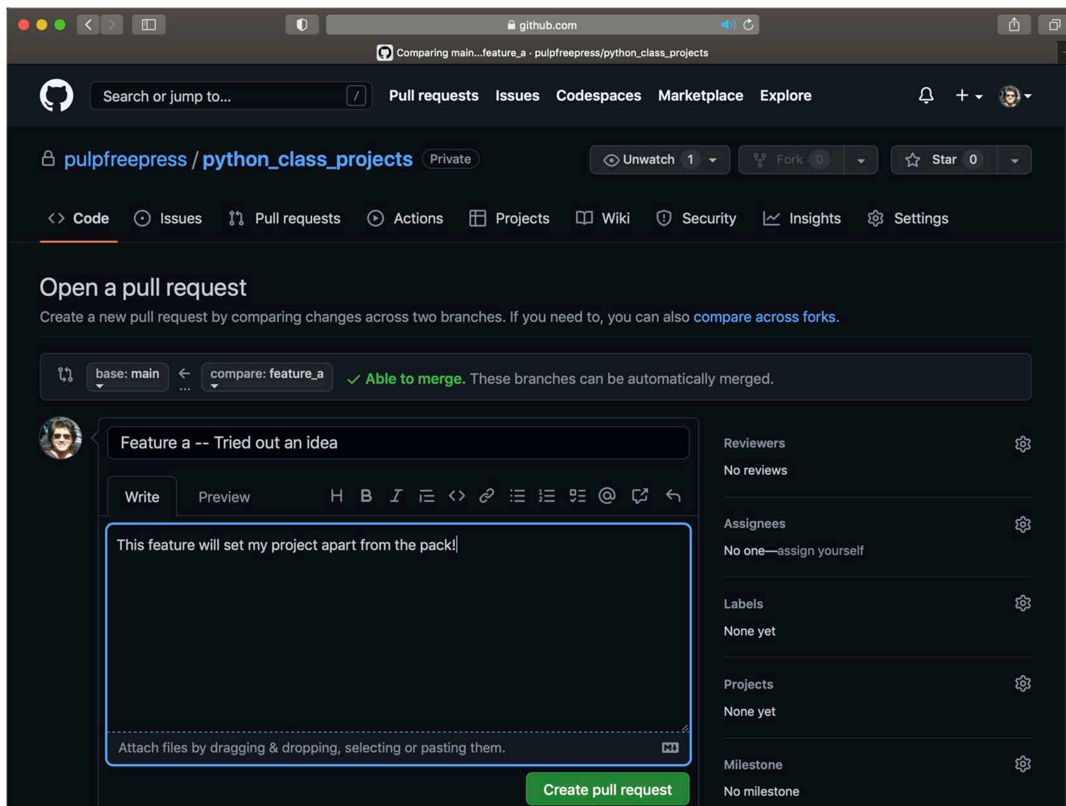

Figure 8-35: Open Pull Request Page

Referring to figure 8-35 — Notice that the `feature_a` branch can be automatically merged with the `main` branch. Working as an individual, this will almost always be the case when you push a new branch to the remote repository. Since you're the only one working on the code base, there are unlikely to be any conflicts between the branch you want to merge and the `main` branch. I say *almost always* because you could accidentally modify and push the `main` branch before pushing the new branch, which would introduce conflicts between the two branches. Just something to think about.

OK, while on the Open Pull Request page, scroll down and explore. It's designed for collaboration. A pull request signals to other developers that you think your code is ready to be merged, but by policy or team practice, and out of an abundance of caution and good engineering practice, nothing's getting merged until it has been peer reviewed and approved. That's the function of the pull request page with its history of comments and other annotations.

When ready, click the **Create pull request** button. This pull request is now officially open and ready for review and approval as shown in figure 8-36.

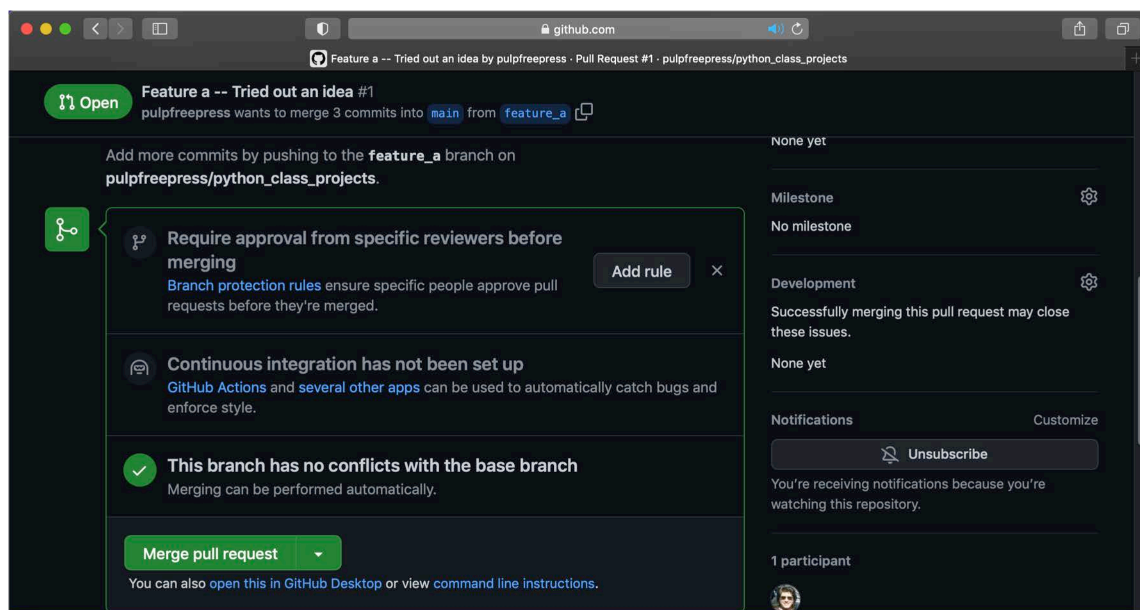

Figure 8-36: Pull Request Opened and Ready for Review and Merging

Referring to figure 8-36 — Click the **Merge pull request** button, then click the **Confirm merge** button. At this point you can delete the `feature_a` branch as shown in figure 8-37.

Referring to figure 8-37 — This deletes the `feature_a` branch from the remote repository. You'll then need to delete the `feature_a` branch from your local repository and update your `main` branch. To do this, use the following commands:

```
git checkout main
```

Switches to the `main` branch.

```
git branch -D feature_a
```

Deletes the `feature_a` branch.

```
git pull
```

Updates the local `main` branch with the changes merged from `feature_a` branch. Figure 8-38 shows the results of running these commands.

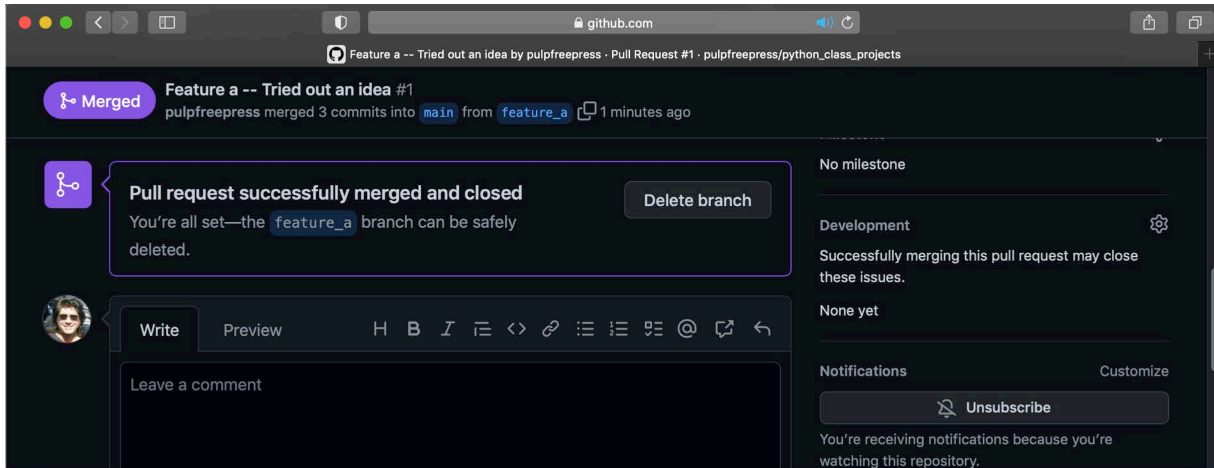

Figure 8-37: Pull Request Merged — You Can Delete `feature_a` Branch

```
-bash
~/d/p/project_1    main    2/17, 7:29 AM
Fri Feb 17 07:27:33 EST 2023
~/dev/python_class_projects/project_1 (feature_a)
[518:18] swodog@macos-mojave-test-bed $  git checkout main
Switched to branch 'main'
Your branch is up to date with 'origin/main'.

Fri Feb 17 07:27:46 EST 2023
~/dev/python_class_projects/project_1 (main)
[519:19] swodog@macos-mojave-test-bed $  git branch -D feature_a
Deleted branch feature_a (was 975c297).

Fri Feb 17 07:28:06 EST 2023
~/dev/python_class_projects/project_1 (main)
[520:20] swodog@macos-mojave-test-bed $  git pull
remote: Enumerating objects: 1, done.
remote: Counting objects: 100% (1/1), done.
remote: Total 1 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (1/1), 635 bytes | 635.00 KiB/s, done.
From github.com:pulpfreepress/python_class_projects
   a011c25..2cb9852  main       -> origin/main
Updating a011c25..2cb9852
Fast-forward
 README.md                                  | 2 ++
 project_1/README.md                        | 3 +++
 project_1/{main.py => src/feature_a_module} | 0
 project_1/src/main.py                      | 0
 4 files changed, 5 insertions(+)
 rename project_1/{main.py => src/feature_a_module} (100%)
 create mode 100644 project_1/src/main.py

Fri Feb 17 07:28:19 EST 2023
~/dev/python_class_projects/project_1 (main)
[521:21] swodog@macos-mojave-test-bed $
```

Figure 8-38: Deleting Local `feature_a` Branch.

Referring to figure 8-38 — At this point, you can return to work on the `main` branch or checkout a new branch and repeat the process described in this section to crank out your next big idea!

6.2 Git Branching Workflow Command Summary

Table 8-3 lists the Git commands related to branching used in this section.

Git Command	Description
`git checkout -b branch_name`	Create a new branch named `branch_name` and switch to it.
`git branch`	List branches in local repository.
`git checkout branch_name`	Switch to `branch_name`
`git branch -D branch_name`	Delete `branch_name`

Table 8-4: Git Commands Related to Branching

Quick Review

Branching allows you to create and work against a *copy* of the repository. Later, if you like the results, you can push the new branch to the remote repository and merge it with the `main` branch. This workflow is used by development teams around the world, and individual developers find it helpful as well, as it allows you to work on an idea without modifying `main` branch code.

Start by creating a new branch with the `git checkout -b` command. This creates a new branch and automatically switches into it. To switch between branches use `git checkout`. Work in the new branch as you would normally work, and when ready, push the new branch to the remote repository. You'll then need to create a new pull request to merge the new branch with the `main` branch. In a team environment, opening a new pull request is a signal to other developers to conduct a peer review of your code. Any problems or conflicts must be resolved before you can merge the new branch into the `main` branch and close the pull request.

When the new branch has been successfully merged with the `main` branch, and the pull request has been closed, you can delete the new branch from the remote repository as well as from your local machine. Be sure to update your local repository `main` branch with the `git pull` command before starting new work.

7 Avoiding Common Mistakes

Above the chart table onboard the USS America (CV-66) hung a plaque that read: "*The superior seaman uses his superior intellect to avoid situations which require the use of his superior skills*." Laymen might say instead: "*The best defense is a good offense*!", and it applies to Git operations as well. In this section, I'd like to present a short list of common issues and how to avoid them, and if you do get into a jam, how to recover. Note that this list is not exhaustive by any stretch of the imagination.

7.1 Mindfulness

By mindfulness, I mean be aware of what where you are in the file system and on what branch you're working. If you're and individual working only on the `main` branch you have it pretty easy.

If you do create a new branch then pay attention to which branch you're in. This is where the command-line Git branch indicator saves your hide.

7.2 Use A Documented Workflow

Follow a documented workflow until you have it memorized. By documented workflow, I mean write each Git command down in the order in which you need to call them. Referencing a diagram like the one in figure 8-4 is super helpful until you learn the ropes. Better yet, make a copy of that picture and paste it into your Engineer's Notebook.

7.3 Think Before You Commit

No, I'm not talking about relationships. This is related to mindfulness. Lots of files have no business being in the remote repository. The purpose of the *.gitignore* file is to maintain a list of artifacts NOT to add to the repository. However, one day, you will be cranking out the code in God mode, in the Flow, kickin' it to your playlist of *retro jams*, and do a `git add .` and `git commit -m "This is a sweet modification!"` in rapid succession only to realize you just committed a top-secret password configuration file, or worse yet, all your private SSH keys. First thing you need to do is to remember to breathe and not panic. The general formula for recovery from this situation goes something like this:

```
git rm --cached file_to_remove
```

Or, to remove a whole directory use:

```
git rm --cached -r directory_to_remove
```

Note that you want to catch this sort of mistake before you push the errant file to the remote repository, especially if it contains sensitive information. Once you have removed the file using the `git rm` command, add it to your *.gitignore* file.

7.4 Push Before You Stop

At the end of the day, push your changes to the remote repository. This can be considered a disaster recovery move and insurance against catastrophe. At least your latest changes will be stored off site and not affected if you spill your latte on your laptop.

Pushing to the `main` branch at quitting time is perfectly fine if you're a student or individual contributor, but if you work as part of a development team, you'll want to work on a separate branch that doesn't trigger an automated build process.

7.5 Pull Before You Work

This is a critical step you need to include in your workflow, especially if you work on the same repository branch with multiple machines. One of the cool things about using Git is that you can work from home or an office on a desktop machine, push your changes, migrate to your favorite coffee shop, do a `git pull` to update your local branch on your laptop, work on stuff, push your changes, return home, switch back to a different desktop, and do a `git pull` to update your local repository. If you mess this up then you'll introduce conflicts into your code base, which, at the least, will be annoying to sort out and resolve.

7.6 If You Really Get Into A Jam

If you really find yourself in a tight spot, I recommend flexing your Google dork chops. You wouldn't be the first developer to need help with Git in special situations. And that's not just a cheap way to end this section, tons of professional developers around the world Google for help with Git every single day.

Quick Review

The best way to avoid common Git problems is to adopt and follow a documented workflow. Write down your workflow process steps and the Git commands associated with each and follow the guide until you memorize the workflow.

The most common mistakes you'll make include committing one or more files that contain sensitive information and should not be in the remote repository (In other words, they should remain untracked.), and getting out of sync with pushes to the remote repository, which will introduce conflicts in the codebase. This can happen especially if you're working on the same repository branch from two different machines.

Summary

Source Code Management, or SCM, is a set of tools and processes designed to track and manage changes to software project source code files and related artifacts. The SCM tools selected for this chapter include Git and GitHub.

Git is a powerful, lightweight, fast, flexible, and distributed source code management system. Its power intimidates both neophytes and experienced SCM users but in reality, you can get a lot done with Git with only a small handful of commands.

Work generally starts with cloning a remote repository. There's usually one branch in the remote repository designated `main` or `master`. Cloning a remote repository makes a complete copy of the repository on your local machine and does an automatic checkout of tracked files into your workspace, ready for editing. As work progresses, you edit files, add new files, or remove files as required. Stage changes with the `git add` command. Save changes to your local repository with the `git commit` command, and push local repository changes to the remote repository with the `git push` command.

GitHub requires the use of SSH keys to enable secure communications between local and remote repositories. SSH keys and configuration files are located in the *~/.ssh* directory. Use a temporary directory, *~/tmp*, to practice SSH key generation until you're confident you're generating the SSH keys you want. You can then copy the public and private keys to the *~/.ssh* directory when you're ready to use them. To use your SSH keys with GitHub, you'll need to add the public SSH key to GitHub, and add the private key to your computer's SSH agent.

You need to give some thought about how you intend to organize your repository. Repository organization is really directory organization. Small software development projects usually have a simple directory structure while large, complex development projects have a directory structure that better supports multiple teams with different development responsibilities working on the same repository. For students, a simple repository structure organized around assigned class projects is ideal.

 © 2023 Pulp Free Press

If you're new to Git and GitHub, I recommend creating a repository on GitHub first and then cloning the repository to your local machine. When you create the repository, add *README.md* and *.gitignore* files.

Use the `git clone` command to clone the repository to your local machine. It helps to see the active branch displayed in your command prompt.

As a student or individual, you can easily accomplish 99% of everything you need to with Git using a simple workflow with the following commands: `git clone`, `git status`, `git add`, `git commit`, and `git push`. Use the `git status` command often, especially before and after staging files for commit with the `git add` command, to verify workspace file additions, changes, and deletions. If you do not what to track changes of a particular file or set of files, add them to the *.gitignore* file.

Branching allows you to create and work against a *copy* of the repository. Later, if you like the results, you can push the new branch to the remote repository and merge it with the `main` branch. This workflow is used by development teams around the world, and individual developers find it helpful as well, as it allows you to work on an idea without modifying `main` branch code.

Start by creating a new branch with the `git checkout -b` command. This creates a new branch and automatically switches into it. To switch between branches use `git branch`. Work in the new branch as you would normally work, and when ready, push the new branch to the remote repository. You'll then need to create a new pull request to merge the new branch with the `main` branch. In a team environment, opening a new pull request is a signal to other developers to conduct a peer review of your code. Any problems or conflicts must be resolved before you can merge the new branch into the `main` branch and close the pull request.

When the new branch has been successfully merged with the `main` branch, and the pull request has been closed, you can delete the new branch from the remote repository as well as from your local repository.

The best way to avoid common Git problems is to adopt and follow a documented workflow. Write down your workflow process steps and the Git commands associated with each, and follow the guide until you memorize the workflow.

The most common mistakes you'll make include committing one or more files that contain sensitive information and should not be in the remote repository (In other words, they should remain untracked.), and getting out of sync with pushes to the remote repository, which will introduce conflicts in the codebase. This can happen especially if you're working on the same repository branch from two different machines.

Skill-Building Exercises

1. **Create Practice GitHub Repository:** Following the guidance given in section 5, create a GitHub repository to practice the Git workflows discussed in this chapter.

2. **Generate and Configure SSH Keys:** Following the guidance given in section 4, generate and configure your SSH keys for use with GitHub. Document the process with any lessons learned in your Engineer's Notebook.

3. **Practice Simple Git Workflow:** Practice the simple Git workflow presented in section 6.

Don't worry about what's in the files. You can create empty files with the `touch` command, and directories with the `mkdir` command. Practice the following Git commands: `git clone`, `git status`, `git add`, `git commit`, and `git push`. Document these commands in your Engineer's Notebook along with any switches and arguments required.

4. **Practice Branching and Merging Git Workflow:** Practice the branching and merging workflow presented in section 7. Document the commands you use in your Engineer's Notebook along with any switches and arguments required.

5. **Practice Recovering From Mistakenly Committed File:** In your practice local repository, add and commit a file named *sensitive_data.txt*. Use the git rm --cache command to remove it from the commit. Next, add the file to your *.gitignore* file. Note the effects this has when you next run the `git status` command.

6. **Practice Recovering From Mistakenly Pushing File To Remote Repository:** This is a bit more challenging, as the process to correct this sort of mistake is not covered in this chapter. Again, create an arbitrary file and push it to the remote repository. Research the actions required to remove that file from being tracked while not deleting the file completely. There is a brute force method to fix this problem. Can you figure out what it is?

7. **Research Git Commands:** Browse the official Git documentation and see what commands are available that were not discussed in this chapter: *https://git-scm.com*

8. **Download the Pro Git Book:** You can download the Pro Git book and use it as a handy reference: *https://github.com/progit/progit2/releases/download/2.1.360/progit.pdf*

9. **Automatically Load Private SSH Key:** Follow the steps in section 4.12 for your particular operating system to configure your SSH private key to load automatically.

10. **Draw The Git Workflows Discussed In This Chapter:** Make your own drawings of the Git workflows discussed in this chapter. Include the local and remote repositories along with workspace and staging areas. Include the commands used with each workflow and how they affect tracked and untracked files.

Suggested Projects

1. **Create Class Programming Projects Repository:** Create a GitHub repository to house your class programming projects. You may want a different repository per class. Adopt the repository structure given in figure 8-16. I recommend initially creating it as a private repository until to guard against prying eyes. If you need to collaborate with another student, you can grant them access to repository.

2. **Explore the .gitignore File:** Create a repository in GitHub and add the README.md and ,gitignore files. Select the Python .gitignore template. Clone the repository to your local

0
0
0
0
1
0
0
0

machine and explore the contents of the .gitignore file.

3. **Study Markdown Language:** The Markdown language is used to add content to the README.md file. Visit the Markdown Guide site and study the Markdown Cheat Sheet: *https://www.markdownguide.org*

4. **Explore The .git Directory:** Located at the root of a local repository is a *.git* hidden directory. Explore the contents of the *.git* directory, make a list of each file or subdirectory you find, and make a note of its purpose.

2. Self-Test Questions

1. Define in your own words the meaning of Source Code Management.

2. List several benefits to adopting Source Code Management.

3. (True/False) Git supports multiple workflows.

4. Explain in your own words what is meant by "Git is distributed." Draw a picture to support your answer.

5. (True/False) Cloning a remote repository creates a copy of the repository on your local machine.

6. What's the difference between tracked and untracked files?

7. In which special file do you list files you want to remain untracked?

8. Explain in your own words why you would want to create and work on a new branch.

9. List and describe the steps and git commands required from the moment you check out a new branch until you merge the branch with the remote repository's `main` branch.

10. What Git command would you use to remove a file from a recent commit?

References

Interactive Git Cheat Sheet, *https://ndpsoftware.com/git-cheatsheet.html#loc=index;*

Do You Have A Git Mess On Your Hands?, *http://justinhileman.info/article/git-pretty/git-pretty.png*

00001000

Git SCM, *https://git-scm.com*

Git For Windows, *https://gitforwindows.org*

OpenSSH, *https://www.openssh.com*

Setup SSH Authentication for Git Bash on Windows, *https://gist.github.com/jherax/979d052ad5759845028e6742d4e2343b*

ssh-agent Man Page, *https://linux.die.net/man/1/ssh-agent*

Working with SSH Passphrases, GitHub, *https://docs.github.com/en/authentication/connecting-to-github-with-ssh/working-with-ssh-key-passphrases?platform=windows*

Markdown Guide Website, *https://www.markdownguide.org*

Notes