```
remote: Compressing objects: 100% (15/15), done.
remote: Total 15 (delta 4), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (15/15), done.
$ cd GitHubForDummiesReaders
```

You can verify where the remote/target repo is with the following command:

```
$ git remote -v
originhttps://github.com/thewecanzone/GitHubForDummiesReaders.
  git (fetch)
originhttps://github.com/thewecanzone/GitHubForDummiesReaders.
  git (push)
```

If you cloned the same repo as I did, you see the exact same origin URLs for `fetch` and `push`. You should see that the remote repo is one owned by `thewecanzone` and not `dra-sarah`. Alternatively, if you run the same command on a repo that you own, you should see your username. For example, if I run the command in the directory where I cloned my website repo that I created in Chapter 4, I would see

```
$ git remote -v
originhttps://github.com/dra-sarah/dra-sarah.github.io.git (fetch)
originhttps://github.com/dra-sarah/dra-sarah.github.io.git (push)
```

If you try using the command on a Git repo that doesn't have a remote origin (meaning it isn't hosted on GitHub.com or any other remote place), you simply won't get any information back. For example, in Chapter 1, I created a simple Git repo called `git-practice`. Running the command in that directory gives you nothing back:

```
$ git remote -v
```

Forking a Repository

The goal of open source is to encourage collaboration among software developers around the world, so being able to contribute code to repositories where you aren't the owner or an explicit collaborator is an important part of the GitHub workflow and mission. To become a collaborator of an open source project, you can reach out to the owner of the repo and request to be a collaborator. However, if the owner doesn't know who you are, they probably won't add you as a collaborator because that would give you push rights to the repository. You'll have to gain their trust first.

REMEMBER

You don't need the owner's permission to fork their repository. You can make your contributions and share them with the owner to show how you can be an asset to the project.

To fork a repo, go to the repo home page and click the Fork button at the top right. If you'd like, you can use `https://github.com/thewecanzone/GitHubForDummiesReaders` to practice forking and contributing to a public repo.

After you click the Fork button, the web page refreshes, and you see a slightly modified version of the repo, as shown in Figure 6-2. At the top of the repo, you see that the repo is attached to your account, but it still has a reference to the original repo.

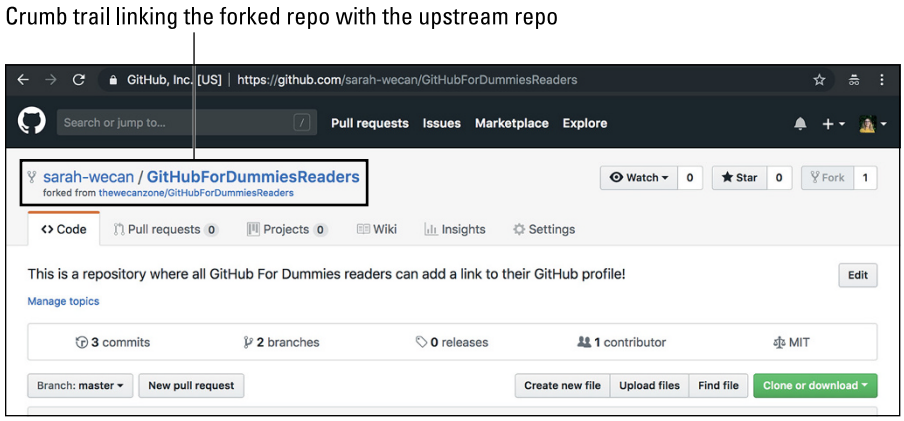

FIGURE 6-2: A forked repo on GitHub.com.

WARNING

If you're part of multiple GitHub organizations, you're asked to choose which organization you want to fork the repository to after you click the Fork button and before the web page refreshes.

After you have your own, forked version of the repo, you can clone it on your local machine to start making changes. Chapter 7 goes over writing code and creating commits, which is the same process whether you're on a forked repo or a regular repo. If you clone the repo using GitHub Desktop, your local Git repository knows about your forked version (remote `origin`) and the original repo (remote `upstream`).

TIP

The concept of a *remote* can be confusing to those new to distributed version control systems like Git. When you clone a GitHub repository, you have a full copy of the repository on your local machine. You may be tempted to think the copy of the repository on GitHub is the canonical copy. However, there is no concept of canonical in Git. The *canonical copy* is whatever the people working on the project

decide it is by consensus. Git does have the concept of a *remote,* which is a pointer to a copy of the same Git repository hosted elsewhere. Typically, a remote is a URL to a Git-hosting platform like GitHub, but it's possible to be a path to a directory with a copy of the repository. When you clone a repository, Git adds a remote named origin with the location (usually a URL) from where you cloned it. But it's possible to add multiple remotes to a Git repository to indicate other locations where you may want to push and pull changes from. For example, if you clone a fork of a repository, you may want to have a remote named upstream that points to the original repository.

If you clone the repo using the command line, you may want to set the upstream remote, which I explain in the section "Getting unstuck when cloning without forking," later in this chapter. You can see both the forked remote origin and original remote upstream if you run the `git remote -v` command in the directory where you cloned the repo:

```
$ git remote -v
originhttps://github.com/dra-sarah/GitHubForDummiesReaders.git
  (fetch)
originhttps://github.com/dra-sarah/GitHubForDummiesReaders.git
  (push)
upstreamhttps://github.com/thewecanzone/GitHubForDummiesReaders.
  git (fetch)
upstreamhttps://github.com/thewecanzone/GitHubForDummiesReaders.
  git (push)
```

The *origin,* which is your fork of the repository where you typically fetch/pull changes from and push changes to, has your username (`dra-sarah` in this example). The *upstream,* which is where the original code is located, and where you eventually want to contribute the code you write back to, has the original author's username (`thewecanzone` in this example). While you can push changes to and pull changes from any remote (origin or upstream), it is good practice to work on your fork of the repositoy, represented typically as *origin*.

Fetching changes from upstream

Having the upstream repo linked to your forked repo is important. As you start making changes, you want to be able to fetch/pull any changes that are being made on the original code into your code to make sure that you have the most up-to-date version.

For example, suppose that you forked and cloned a website project a week ago with plans to change the website's About page. While you were working on those

changes, someone else made a change to the About page. Their changes may conflict with your changes, or they may introduce something new that you want to use in your changes. Pulling those changes into your local repository before you submit your changes back to the original repository makes sense. It reduces the chance that your changes conflict with the changes others are making to the About page and makes it more likely the owner can accept them.

If you find yourself in a situation where you need to get the change from the upstream, original repo, you can go to the directory where your forked repo is and type

```
$ git fetch upstream
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/thewecanzone/GitHubForDummiesReaders
8404f3b..e02a4d2 master -> upstream/master
$ git checkout -b new-branch
Switched to a new branch 'new-branch'
$ git merge upstream/master
Updating 8404f3b..e02a4d2
Fast-forward
README.md | 2 +-
1 file changed, 1 insertion(+), 1 deletion(-)
```

These three commands fetch the changes from the upstream repo, ensure that you're on your local, forked repo on a new branch, and then merges the changes from the upstream repo into your forked repo.

Contributing changes to upstream

After you make changes and publish them to a new branch in your forked repository, you're ready to suggest your changes to the original owner. If you go to the original, upstream repo on GitHub.com, your branch shows up on the home page, and GitHub asks whether you want to open a pull request to merge the changes with the original repo (see Figure 6-3).

On your forked repo on GitHub.com, your branch shows up, and GitHub asks whether you want to create a pull request for it (see Figure 6-4).

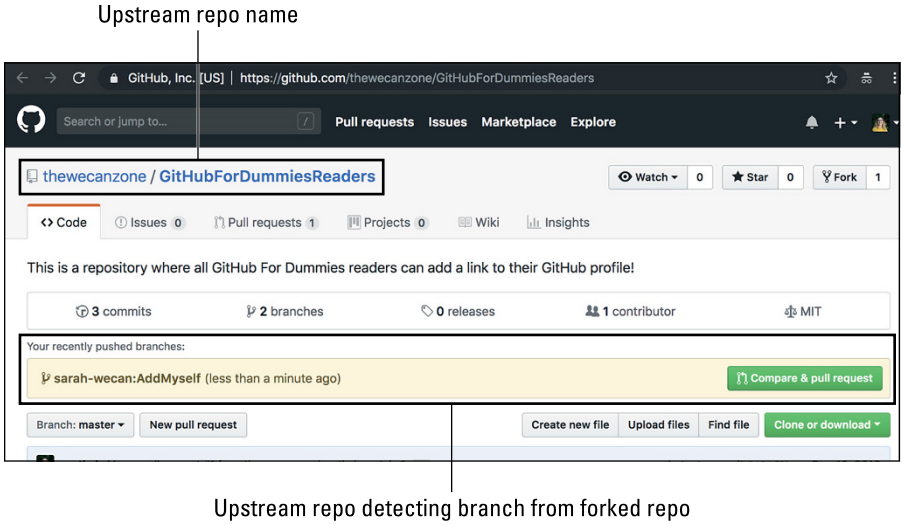

FIGURE 6-3: Original, upstream repo detecting a new branch from a forked repo.

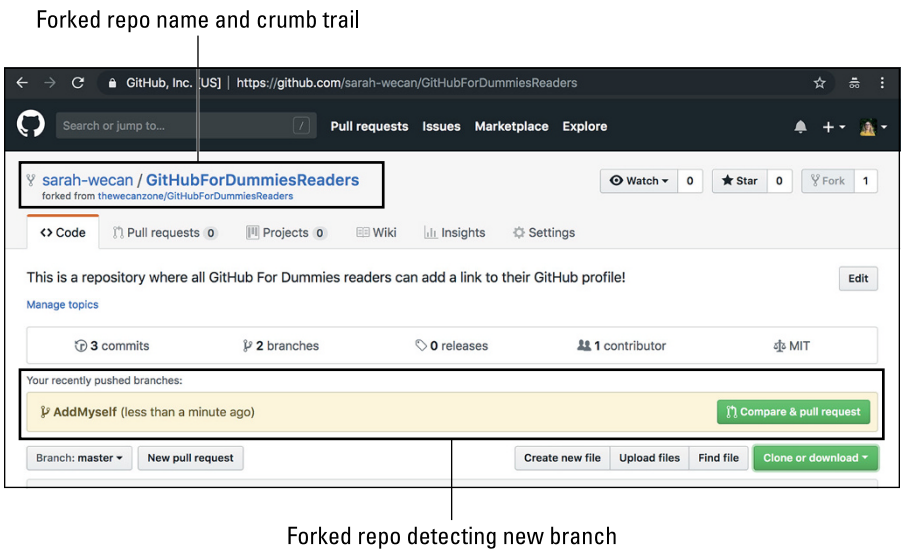

FIGURE 6-4: Forked repo detecting a new branch.

Click the Compare & Pull Request button, and a pull request creation page gives you the option to request to merge your changes with the upstream repo or your forked repo (see Figure 6-5). Choose the upstream repo, add a comment, and click the Create Pull Request button.

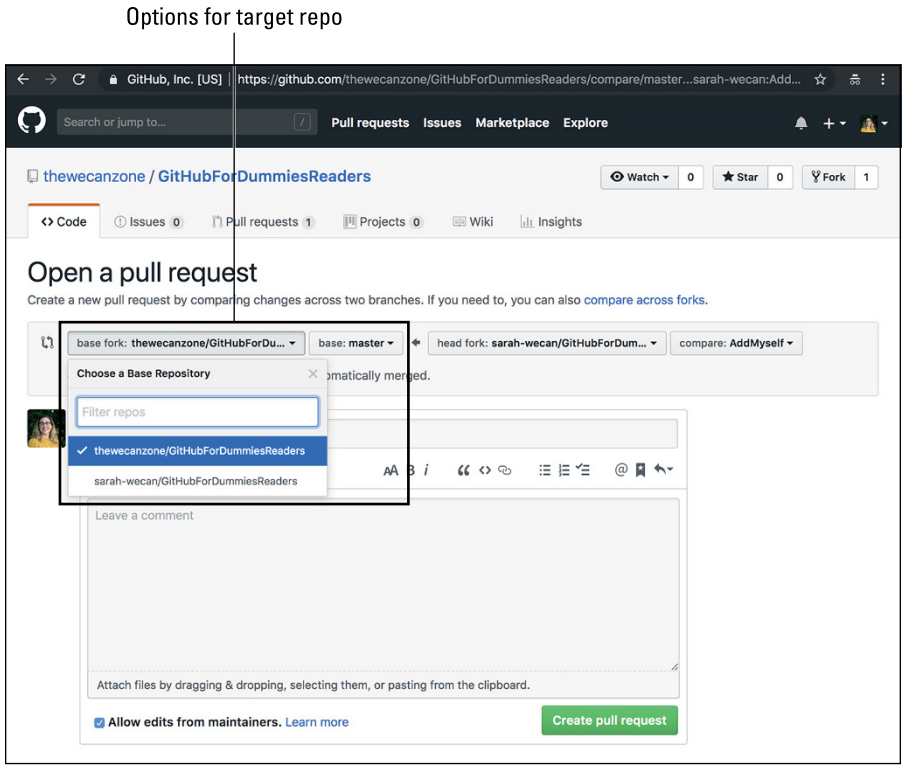

FIGURE 6-5: Forked repo detecting a new pull request with the option for upstream or forked.

You see that the branch can be merged, but you have no way to personally merge the pull request because you aren't the owner of the target branch (upstream, original repo); only the owner (or a specified collaborator) has permission to merge code. Figure 6-6 shows the pull request on your repo without the option to merge.

As the owner of the upstream, original pull request, I can see the pull request and have the option to merge it (see Figure 6-7). If you're creating a pull request on this repo, I will continually merge pull requests so that I can keep an up-to-date table of all the *GitHub For Dummies* readers!

TIP

If you have a lot of changes that you want to add to your fork before requesting that they get merged into the upstream, original repo, then you can first create the pull request to target your forked repo instead of the upstream repo. This is a change in what is shown in Figure 6-5. When you're ready to merge your changes into upstream, you can create a new pull request to request the target of the merge be the upstream repo.

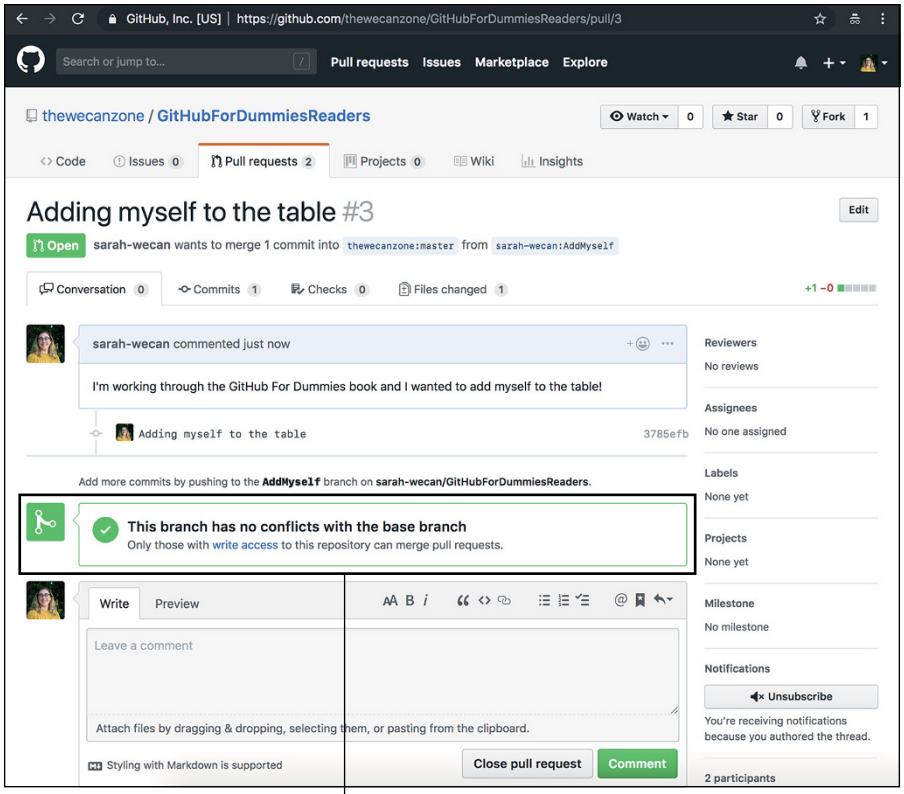

FIGURE 6-6: Pull request without the option to merge.

Getting unstuck when cloning without forking

One common problem people run into is they forget to fork a repository before they try to contribute to it. The following scenario describes one example of getting into this situation.

Here's the scenario: You clone a repository onto your local computer, modify the code, commit changes to `main`, and are ready to push your changes. But then you get a scary-looking error message. You may get the message in VS Code (see Figure 6-8), GitHub Desktop (see Figure 6-9), or in the terminal:

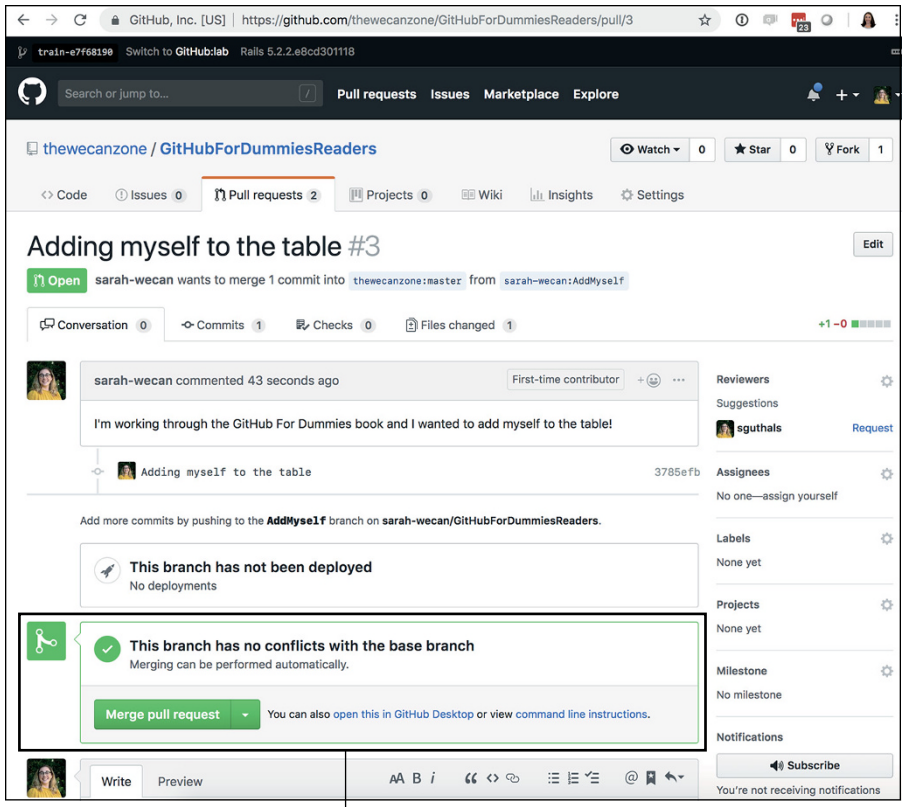

FIGURE 6-7: Pull request with the option to merge.

```
$ git push origin main
remote: Permission to thewecanzone/GitHubForDummiesReaders.git
  denied to dra-sarah.
fatal: unable to access 'https://github.com/thewecanzone/
  GitHubForDummiesReaders.git/': The requested URL returned
  error: 403
```

The error message tells you that you don't have permission to push to this repository. You should have forked the repository first. You also made the mistake of committing directly to the `development` branch. As I recommend elsewhere in the book, it's a good practice to make all your changes in a temporary branch; this applies to any protected branches such as `main` or `development`.

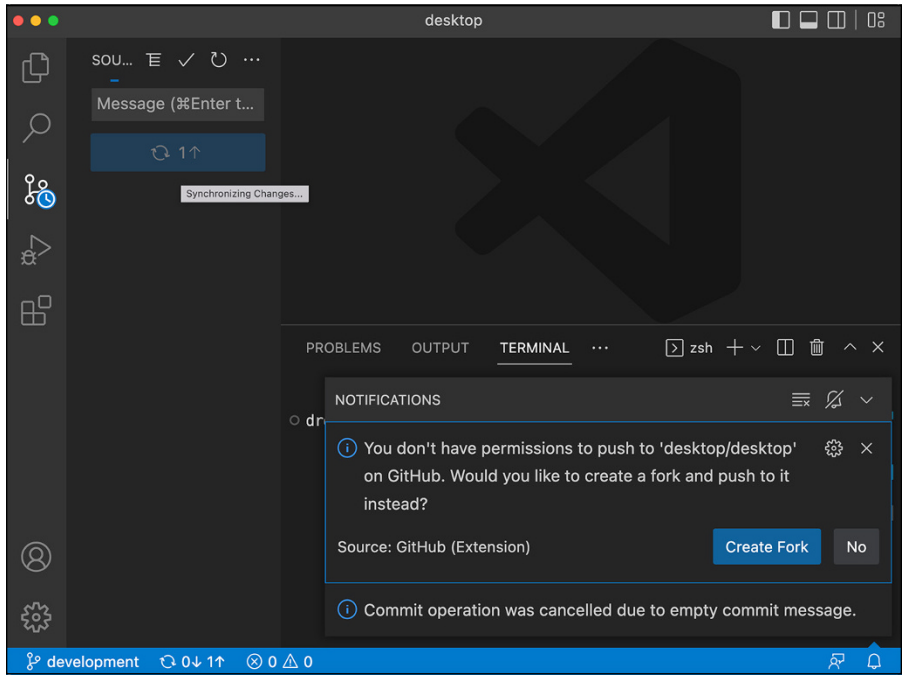

FIGURE 6-8: Push permission error message in VS Code.

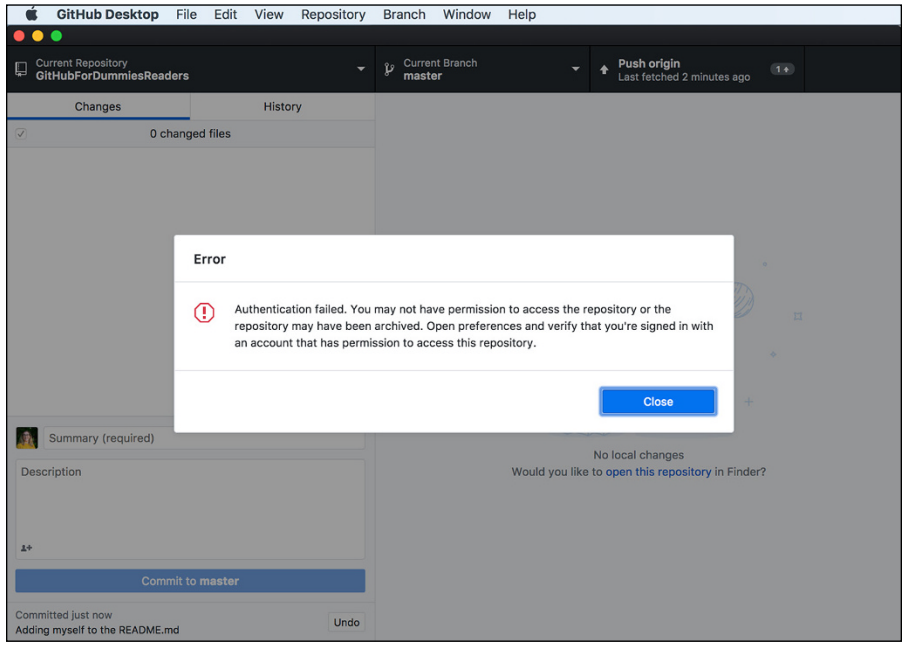

FIGURE 6-9: Push permission error message in GitHub Desktop.

To fix this mistake, you need to move your changes to a new branch, fork the repo, change the remote URLs for your local repository to point to your fork, and push your changes. This process can get tricky, but if you don't follow the VS Code prompts, these steps can help you out of this predicament from the terminal:

1. **Migrate your changes to a new branch.**

   Right when you discover you're targeting the incorrect remote repository, you should move your changes to a new branch. You don't want to accidentally pull in changes from the upstream, original branch onto all the hard work you just finished. This step can get tricky, but luckily there's a Git alias to help. See the nearby sidebar "Creating a Git Alias" for help. After you have the `git migrate` alias, go to the directory where your repo is in your terminal and type

   ```
   $ git migrate new-branch
   Switched to a new branch 'new-branch'
   Branch 'main' set up to track remote branch 'main' from
     'origin'.
   Current branch new-branch is up to date.
   ```

   Confirm that the new branch has been created:

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


TIP

   You can also confirm that your commits are only in this new branch and no longer in the old branch by running a `log` command to compare the two branches:

   ```
   $ git log main..new-branch --oneline
   ```

   This lists the commits in `new-branch` that are not in `main`. The `--oneline` flag prints each commit on a single line, which is useful when you just need a summary of commits and not the full details.

2. **Set the upstream remote to be the original GitHubForDummiesReaders repo.**

   To add an upstream remote to your repo, go to the terminal and type

   ```
   $ git remote add upstream https://github.com/thewecanzone/
     GitHubForDummiesReaders.git
   ```

Confirm that the upstream remote was added correctly:

```
$ git remote –v
originhttps://github.com/thewecanzone/
  GitHubForDummiesReaders.git (fetch)
originhttps://github.com/thewecanzone/
  GitHubForDummiesReaders.git (push)
upstreamhttps://github.com/thewecanzone/
  GitHubForDummiesReaders.git (fetch)
upstreamhttps://github.com/thewecanzone/
  GitHubForDummiesReaders.git (push)
```

3. **Fork the repo.**

   Back on GitHub.com, go to the original repo and click Fork at the top right of the repo home page. The page refreshes, and you see your own version of the repo, referencing the original repo (refer to Figure 6-2).

4. **Set the origin remote to be your forked repo.**

   After you have your own fork of the repo, you can change your remote origin to be your version:

```
$ git remote set–url origin https://github.com/dra–sarah/
  GitHubForDummiesReaders.git
```

   You can also confirm that all your remote URLs are correctly set:

```
$ git remote –v
originhttps://github.com/dra–sarah/GitHubForDummiesReaders.
  git (fetch)
originhttps://github.com/dra–sarah/GitHubForDummiesReaders.
  git (push)
upstreamhttps://github.com/dra–sarah/GitHubForDummiesReaders.
  git (fetch)
upstreamhttps://github.com/dra–sarah/GitHubForDummiesReaders.
  git (push)
```

5. **Push your branch to your forked version.**

   You're now in the same state that you would be in had you forked the repo before cloning. Back in VS Code, you can publish your branch.

6. **Create a pull request.**

   Your forked repo detects a new branch and offers to have you create a pull request (refer to Figure 6-4).

CREATING A GIT ALIAS

A *Git alias* is an easy way to automate and extend Git commands. If you're doing a lot of Git commands on the terminal, creating Git aliases can make your software development more efficient. For example, in your terminal you can type

```
$ git config --global alias.st status
```

Now, instead of typing `git status`, you can type `git st`, and Git returns the current status of your repository. Getting rid of just four letters may seem a little silly, but it can end up making your Git command experience a lot more efficient over time.

A Git alias is a lot more powerful than just reducing the number of keys you have to press. You can read about a tricky scenario at `https://haacked.com/archive/2015/06/29/git-migrate` where you have to migrate the commits you've made on a branch to another branch. This migration is critical if you get stuck in the position where you've started working on a clone of a repository where you don't have write permissions, as I discuss in the section "Getting unstuck when cloning without forking," earlier in this chapter.

The Git alias to migrate commits from one branch to another is complex. It's a few complicated steps all rolled into one simple `git migrate` command. To make this command accessible for you to use when you get stuck on an unforked clone, follow these steps in your terminal:

```
$ open ~/.gitconfig
$
```

Your `.gitconfig` file opens in your default editor. Add the following code to the bottom of your `.gitconfig` file:

```
[alias]
migrate = "!f(){ CURRENT=$(git symbolic-ref --short HEAD); git
   checkout -b $1 && git branch --force $CURRENT ${3-$CURRENT@
   {u}} && git rebase --onto ${2-master} $CURRENT; }; f"
```

If your `.gitconfig` file already has an `[alias]` section, don't retype that line. Save and close the `.gitconfig` file.

Now you can use the `git migrate` command to migrate commits from one branch to another branch! This `Git` command has one required parameter and two optional parameters:

```
git migrate <new-branch-name> <target-branch> <commit-range>
```

(continued)

(continued)

The parameter `<new-branch-name>` is required. This branch is where you move the commits to. If you don't specify anything else, then the `migrate` command moves all commits from the `main` branch `to` this new branch.

The parameters `<target-branch>` and `<commit-range>` are optional. `<target-branch>` allows you to move commits from a branch other than the `main` branch to the `<new-branch-name>` that you specify in the first parameter. `<commit-range>` allows you to specify which commits you want to move over. This parameter can be useful if you accidentally made one commit on the wrong branch, and you just want to move that one commit over to `<new-branch-name>`.

IN THIS CHAPTER

- Committing code in a terminal
- Creating a good commit
- Writing a commit message
- Committing other tools

Chapter 7
Writing and Committing Code

In this chapter, you write and commit code. The first part, writing code, is a very broad topic — too broad to be covered in this (or any single) book. The code I write in this chapter sets the stage for covering how to create good commits. Most of this chapter focuses on committing code. No matter what kind of code you write, the act of committing that code remains the same.

The code example I use throughout this chapter may seem contrived and overly simplistic. That's because it is contrived and simple. Don't let the simplicity, though, distract you because the information in this chapter also applies to large code bases.

Creating a Repository

A *commit* is the smallest unit of work with Git. It represents a small logical group of related changes to the repository. A commit additionally represents a snapshot in time — the state of the entire repository can be represented by referencing a single commit.

Before writing code, you need to create a local repository to store the code. In the following examples, I create a repository in a directory named `best-example`. Feel free to change `best-example` to a directory of your choice. Fortunately, this process is quick and painless:

1. **Open the terminal on your computer.**

   If you don't know how to do so, see Chapter 1 for guidance.

2. **Go to the directory where you want your project folder to be stored and type the following commands:**

   ```
   $ git init best-example
   $ cd best-example
   ```

   The first command creates an empty Git repository in the specified directory, `best-example`. Because the `best-example` directory doesn't already exist, Git creates it. The second command changes the current directory to this new directory.


TIP

Nearly every Git tutorial I've seen that covers initializing a Git repository does it in the current directory by calling `git init` with no parameters or `git init .` where the `.` represents the current directory. People can be forgiven for not realizing you can both create the repository directory and initialize it in one step by passing in the path to the new repository like I do here. In fact, you can combine both of these commands into a single command: `git init best-example && cd best-example`. This tip can help you gain the admiration and adulation of your less efficient peers!

Writing Code

After you're in a Git repository directory, you can start adding files. (If you aren't in a directory, see the previous section, "Creating a Repository" where I created the `best-example` directory.)

For this example, you create three files by typing the following code:

```
$ touch README.md
$ touch index.html
$ mkdir js
$ touch js/script.js
```

Note that one of the files you create is a `README.md` file. To find out why every repository should have a `README.md` file, see Chapter 10.

After running these commands, you have three files:

```
» README.md
» index.html
» script.js
```

`script.js` is in a subdirectory named `js`. You guessed it — you're making a simple website!

You can flesh out the `README.md` file first. In this example, I use VS Code to open and edit the files in the current directory. (If you need any guidance setting up VS Code, see Chapter 2.)


TIP

Make sure you have installed VS Code on your PATH. If you need help, you can follow the getting started guides at `https://code.visualstudio.com/docs/setup/setup-overview`.

You can add some simple Markdown text to the `README.md` document. *Markdown* is language that offers a simple way to format and style your text. You can check out a guide on Markdown on the GitHub guides `https://guides.github.com/features/mastering-markdown`.

Open the `README.md` in the editor by clicking in the file tree in VS Code. Then add some Markdown relevant to your project. In this example, add the following text:

```
# The Best Example Ever
Which will be a part of the best commit ever.
```

Then add the following code to `index.html`.

```
<!doctype html>
<html lang="en">
  <head>
        <meta charset="utf-8">
        <title>It is the cod3z</title>
        <script src="js/script.js"></script>
  </head>
  <body>
        <h1>The Best Cod3z!</h1>
  </body>
</html>
```

This HTML file references `script.js`. Open `script.js` in VS Code and add the following code.

```
document.addEventListener(
         "DOMContentLoaded",
         function(event) {
         alert('The page is loaded and the script ran!')
         }
);
```

REMEMBER

Make sure to save your changes to each file. Now test the code by opening `index.html` in your browser from the terminal.

```
$ open index.html
```

The page loads in your default browser, and the alert message, shown in Figure 7-1, appears.

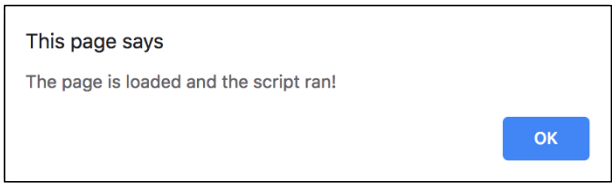

FIGURE 7-1: An alert message from my running code.

Creating a Commit

This section assumes you have code that you've changed on your local computer and that the code is in a working state. If you need an example of working code, see the previous section in this chapter to get to this state.

After you have running code, you can commit it to the repository. To create a commit is a two-step process:

1. **Stage the changes you want to commit.**
2. **Create the commit with a commit message.**

Staging changes

Staging changes can be confusing to the Git beginner. In concept, it's similar to a staging environment for a website. Staging changes is an intermediate place where you can see the changes you're about to commit before you commit them.

Why would you want to stage changes before committing them? In Git, a commit should contain a group of related changes. In fact, Git encourages this setup.

Suppose that you've been working for a few hours and now have a large set of unrelated changes that aren't committed to the Git repository.

You may be tempted to just commit everything with some generic commit message like "A bunch of changes." In fact, an XKCD comic makes light of this phenomena at `https://xkcd.com/1296`.

Committing a bunch of unrelated changes is generally a bad idea. The commit history of a repository tells the story of how a project changes over time. Each commit should represent a distinct cohesive set of changes. This approach to commits isn't just about being fastidious and organized. Having a clean Git history has concrete benefits.


TIP

One benefit of a clean Git history is that a command like `git bisect` is way more useful when each commit is a logical unit of work. The `git bisect` command is an advanced command, and full coverage of what it does is beyond the scope of this book. In short, though, `git bisect` provides a way to conduct a binary search through your Git history to find the specific commit that introduces a particular behavior, such as a bug. If every commit contains a large group of unrelated changes, finding the specific commit that introduces a bug isn't as useful as it would be if every commit contains a single logical unit of change.

In the example for this chapter, I can probably stand to create two commits:

- One that just contains the `README.md` file
- Another that contains the `index.html` and `script.js` files

Because the `index.html` file references the `script.js` file, checking in one without the other doesn't make sense at this point.

Start by staging the `README.md` file:

```
$ git add README.md
```

The `README.md` file is added to the Git index. The Git index is the staging area for creating commits to the repository. You can check the status of the repository to see that the file has been added to the index:

```
$ git status
On branch main
No commits yet
Changes to be committed:
(use "git rm --cached <file>..." to unstage)
        new file: README.md
Untracked files:
(use "git add <file>..." to include in what will be committed)
        index.html
        js/
```

As you can see, the `README.md` file is staged for commit. Meanwhile, the `index.html` and `js/` directory aren't yet tracked by this repository.


TIP

Why isn't `script.js` listed in the untracked files section? Git is taking a shortcut here. It notices that no files within the `js/` directory are tracked, so it can simply list the directory rather than list every file in the directory. In a larger code base, you'll be glad Git isn't listing every file in every subdirectory.

Committing a file

After you stage changes (see preceding section), you can create a commit. In this example, I use the `-m` flag with the `git` commit command to specify a short commit message. The following commands demonstrate how to create a commit and specify the commit message in one step:

```
$ git commit -m "Add a descriptive README file"
[main (root-commit) 8436866] Add a descriptive README file
1 file changed, 3 insertions(+), 0 deletions(-)
create mode 100644 README.md
```

The file is committed. If you run the `git status` command again, you see that you still have untracked files. The `git commit` command commits only the changes that are staged.

Committing multiple files

After you commit the first file, you're ready to stage the rest of the files for a commit.

```
$ git add -A
$ git status
On branch main
Changes to be committed:
(use "git reset HEAD <file>..." to unstage)
        new file: index.html
        new file: js/scripts.js
```

The `-A` flag indicates that you want to add all changes in the working directory to the Git index. When you run the `git status` command, you can see that you've staged two files.


TIP

When the `js` directory is untracked, `git status` lists only the `js` directory and none of the files in the directory. Now that you're trying to stage the `js` directory, Git lists the file in the `js` directory. Why the discrepancy? Git doesn't actually track directories. It tracks only files. Therefore, when you add a directory to a Git repository, it needs to add each file to the index.

Sometimes you need to write a more detailed commit message. In this example, I didn't specify a commit message when I run the commit command because I plan to write a more detailed commit message:

```
$ git commit
```

If you don't specify a commit message using the `-m` flag, Git launches an editor to create a commit message. If you haven't configured an editor with Git, it uses the system default editor, typically VI or VIM.

There are legions of jokes about how difficult it is to exit VIM, so I won't rehash them all here. I'll simply take a moment of silence in remembrance for friends still stuck in the VIM editor.


TIP

For the record, to exit VIM, press the ESC key to exit the edit mode and type `:wq` to exit and save or `:q!` to exit without saving.

To change the default editor to something like VS Code, run the following command in the terminal:

```
gitconfig --global core.editor"code--wait"
```


TIP

The editor opens a temporary file named `COMMIT_EDITMSG`, which contains some instructions that are commented out:

```
# Please enter the commit message for your changes. Lines starting
```

```
# with '#' will be ignored, and an empty message aborts the commit.
#
# On branch main
#
# Initial commit
#
# Changes to be committed:
# new file: README.md
#
# Untracked files:
# index.html
# js/
#
```

You enter your commit message in the file that gets opened. You can write your message before all the comments or simply replace everything in the file with your own commit message.

In this case, I replaced everything in that file with

```
Add index.html and script.js
This adds index.html to the project. This file is the
default page when visiting the website.
This file references js/script.js, which is also added
in this commit.
```

After you save the commit message and close the file or editor, Git creates a commit with the message you wrote.

Writing a Good Commit Message

What should you write in a commit message? What makes a good commit message?

A Git commit should contain a logical and cohesive change or set of changes. The message should describe that change in clear terms so that anyone who reads the message later understands what changed in the commit.

REMEMBER

The audience for the commit message are current and future collaborators on the project. Those collaborators may include yourself in the future. Someday you may be tracking down a bug and want to understand why you made some change. You'll thank past-you for writing a well-written commit message that answers that question. So write clear commit messages and be nice to future-you.

If you find that you have trouble describing a commit, it may be that the commit contains too many changes. In writing code, well written functions do one thing and do it well. Similarly, a commit should represent one change to the system. The commit message describes the change and why it's being made.

A good commit message should also follow a specific structure. In general, a commit message has two parts:

- The **summary** should be short (50 characters or less) and in the imperative present tense. For example, instead of writing "I added a method to Frobnicate widgets," write "Add method that Frobnicates widgets."
- The **description** provides detailed explanatory text, if needed. Not every change requires explanatory text. For example, if you rename a function, a commit message with just a summary of "Rename Frobnicate to Bublecate" may suffice. You should wrap the description text at 72 characters. This ensures it looks good when displayed in the terminal as part of the output from the `git log` command.

By convention, a new line character separates the summary from the description.

Here's an example commit message in an open source project `https://git.io/fhZ5a`:

```
Avoid potential race condition
In theory, if "ClearFormCache" is called after we
check `contains` but before we execute the `return`
line, we could get an exception here.
If we're concerned about performance here, we could
consider switching to the ConcurrentDictionary.
```

There are a few conventions you can use within a Git commit message that Git will ignore, but GitHub will recognize. For example, you can specify that a commit resolves a specific issue with something like "fixes #123" where 123 is the issue number. When a commit with this pattern is pushed to GitHub, the issue number is linked to the issue. And when the branch that contains that commit is merged into the default branch of the repository (typically `main`), GitHub closes the referenced issue. That's pretty handy!


TIP

You can also use emojis in a commit message. For example, one pattern some teams use is to indicate that a commit contains only cosmetic changes by prefacing it with `:art:`. When that commit is rendered on GitHub.com, GitHub renders the emoji. You can see this in action in Figure 7-2, which shows a list of commits from the GitHub for Visual Studio open source project `https://git.io/fhnDu`.

The "Using GitHub Conventions in Commit Messages" section also outlines many helpful things you can include in your messages.

FIGURE 7-2: A list of commit messages, some with emojis in the summary.

Committing Code with GitHub Desktop

Even though committing from the terminal is pretty straightforward, many people prefer to use a Graphical User Interface (GUI) application to commit code. Using a GUI has these benefits:

- A GUI can provide guidance on conventions with commit messages, such as keeping the summary to 50 characters and separating it from the description by new lines. A GUI can simply present two fields: summary and description.
- A GUI can provide support for GitHub specific conventions, such as the one where you can specify that a commit resolves an issue.

GitHub Desktop is a GUI created by GitHub that is great for committing code.

Tracking a repository in Desktop

Choose a repository that you have never opened in Desktop, but that you have locally on your computer. (See Chapter 2 if you haven't worked with Desktop yet.) If you need an example, use the `best-example` repository that you can create in the section "Creating a Repository," earlier in this chapter. When you launch Desktop, the `best-example` repository isn't listed in the list of repositories. Desktop doesn't scan your computer for Git repositories to manage. Instead, you have to tell Desktop about each repository you want to manage.

As expected, if you use Desktop to clone or create the repository, it's already tracking it. But sometimes you have a repository that you cloned or created outside of Desktop — for example, I created `best-example` using the terminal. Now you need to tell Desktop to track the repository you have chosen. Fortunately, this task is easy from the terminal.

The Desktop command line tool allows you to launch Desktop from your terminal, which allows you to easily integrate Desktop as much or as little as you want into your existing terminal-based Git workflow.

On Windows, you don't need to install the command line tool; it's done automatically. On the Mac, you have to take a separate step.

To install the command line tool on a Mac:

1. **Make sure Desktop is the active application and then, in the application menu bar, choose GitHub Desktop➪ Install Command Line Tool.**

2. **From the terminal, make sure that you're in the repository you want Desktop to track.**

   For this example, I'm in the `best-example`.

3. **Run the following command:**

   ```
   $ github .
   ```

   The `.` in the command represents the current directory. It could, instead, be a fully qualified path to a directory. GitHub Desktop launches (if it's not already running) and opens the specified directory. Because the current directory is already a Git repository, Desktop adds it to the list of repositories that it tracks. It then sets this repository as the current repository so that you can browse the repository's history, switch branches, and create commits, as shown in Figure 7-3.

If the current directory wasn't a repository, Desktop prompts you to create a Git repository in that directory. How convenient!

Publishing a repository in Desktop

To experience the full power of Desktop's integration with GitHub, you need to publish this repository to GitHub:

1. **Clicking the Publish repository button.**

   A dialog box to publish the repository appears (see Figure 7-4).

2. **Fill in the details and click the Publish Repository button.**

   The repository is created on your GitHub.com account.

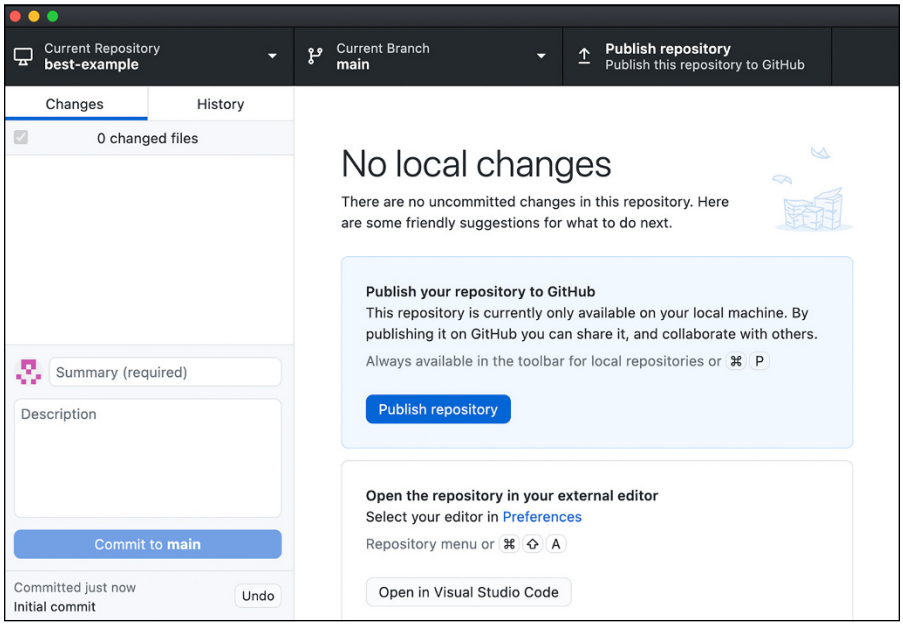

FIGURE 7-3: GitHub Desktop opened to the `best-example` repository.

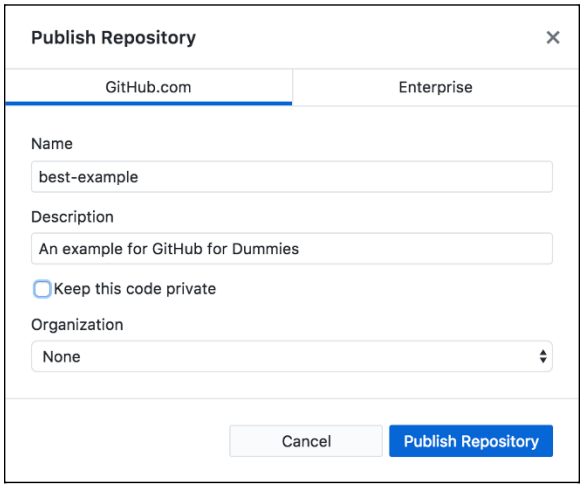

FIGURE 7-4: The Publish dialog box used to publish a repository to GitHub.com.

TIP

Desktop provides a keyboard shortcut to open the browser to the repository: ⌘-Shift-G (Ctrl+Shift+G on Windows).

If you want, you can create a few issues in the repository. (See Chapter 3 to find out how to create issues.) For this example, I created five issues:

- Provide more details on the README.

- Mention in the README that this is a collaborative effort.
- Add a contribution section to README.
- Do not use an alert message.
- Set up website alerts.

You can also see these issues at this repo: `https://github.com/FakeHaacked/best-example/issues`.

Committing in Desktop

Desktop is used only for Git operations. To edit the files in the repository, you still need to use your editor of choice.

Make some changes so you have something to commit. In the example for this chapter, you can make some changes to `index.html` shown in bold.

```
<!doctype html>
<html lang="en">
        <head>
                <meta charset="utf-8">
                <title>The Best Example</title>
                <script src="js/script.js"></script>
        </head>
        <body>
                <h1>The Best Cod3z!</h1>
                <div id="message"></div>
        </body>
</html>
```

Update `script.js` to populate the new `DIV` element, like civilized people would, rather than use an alert message. Changes are in bold:

```
document.addEventListener(
        "DOMContentLoaded",
        function(event) {
        var message = document.getElementById('message')
        message.innerText = 'The script ran!'
        }
);
```

Switch back to Desktop and click the Changes tab, shown in Figure 7-5.

The left pane lists the set of changed files. If you select a file, you can see the

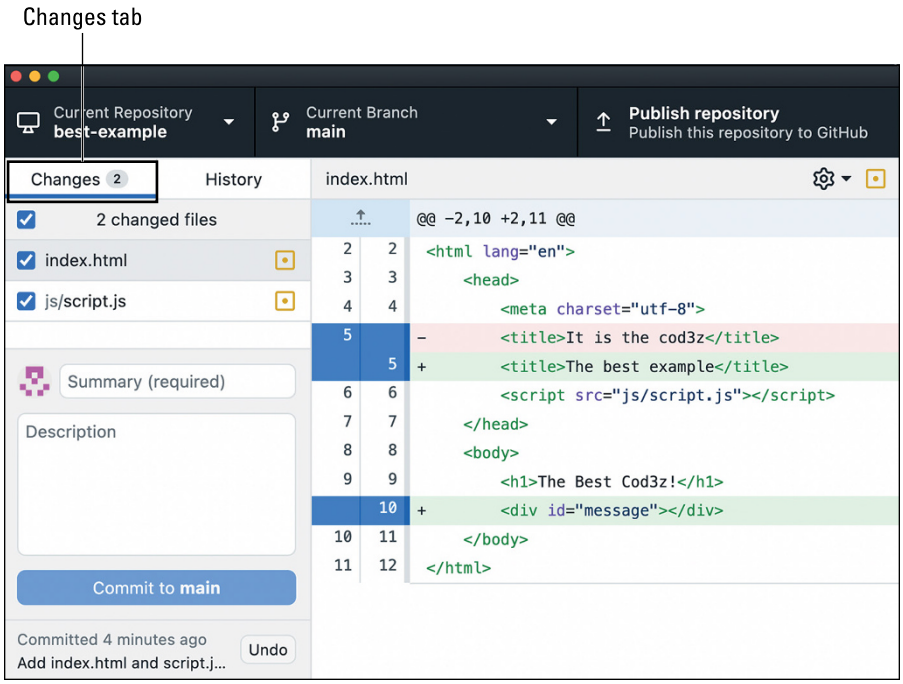

FIGURE 7-5: The Changes View showing the uncommitted changes.

specific changes to that file in the right pane, which is called the *diff view*.

You can commit all the changes as a single commit, but sometimes you might have unrelated changes. In this example, I have two unrelated changes:

- The change to the title in `index.html`
- The message changes to both `index.html` and `script.js`

Each of these changes should be in their own commit. How do you do that when `index.html` contains two unrelated changes? Fortunately, Desktop provides a nice way to commit a portion of the changes in a file. This process is known as a partial commit:

1. **Deselect all changes.**

   In the left pane, uncheck the check box next to the label `2 changed files` to deselect all changes.

2. **Select index.html in the left pane.**

Click the filename in the left pane to display the changes for `index.html`.

3. **Select the title changes.**

In the diff view, click the line numbers in the gutter to select the changes you want to keep. To select a whole code block, click the thin line just to the right of the line number. Select the code block next to line 5 by clicking the thin line next to line 5. After you select the code block, both lines labeled line 5 should be selected (selected lines show up as blue), as shown in Figure 7-6.

WARNING

You may be confused about why two lines are labeled 5 in the diff view. The numbers on the left represent what the file was originally named before you made the changes. The lines on the right represent the lines of the changed lines. Because I changed line 5, it's listed twice. Line 10 is a new line that didn't exist before, so it is listed only on the right.

4. **With those lines selected, enter a commit message and then click the Commit to Main button.**

As you can see in the bottom left portion of Figure 7-6, Desktop provides two fields for commit messages. Go ahead and enter **Change the title** into the summary and click the Commit to Main button.

Notice that the diff view updates to have the change only on line 10 (see Figure 7-7). That's because I committed the change on line 5.

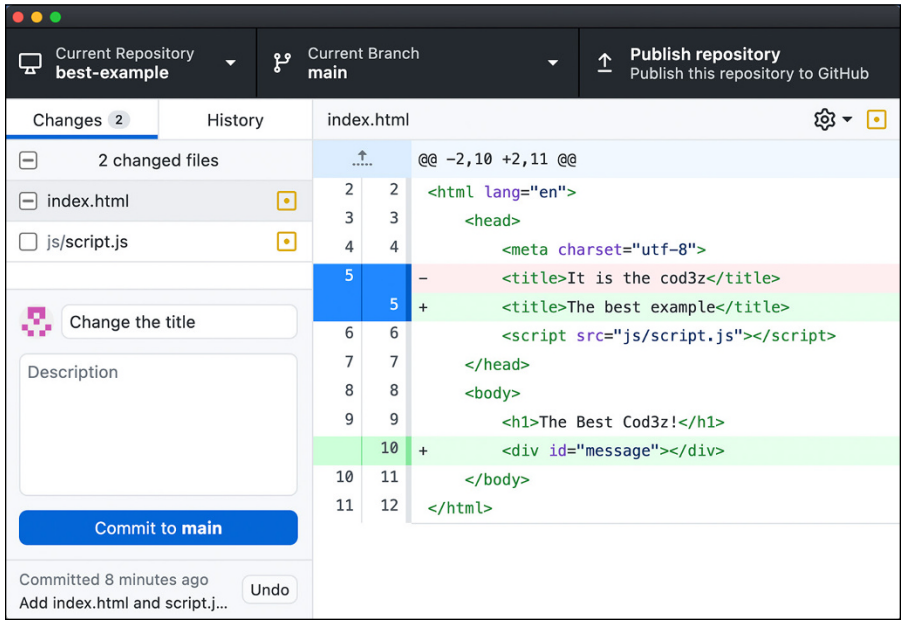

FIGURE 7-6: The diff view with one change selected.

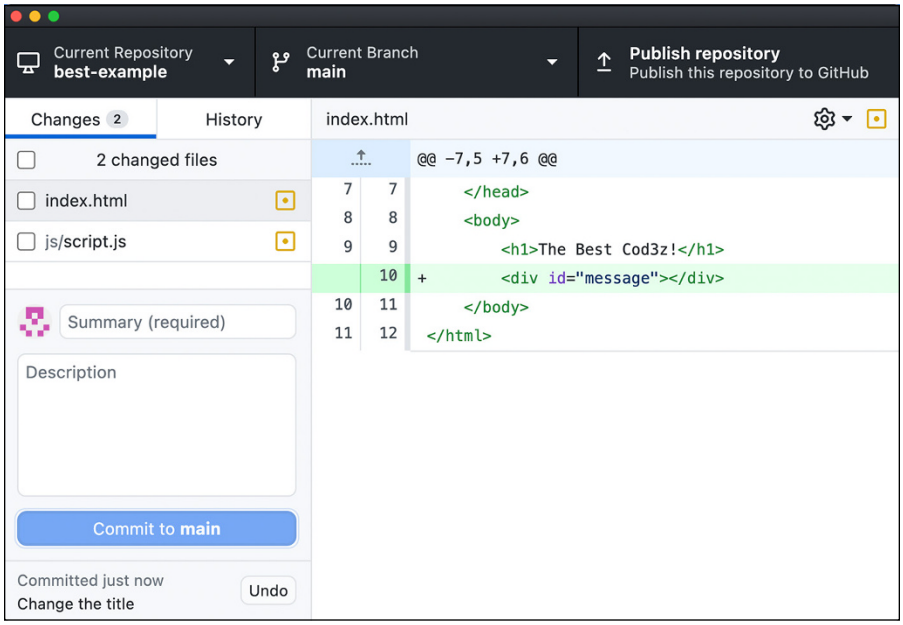

FIGURE 7-7: The Changes tab after a partial commit.

If you're following the example in this chapter, make sure all the remaining changes are selected by clicking the check box next to the label `2 changed files` until the check box is selected.

Using GitHub Conventions in Commit Messages

You can enhance your commit messages with GitHub-specific features, such as emojis, issue references, and coauthor credits.

Emojis

Emojis are little images or icons that convey an emotion or concept. Widely used on GitHub.com, emojis can bring a bit of levity and whimsy to an otherwise serious occupation.

In the commit summary box, you can initiate the emoji picker by typing the `:` character. If you keep typing, you can list all emojis that start with the letters you type. For example, Figure 7-8 lists all emojis that start with `ar` as the result of typing **:ar**.

FIGURE 7-8: The emoji picker listing emojis.

You can select the one you want with the arrow keys and then press Tab to complete it. Desktop then fills in the full text of the emoji, which in this case is `:art:`.

Issue references

GitHub also lets you reference an issue in a commit message with the format #123 where 123 is the issue number. Desktop has support for looking up an issue when writing a commit message. To try this out, create an issue ahead of time so that you can reference the issue in a commit message. As an example, I created an issue that describes the need to test the greeting created by a GitHub Action for new contributors to this repository. I reference that issue in this commit message.

To reference an issue in a commit message:

1. **In the commit description field, type** Fixes #.

   A few recent issues appear. If you don't see the issue you want to reference and you don't remember the issue number, you can start typing a word that's in the issue that you remember. For example, when I type **# greetings** an issue pops up (see Figure 7-9).

2. **Select the issue you want to reference and press Tab.**

   In this example, I selected issue 15. Desktop replaces `#greetings` with `#15`.

TIP

Giving credit to coauthors

Git doesn't support multiple authors directly. However, the Git community created a convention for specifying multiple coauthors in a commit that is now supported by GitHub.

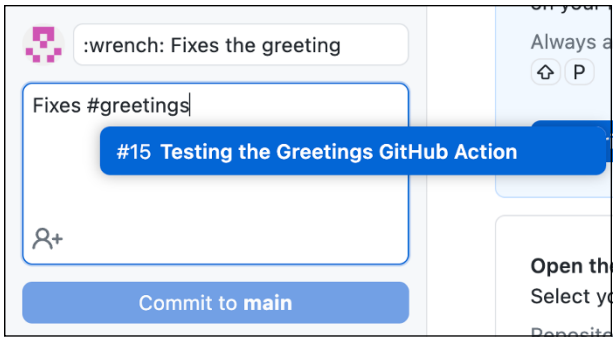

FIGURE 7-9: A list of issues with the word "greetings" in them, in this case there is only one.

To give credit to coauthors:

1. **With the Desktop open, in the commit box with the Description label, click the little icon with a person and a plus sign in the bottom-left corner.**

   Desktop adds a textbox to enter a coauthor's GitHub username.

2. **Click the @ symbol to see a list of potential users, as shown in Figure 7-10.**

   GitHub lists only users who have access to the repository — for example, collaborators and org members (if the repository belongs to an organization).

   Just like the issue selector, you can also search by first name, last name, or username by appending a bit after the @. Press Tab, and Desktop replaces whatever you typed so far with the selected user's full username.

3. **To create the commit, click the Commit to Main button.**

TIP

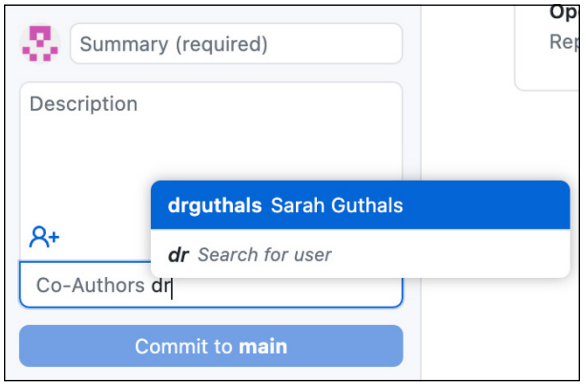

FIGURE 7-10: A list of potential coauthors.

To see your commit, click the History tab and click the commit you just created (see Figure 7-11).

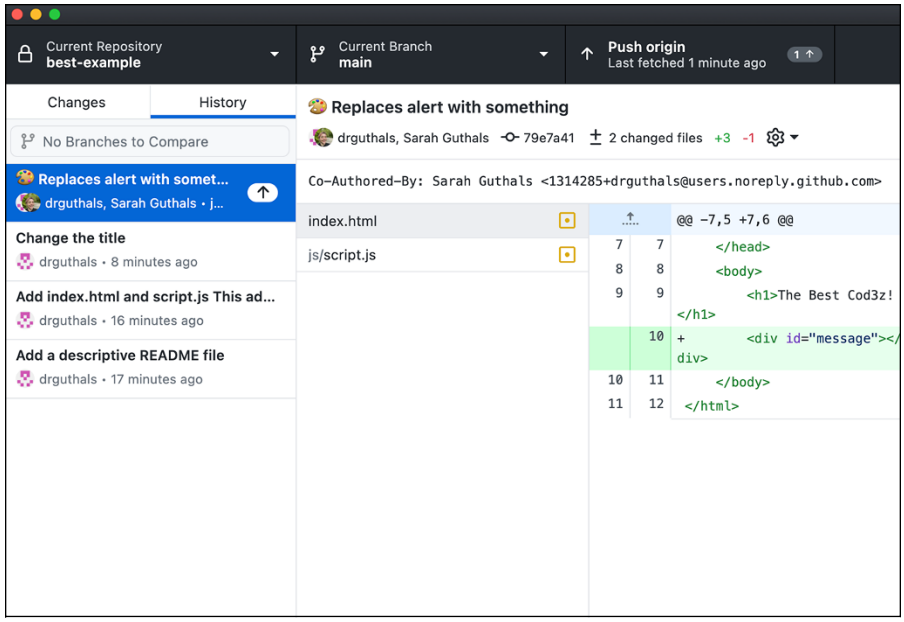

FIGURE 7-11: The newly created commit.

You can see in the commit message in the right pane that my username was replaced by the line

```
Co-Authored-By: Sarah Guthals <1314285+drguthals@users.noreply.github.com>
```

REMEMBER

That's the actual convention for specifying coauthors in Git commit messages. By using Desktop, you don't have to remember the exact format. You can just specify a username and let Desktop handle the rest.

Committing Code from Your Editor

Many editors have built-in support for committing code. Built-in support allows you commit code without having to switch to another application. The downside is that different editors have different levels of support for the various conventions you can use in a commit message.

But for quick and dirty commits, built-in support is very useful. Covering how every editor supports Git commits is out of the scope of this book, but you can see this in action with VS Code in Chapter 5. For other editors, refer to their specific documentation.

FOR MORE READING

A lot of great guidance is out there for writing good commits. For example, the Git documentation at `https://git-scm.com/book/en/v2/Distributed-Git-Contributing-to-a-Project` has a section on contributing to a project and includes some Commit Guidelines.

I'm also a fan of a blog post by Chris Beams entitled "How to Write a Git Commit Message," which you can read at `https://chris.beams.io/posts/git-commit/`.